SYBASE iAnywhere

SQL Anywhere® 服务器
SQL 的用法

2009 年 2 月

11.0.1 版

版权和商标

目录

关于本书

主题

本手册介绍如何设计和创建数据库；如何导入、导出和修改数据；如何检索数据以及如何建立存储过程和触发器。

读者

本书适用于所有 SQL Anywhere 用户。

开始之前

本手册假定您了解有关数据库管理系统（特别是 SQL Anywhere）的基础知识。如果您对此还不熟悉，应考虑先阅读 *SQL Anywhere 11 - 简介*，然后再阅读本手册。

关于 SQL Anywhere 文档

完整的 SQL Anywhere 文档以四种形式提供，但所包含信息均相同。

- **HTML 帮助** 联机帮助文档包含完整的 SQL Anywhere 文档，其中包括手册和 SQL Anywhere 工具的上下文相关帮助。

  如果使用 Microsoft Windows 操作系统，则联机帮助文档以 HTML 帮助 (CHM) 格式提供。若要访问此文档，请选择 [开始] » [程序] » [**SQL Anywhere 11**] » [文档] » [联机手册]。

  管理工具使用同一联机文档来实现帮助功能。

- **Eclipse** 在 Unix 平台上以 Eclipse 格式提供完整的联机帮助。要访问文档，请从 SQL Anywhere 11 安装的 *bin32* 或 *bin64* 目录下运行 *sadoc*。

- **DocCommentXchange** DocCommentXchange 是一个用于访问和讨论 SQL Anywhere 文档的社区。

  使用 DocCommentXchange 可以执行以下任务：

  - 查看文档
  - 检查是否有用户对文档各部分所做出的阐明
  - 提供建议和修正意见以在将来的版本中为所有用户改进文档

  访问 http://dcx.sybase.com。

- **PDF** 整套 SQL Anywhere 手册会以一组 Portable Document Format (PDF) 文件的形式提供。您必须有 PDF 阅读器才能查看信息。要下载 Adobe Reader，请访问 http://get.adobe.com/reader/。

  若要在 Microsoft Windows 操作系统上访问 PDF 文档，请选择 [开始] » [程序] » [**SQL Anywhere 11**] » 文档 » [联机手册 - PDF 格式]。

  要在 Unix 操作系统上访问 PDF 文档，请使用 Web 浏览器打开 *install-dir/documentation/zh/pdf/index.html*。

关于文档集中的手册

SQL Anywhere 文档由以下手册组成：

- **SQL Anywhere 11 - 简介** 本手册介绍 SQL Anywhere 11，一个提供数据管理和数据交换技术的综合数据包，通过它可以为服务器环境、台式机环境、移动环境以及远程办公环境快速开发由数据库驱动的应用程序。

- **SQL Anywhere 11 - 更改和升级** 本手册介绍 SQL Anywhere 11 以及该软件以前版本中的新功能。

- **SQL Anywhere 服务器 - 数据库管理** 本手册介绍如何运行、管理及配置 SQL Anywhere 数据库。它介绍了数据库连接、数据库服务器、数据库文件、备份过程、安全性、高可用性、使用复制服务器进行复制以及管理实用程序和选项。

- **SQL Anywhere 服务器 - 编程** 本手册介绍如何使用 C、C++、Java、PHP、Perl、Python 和 .NET 编程语言（例如 Visual Basic 和 Visual C#）建立和部署数据库应用程序。其中介绍了各种编程接口，如 ADO.NET 和 ODBC。
- **SQL Anywhere 服务器 - SQL 参考** 本手册提供了系统过程和目录（系统表和视图）的参考信息。也介绍了 SQL 语言（搜索条件、语法、数据类型和函数）的 SQL Anywhere 实现。
- **SQL Anywhere 服务器 - SQL 的用法** 本手册介绍如何设计和创建数据库；如何导入、导出和修改数据；如何检索数据以及如何建立存储过程和触发器。
- **MobiLink - 入门** 本手册介绍基于会话的关系数据库同步系统 MobiLink。MobiLink 技术支持双向复制并且非常适用于移动计算环境。
- **MobiLink - 客户端管理** 本手册介绍如何设置、配置和同步 MobiLink 客户端。MobiLink 客户端可以是 SQL Anywhere 或者 UltraLite 数据库。本手册同时也介绍了 Dbmlsync API，通过它可无缝地将同步集成到 C++ 或 .NET 客户端应用程序中。
- **MobiLink - 服务器管理** 本手册说明如何设置和管理 MobiLink 应用程序。
- **MobiLink - 服务器启动的同步** 本手册介绍 MobiLink 服务器启动的同步，这种功能允许 MobiLink 服务器启动同步或在远程设备上进行操作。
- **QAnywhere** 本手册介绍 QAnywhere，一个用于移动、无线、台式机和膝上型客户端的消息传递平台。
- **SQL Remote** 本手册介绍用于移动计算的 SQL Remote 数据复制系统，此系统支持使用电子邮件或文件传输等间接链接共享 SQL Anywhere 统一数据库和多个 SQL Anywhere 远程数据库之间的数据。
- **UltraLite - 数据库管理和参考** 本手册介绍适用于小型设备的 UltraLite 数据库系统。
- **UltraLite - C 及 C++ 编程** 本手册介绍 UltraLite C 和 C++ 编程接口。利用 UltraLite，可以开发数据库应用程序，并将它们部署到手持式设备、移动设备或嵌入式设备。
- **UltraLite - M-Business Anywhere 编程** 本手册介绍 UltraLite for M-Business Anywhere。利用 UltraLite for M-Business Anywhere，用户可以开发基于 Web 的数据库应用程序，并将它们部署到运行 Palm OS、Windows Mobile 或 Windows 的手持式设备、移动设备或嵌入式设备。
- **UltraLite - .NET 编程** 本手册介绍 UltraLite.NET。利用 UltraLite.NET，您可以开发数据库应用程序，并将它们部署到计算机、手持式设备、移动设备或嵌入式设备。
- **UltraLiteJ** 本手册介绍 UltraLiteJ。利用 UltraLiteJ，可以在支持 Java 的环境中开发和部署数据库应用程序。UltraLiteJ 支持 BlackBerry 智能手机和 Java SE 环境。UltraLiteJ 基于 iAnywhere UltraLite 数据库产品。
- **错误消息** 本手册提供了 SQL Anywhere 错误消息及其诊断信息的完整列表。

文档约定

本节列出了本文档中使用的约定。

操作系统

SQL Anywhere 可以在各种平台上运行。在大多数情况下，该软件在所有平台上的行为都是相同的，但也有变动或限制。这些变动或限制通常基于基础操作系统（Windows、Unix），很少基于特定变型（AIX、Windows Mobile）或版本。

为了简化对操作系统的提及，本文档按如下方式对支持的操作系统进行分组：

- **Windows** Microsoft Windows 系列包括 Windows Vista 和 Windows XP（主要用于服务器、台式计算机和膝上型计算机），以及 Windows Mobile（用于移动设备）。

  除非另外指定，否则当本文档提及 Windows 时，是指所有基于 Windows 的平台，包括 Windows Mobile。

- **Unix** 除非另外指定，否则当本文档提及 Unix 时，是指所有基于 Unix 的平台，包括 Linux 和 Mac OS X。

目录和文件名

大部分情况下，对目录和文件名的引用在所有支持的平台上都是类似的，只需在不同形式之间进行简单的转换。这时需使用 Windows 约定。在细节更为复杂的情况下，文档显示所有相关形式。

下面是文档编写中用于简化目录和文件名的约定：

- **大写和小写目录名** 在 Windows 和 Unix 上，目录和文件名可以包括大写和小写字母。创建目录和文件时，文件系统会保留字母大小写。

  在 Windows 上，对目录和文件的提及*不区分大小写*。混合使用大小写的目录和文件名很常见，但使用所有小写字母来提及目录和文件的形式也很常见。SQL Anywhere 安装包含诸如 *Bin32* 和 *Documentation* 的目录。

  在 Unix 上，对目录和文件的提及*区分大小写*。混合使用大小写的目录和文件名不常见。大多数的目录和文件名全部使用小写字母。SQL Anywhere 安装包含诸如 *bin32* 和 *documentation* 的目录。

  本文档采用 Windows 形式的目录名。大多数情况下，在 Unix 上可以将大小写混合形式的目录名转换成小写字母的等效目录名。

- **分隔目录和文件名的斜线** 文档使用反斜线作为目录分隔符。例如，PDF 格式的文档位于 *install-dir\Documentation\zh\PDF*（Windows 形式）。

  在 Unix 上，用正斜线替换反斜线。PDF 文档位于 *install-dir/documentation/zh/pdf* 下。

- **可执行文件** 文档使用 Windows 约定显示可执行文件名（带有诸如 *.exe* 或 *.bat* 后缀）。在 Unix 上，可执行文件名没有后缀。

  例如，在 Windows 上，网络数据库服务器是 *dbsrv11.exe*。在 Unix 上是 *dbsrv11*。

- ***install-dir*** 在安装过程中，选择 SQL Anywhere 的安装位置。创建环境变量 SQLANY11，用来表示此位置。文档中以 *install-dir* 表示此位置。

  例如，本文档将此文件表示为 *install-dir*\readme.txt。在 Windows 上，这等同于 *%SQLANY11%\readme.txt*。在 Unix 上，这等同于 *$SQLANY11/readme.txt* 或 *${SQLANY11}/readme.txt*。

有关 *install-dir* 缺省位置的详细信息，请参见“SQLANY11 环境变量”一节 《SQL Anywhere 服务器 - 数据库管理》。

- ***samples-dir*** 在安装过程中，选择 SQL Anywhere 随附的示例的安装位置。创建环境变量 SQLANYSAMP11，用来表示此位置。文档中以 *samples-dir* 表示此位置。

  要在 *samples-dir* 中打开 Windows 资源管理器窗口，请在 [**开始**] 菜单中，选择 [**程序**] » [**SQL Anywhere 11**] » [**示例应用程序和项目**]。

  有关 *samples-dir* 缺省位置的详细信息，请参见“SQLANYSAMP11 环境变量”一节 《SQL Anywhere 服务器 - 数据库管理》。

命令提示符和命令 shell 语法

大多数操作系统都提供一种或多种使用命令 shell 或命令提示符来输入命令和参数的方法。Windows 命令提示符包括 Command Prompt（DOS 提示符）和 4NT。Unix 命令 shell 包括 Korn shell 和 bash。每个 shell 都具有一些功能，其能力不仅仅局限于简单命令。这些功能通过特殊字符来驱动。特殊字符和功能随 shell 的不同而不同。如果没有正确使用这些特殊字符，通常会导致语法错误或意外行为。

本文档以普通形式提供命令行示例。如果这些示例中包含 shell 的特殊字符，则命令需要根据特定 shell 进行修改。修改方法不在本文档所述范围之内，但通常是在包含这些特殊字符的参数两旁加上引号，或是在特殊字符前面使用转义字符。

下面是命令行语法的一些示例，不同的平台可能会有不同的形式：

- **括号和大括号** 有些命令行选项需要一个参数，该参数将以列表形式接受详细的值指定。该列表通常用括号或大括号括起来。本文档使用括号。例如：

  ```
  -x tcpip(host=127.0.0.1)
  ```

  如果括号导致出现语法问题，用大括号替代：

  ```
  -x tcpip{host=127.0.0.1}
  ```

  如果两种形式都将产生语法问题，应按照 shell 的要求，用引号将整个参数括起来：

  ```
  -x "tcpip(host=127.0.0.1)"
  ```

- **引号** 如果必须在参数值中指定引号，该引号可能会与用于括参数的引号的传统用法发生冲突。例如，要指定值中包含双引号的加密密钥，则可能必须用引号括起密钥，然后转义嵌入的引号：

  ```
  -ek "my \"secret\" key"
  ```

  在许多 shell 中，密钥的值为 `my "secret" key`。

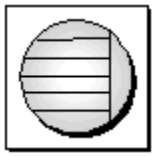
- **环境变量** 本文档介绍设置环境变量。在 Windows shell 中，环境变量使用语法 *%ENVVAR%* 来指定。在 Unix shell 中，环境变量使用语法 *$ENVVAR* 或 *${ENVVAR}* 来指定。

图标

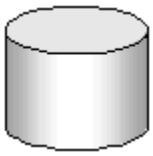
本文档中使用了下列图标。

- 客户端应用程序。

- 数据库服务器，如 Sybase SQL Anywhere。



- 数据库。在某些高水平的图中，可以使用此图标表示数据库和管理该数据库的数据库服务器。



- 复制或同步中间件。用于帮助在数据库之间共享数据。例如 MobiLink 服务器和 SQL Remote 消息代理。

- 编程接口。

联系文档小组

我们欢迎您就本帮助文档提出意见、建议和反馈信息。

要提交意见和建议，请发送电子邮件到 SQL Anywhere 文档小组，地址为 iasdoc@sybase.com。虽然我们不对这些电子邮件进行回复，但您的反馈会帮助我们改进文档，因此我们真诚地欢迎您提出宝贵的意见和建议。

DocCommentXchange

也可以使用 DocCommentXchange 将意见或建议直接置于帮助主题中。DocCommentXchange (DCX) 是一个用于访问和讨论 SQL Anywhere 文档的社区。使用 DocCommentXchange 可以执行以下任务：

- 查看文档
- 检查是否有用户对文档各部分所做出的阐明
- 提供建议和修正意见以在将来的版本中为所有用户改进文档

访问 http://dcx.sybase.com。

查找详细信息并请求技术支持

附加信息和资源可从 Sybase iAnywhere 开发人员社区获得，网址是 http://www.sybase.com/developer/library/sql-anywhere-techcorner。

如果您有问题或是需要帮助，可将邮件发布到下面所列的 Sybase iAnywhere 新闻组。

当您向这些新闻组发布邮件时，请务必提供问题的详细信息，包括 SQL Anywhere 版本的内部版本号。可以通过运行以下命令找到此信息：**dbeng11 -v**。 **dbeng11 -v**.

新闻组位于 *forums.sybase.com* 新闻服务器上。

这些新闻组包括：

- sybase.public.sqlanywhere.general
- sybase.public.sqlanywhere.linux
- sybase.public.sqlanywhere.mobilink
- sybase.public.sqlanywhere.product_futures_discussion
- sybase.public.sqlanywhere.replication
- sybase.public.sqlanywhere.ultralite
- ianywhere.public.sqlanywhere.qanywhere

有关 Web 开发问题，请访问 http://groups.google.com/group/sql-anywhere-web-development。

新闻组免责声明

iAnywhere Solutions 没有义务为其新闻组提供解决方案、信息或建议，除提供系统操作员监控服务和确保新闻组的运行和可用性外，iAnywhere Solutions 也没有义务提供任何其它服务。

如果时间允许，iAnywhere 技术顾问以及其他员工也会对新闻组服务提供帮助。他们是在自愿的基础上提供帮助的，所以可能无法定期提供解决方案和信息。他们可以提供多少帮助取决于他们的工作量。

创建数据库

本节将介绍如何创建 SQL Anywhere 数据库。本节介绍如何使用数据库对象，例如表、视图、实例化视图、索引等。本节提供了有关如何使用键和约束维护参照完整性的信息，并说明了如何在不同的隔离级别下处理事务。

在 SQL Anywhere 中创建数据库

目录

设计注意事项

在 SQL Anywhere 中创建数据库之前，应做好如下准备工作：定义数据库将包含的所有表（实体）、每个表中要包含的列（属性），以及每个表与其它表之间的关系（键和约束）。

考虑为数据库构建一个概念数据库模型 (CDM)。CDM 可直观地将数据库以图的形式表示。可以在纸上构建 CDM，也可以使用 Sybase PowerDesigner 等软件构建。CDM 工具可帮助您在构建 CDM 时对您的设计进行校验，还可以帮助您创建数据库。

有关设计数据库（包括如何构建概念数据库模型）的详细信息，请访问 http://infocenter.sybase.com/help/index.jsp 上的 PowerDesigner 文档。

有关表和视图等数据库对象的详细信息，请参见“使用数据库对象”第 13 页。

选择对象名称

避免使用保留字命名数据库对象。有关 SQL Anywhere 保留字的列表，请参见“保留字”一节《SQL Anywhere 服务器 - SQL 参考》。

如果列名包含除字母、数字或下划线之外的字符、不是以字母开头或名称与关键字的名称相同，则必须将列名用双引号引起来。

选择列数据类型

SQL Anywhere 中提供以下数据类型：

- 整数数据类型
- 小数数据类型
- 浮点数据类型
- 字符数据类型
- 二进制数据类型
- 日期/时间数据类型
- 域（用户定义的数据类型）

有关数据类型的详细信息，请参见“SQL 数据类型”《SQL Anywhere 服务器 - SQL 参考》。

任何字符或二进制字符串数据类型（例如 CHAR、VARCHAR、LONG VARCHAR、NCHAR、BINARY、VARBINARY 等等）都可用于存储图像、字处理文档及声音文件等大对象。

有关 BLOB 存储的详细信息，请参见“存储 BLOB”一节第 5 页。

在 NULL 与 NOT NULL 之间进行选择

如果某行要求某列必须有值，应将该列定义为 NOT NULL。否则，该列可以包含 NULL 值，表示没有值。SQL Anywhere 中的缺省设置是允许 NULL 值，但如果没有充分的理由来允许 NULL 值，则应将相应的列显式声明为 NOT NULL。

SQL Anywhere 示例数据库中有一个部门表，在该表中包含名为 DepartmentID、DepartmentName 和 DepartmentHeadID 的列。其定义如下所示：

列	数据类型	大小	NULL/NOT NULL	约束
DepartmentID	整数	—	NOT NULL	无
DepartmentName	char	40	NOT NULL	无
DepartmentHeadID	整数	—	NULL	无

如果指定 NOT NULL，则必须为该表中的每一行提供列值。

有关 NULL 值的详细信息，请参见“NULL 值”一节 《SQL Anywhere 服务器 - SQL 参考》。有关在比较中使用 NULL 值的信息，请参见“搜索条件”一节 《SQL Anywhere 服务器 - SQL 参考》。

存储 BLOB

BLOB 是未经解释的字节串或字符串，作为值存储在列中。常见的 BLOB 示例有图片文件或声音文件。由于 BLOB 通常都很大，所以可将它们以任何字符串或二进制字符串数据类型（如 CHAR、VARCHAR、NCHAR、BINARY、VARBINARY 等等）的形式进行存储。根据需要存储的 BLOB 的内容和长度来选择数据类型和长度。

注意
通常将字符大对象称作 CLOB，将二进制大对象称作 BLOB，将二者的组合称作 LOB。在本文档中仅使用首字母缩写词 BLOB。

在创建一列以存储 BLOB 值时，您可以控制存储的各个方面。例如，可以指定将不超过指定大小的 BLOB 存储在行中（内置），而将超过指定大小的 BLOB 存储在表扩展页中行以外的位置。此外，您还可以指定，对于存储在行以外位置的 BLOB，该 BLOB 的前 *n* 个字节（也称作前缀）也会存储在行中。这些存储方面由在 CREATE TABLE 和 ALTER TABLE 语句中指定的 INLINE 和 PREFIX 设置来控制。您为这些设置指定的值会对性能或磁盘存储需求产生意想不到的影响。

如果 INLINE 和 PREFIX 都未指定，或者指定了 INLINE USE DEFAULT 或 PREFIX USE DEFAULT，则如下所述应用缺省值：

- 对于字符数据类型列（如 CHAR、NCHAR、LONG VARCHAR 和 XML），INLINE 的缺省值为 256，PREFIX 的缺省值为 8。
- 对于二进制数据类型列（如 BINARY、LONG BINARY、VARBINARY、BIT、VARBIT、LONG VARBIT、BIT VARYING 和 UUID），INLINE 的缺省值为 256，PREFIX 的缺省值为 0。

建议您不要设置 INLINE 和 PREFIX 值，除非缺省值无法满足某些特定要求。选择缺省值可以在性能和磁盘空间需求之间获得均衡。例如，如果将 INLINE 设置为较大值，并且所有 BLOB 都被内置存储，则行处理性能可能会下降。如果将 PREFIX 值设置得过大，则会增加存储 BLOB 所需的磁盘空间量，因为前缀数据会复制 BLOB 的一部分内容。

如果您确实决定设置 INLINE 或 PREFIX 值，则 INLINE 的长度不得超过列的长度。同样，PREFIX 的长度不得超过 INLINE 的长度。

压缩列的前缀数据未经压缩存储，因此如果满足请求所需的所有数据都存储在前缀中，则不必进行解压缩。

有关 INLINE 和 PREFIX 子句的缺省值的信息，请参见“CREATE TABLE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

BLOB 共享

如果 BLOB 超过了内置大小，并且需要多个数据库页面才能存储它，则数据库服务器会存储该 BLOB，这样，同一表中的其它行能够在需要时引用该 BLOB。这称作 BLOB 共享。BLOB 共享在内部处理，用于减少数据库中不必要的 BLOB 复制。

仅当您将一列的值设置为等于另一列的值时，才会发生 BLOB 共享。例如，`UPDATE t column1=column2;`。在本示例中，如果 column2 包含 BLOB，则在 column1 中不会复制这些 BLOB，而是使用指向 column2 中这些值的指针。

当某个 BLOB 被共享时，数据库服务器会对到该 BLOB 的其它引用数目加以跟踪。一旦数据库服务器确定某个 BLOB 在表内不再被引用，该 BLOB 即被删除。

如果某个 BLOB 在两个未压缩的列之间共享，然后其中的一个列被压缩，则该 BLOB 将不再被共享。

选择是否压缩列

可以压缩 CHAR、VARCHAR 和 BINARY 列以节省磁盘空间。例如，可以压缩其中存储了较大 BLOB 文件（如 BMP 和 TIFF）的列。使用 deflate 压缩算法可以实现压缩。此算法与 COMPRESS 函数使用的算法相同，它还是在 Windows ZIP 文件中使用的算法。

压缩列可以驻留在加密的表中。在这种情况下，数据先压缩，然后加密。

对于所含值少于 130 个字节的列或者所含值已经是压缩格式（如 JPG 文件）的列而言，不要使用列压缩。尝试对包含已压缩值的列进行压缩实际上会增加该列所需的存储空间。

要压缩列，可使用 CREATE TABLE 和 ALTER TABLE 语句的 COMPRESS 子句。

通过 sa_column_stats 系统过程，可以确定压缩列所获得的益处。

另请参见

- “CREATE TABLE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “ALTER TABLE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “sa_column_stats 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》
- “表加密”一节 《SQL Anywhere 服务器 - 数据库管理》

选择约束

虽然列的数据类型可以限制该列中允许包含的值（例如，只允许包含数字或只允许包含日期），但可能需要进一步限制允许的值。

可以通过指定 CHECK 约束来限制任何列的值。可以使用可能出现在 WHERE 子句中的任何有效条件来限制所允许的值。大多数 CHECK 约束使用 BETWEEN 或 IN 条件。

有关有效条件的详细信息，请参见“搜索条件”一节 《SQL Anywhere 服务器 - SQL 参考》。有关为表和列指定约束的详细信息，请参见“确保数据完整性”第 73 页。

教程：创建 SQL Anywhere 数据库

本教程介绍如何使用 Sybase Central 模仿 SQL Anywhere 示例数据库的 Products、SalesOrderItems、SalesOrders 和 Customers 表创建一个简单的数据库。

有关 SQL Anywhere 示例数据库的信息，请参见“SQL Anywhere 示例数据库”《SQL Anywhere 11 - 简介》。

有关数据库设计注意事项的信息，请参见“设计注意事项”一节第 4 页。

第 1 课：创建数据库文件

在这一课，您将创建一个用于存放数据库的数据库文件。该数据库文件包含对所有数据库都通用的一些系统表和其它系统对象；您将在稍后添加表。

♦ 新建数据库文件

1. 启动 Sybase Central。
2. 选择 [**工具**] » [**SQL Anywhere 11**] » [**创建数据库**]。
3. 阅读 [**欢迎**] 页上的信息，然后单击 [**下一步**]。
4. 选择 [**在这台计算机上创建数据库**]，然后单击 [**下一步**]。
5. 在 [**将主数据库文件保存到以下文件**] 字段中键入 *c:\temp\mysample.db*。

   如果临时目录不是 *c:\temp*，则请指定合适的路径。
6. 单击 [**完成**]。
7. 单击 [**关闭**]。

另请参见

- “设计注意事项”一节第 4 页
- “第 2 课：连接到数据库”一节第 8 页
- “第 3 课：向数据库中添加表”一节第 10 页
- “第 4 课：在列上设置 NOT NULL 约束”一节第 11 页
- “第 5 课：创建外键”一节第 12 页

第 2 课：连接到数据库

在这一课，您将使用 Sybase Central 连接到已创建的数据库文件。但是，如果您刚创建完数据库，则表示您已经连接到该数据库，从而您可以直接跳到下一课学习创建表。请参见“第 3 课：向数据库中添加表”一节第 10 页。

♦ 连接到数据库

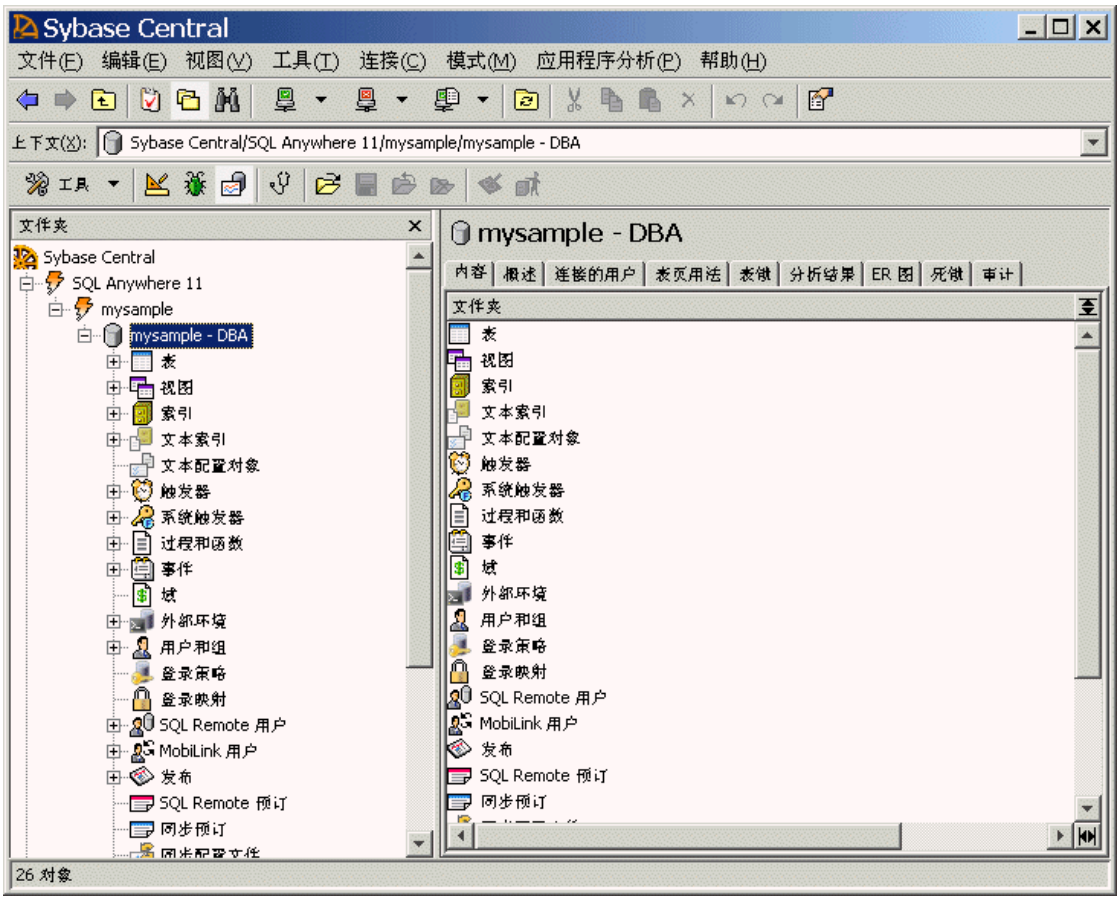
1. 启动 Sybase Central。

2. 选择 [**连接**] » [**使用 SQL Anywhere 11 连接**]。
3. 在 [**用户 ID**] 字段键入 **DBA**。这是新数据库的缺省用户 ID。
4. 在 [**口令**] 字段中键入 **sql**。这是新数据库的缺省口令。
5. 在缺省连接区域内选择 [**无**]。
6. 单击 [**数据库**] 选项卡，在 [**数据库文件**] 字段中键入数据库文件的完整路径。例如，键入：

   ```
   c:\temp\mysample.db
   ```

7. 单击 [**确定**]。

   数据库会启动，且有关数据库和运行该数据库的数据库服务器的信息会显示在 Sybase Central 中。



另请参见

- “设计注意事项”一节第 4 页
- “第 1 课：创建数据库文件”一节第 8 页
- “第 3 课：向数据库中添加表”一节第 10 页
- “第 4 课：在列上设置 NOT NULL 约束”一节第 11 页
- “第 5 课：创建外键”一节第 12 页

第 3 课：向数据库中添加表

在这一课，您将创建一个名为 Products 的表。

有关表设计注意事项的信息，请参见“设计注意事项”一节第 4 页。

♦ 创建表

1. 在 Sybase Central 的右窗格中，双击 [**表**]。
2. 右击 [**表**]，并选择 [**新建**] » [**表**]。
3. 在 [**您要给新表指定什么名称**] 字段中键入 **Products**。
4. 单击 [**完成**]。

   数据库服务器将使用缺省值创建该表并在右窗格中显示 [**列**] 选项卡。新列的 [**名称**] 字段被选定，并会出现提示，等待您为新列指定名称。
5. 键入 **ProductID** 作为新列的名称。

   因为这是表中的第一列，所以要选择 [**主键**]，表示该列是表的主键。

   创建表时，可以创建由多列组成的主键，方法是：创建多个列，并将复选标记放置在 [**主键**] 列中。请参见“主键”一节 《SQL Anywhere 11 - 简介》。
6. 在 [**数据类型**] 列表中，选择 [**Integer**]。
7. 单击省略号（三个句点）按钮。
8. 单击 [**值**] 选项卡，然后选择 [**缺省值**] » [**系统定义**] » [**自动增量**]。

   表中每增加一行，自动增量值会随之增加。这可以确保列中的值是唯一的—这是主键的一个必要条件。请参见“主键”一节 《SQL Anywhere 11 - 简介》。
9. 单击 [**确定**]。
10. 从 [**文件**] 菜单选择 [**新建**] » [**列**]。
11. 完成以下字段：
    - 在 [**名称**] 字段中键入 **ProductName**。
    - 在 [**数据类型**] 列表中，选择 [**Char**]。
    - 在 [**大小**] 列表中，选择 [**15**]。
12. 向数据库中添加以下表：
    - **Customers table**　添加一个名为 **Customers** 的表，它包含以下列：

- **CustomersID** 每个客户的标识号。确保 [**主键**] 已选中，并将 [**数据类型**] 设置为 [**Integer**]，将 [**缺省值**] 设置为 [**自动增量**]。
- **CompanyName** 每个公司的名称。将 [**数据类型**] 设置为最大长度为 **35** 个字符的 [**Char**]。

- **SalesOrders table** 添加一个名为 **SalesOrders** 的表，它包含以下列：
  - **SalesOrdersID** 每个销售订单的标识号。将 [**数据类型**] 设置为 [**Integer**]，并确保 [**主键**] 已选中。将 [**缺省值**] 设置为 [**自动增量**]。
  - **OrderDate** 下订单的日期。将 [**数据类型**] 设置为 [**Date**]。
  - **CustomerID** 下销售订单的客户的标识号。将 [**数据类型**] 设置为 [**Integer**]。
- **SalesOrderItems table** 添加一个名为 **SalesOrderItems** 的表，它包含以下列：
  - **SalesOrderItemsID** 项目所属的销售订单的标识号。将 [**数据类型**] 设置为 [**Integer**]，并确保 [**主键**] 已选中。
  - **LineID** 每个销售订单的标识号。将 [**数据类型**] 设置为 [**Integer**]，并确保 [**主键**] 已选中。

    注意
    由于同时为 SalesOrderItemsID 和 LineID 设置了 [**主键**]，这表示表的主键由这两列的连接值组成。
  - **ProductID** 所订购产品的标识号。将 [**数据类型**] 设置为 [**Integer**]。

13. 从 [**文件**] 菜单中选择 [**保存**]。

另请参见

- “设计注意事项”一节第 4 页
- “第 1 课：创建数据库文件”一节第 8 页
- “第 2 课：连接到数据库”一节第 8 页
- “第 4 课：在列上设置 NOT NULL 约束”一节第 11 页
- “第 5 课：创建外键”一节第 12 页

第 4 课：在列上设置 NOT NULL 约束

在这一课，您将学习如何在列上添加一个 NOT NULL 约束。

♦ 在列上添加和删除约束

1. 在 Sybase Central 的左窗格中，双击 [**表**]。
2. 单击 [**Products**]，然后单击右窗格中的 [**列**] 选项卡。
3. 选择 [**ProductName**] 列。
4. 从 [**文件**] 菜单中选择 [**属性**]。
5. 单击 [**约束**] 选项卡，然后选择 [**值不能为空**]。

缺省情况下，列允许 NULL，但如果没有充分的理由来允许 NULL，则将列声明为 NOT NULL 是一种很好的做法。请参见“NULL 值”一节 《SQL Anywhere 服务器 - SQL 参考》。

6. 单击 [**确定**]。

   此约束意味着对于添加到 [**Products**] 表的每一行，[**ProductName**] 列都必须有一个值。

7. 从 [**文件**] 菜单中选择 [**保存**]。

另请参见

- “设计注意事项”一节第 4 页
- “第 1 课：创建数据库文件”一节第 8 页
- “第 2 课：连接到数据库”一节第 8 页
- “第 3 课：向数据库中添加表”一节第 10 页
- “第 5 课：创建外键”一节第 12 页

第 5 课：创建外键

在这一课，您将学习如何使用外键来创建表间关系。使用上一课中所创建的表。

♦ **创建外键：**

1. 在 Sybase Central 的左窗格中，双击 [**表**]。
2. 在左窗格中，单击 [**SalesOrders**] 表以将其选定。
3. 在右窗格中，选择 [**约束**] 选项卡。
4. 选择 [**文件**] » [**新建**] » [**外键**]。
5. 在 [**希望此外键引用哪个表**] 列表中，选择 [**Products**] 表。
6. 在 [**您要给新外键指定什么名称**] 字段中键入 **ProductIDkey**。
7. 单击 [**下一步**]，并为 [**您希望此外键引用主键还是引用唯一约束**] 选择 [**主键**]。
8. 在 [**外列**] 列表中，单击 [**SalesOrdersItemsID**]。
9. 单击 [**完成**]。

至此，您就学完了有关创建关系数据库的入门章节。

另请参见

- “设计注意事项”一节第 4 页
- “第 1 课：创建数据库文件”一节第 8 页
- “第 2 课：连接到数据库”一节第 8 页
- “第 3 课：向数据库中添加表”一节第 10 页
- “第 4 课：在列上设置 NOT NULL 约束”一节第 11 页

使用数据库对象

目录

本节介绍添加数据库对象和设置数据库属性的过程。

用于创建、更改和删除数据库对象的 SQL 语句称为**数据定义语言** (DDL)。数据库对象的定义构成了数据库模式。模式是数据库的逻辑框架。

有关过程和触发器的详细信息，请参见“使用过程、触发器和批处理”第 777 页。

有关数据库创建和设计的概念上的信息，请参见：

- “在 SQL Anywhere 中创建数据库”第 3 页
- “确保数据完整性”第 73 页
- “使用 Sybase Central”一节 《SQL Anywhere 服务器 - 数据库管理》
- “使用 Interactive SQL”一节 《SQL Anywhere 服务器 - 数据库管理》

设置数据库对象的属性

可以查看或设置数据库和大多数数据库对象的属性。某些在创建数据库时选择的属性是不可配置的。

使用 Sybase Central 中的属性窗口查看和设置属性。如果未使用 Sybase Central，可以在以 CREATE 语句创建对象时指定属性。如果对象已经存在，可使用 ALTER 语句修改属性。

♦ 查看和编辑数据库对象的属性 (Sybase Central)

1. 在 Sybase Central 中，打开对象所在的文件夹。
2. 选择该对象。该对象的属性会出现在 Sybase Central 的右窗格中。
3. 在右窗格中，单击相应的选项卡来编辑属性。

   也可以在对象的属性窗口上查看和编辑属性。要查看属性窗口，右击对象，然后选择 [**属性**]。

另请参见

- “连接属性、数据库属性和数据库服务器属性” 《SQL Anywhere 服务器 - 数据库管理》

查看数据库中的系统对象

系统对象（例如系统表、系统视图、存储过程和域）保存有关数据库对象的信息以及它们彼此之间如何相关的信息。系统视图、过程和域在很大程度上支持 Sybase Transact-SQL 兼容性。

♦ 显示数据库中的系统对象 (Sybase Central)

1. 启动数据库服务器。
2. 以具有 DBA 权限的用户身份连接到数据库。
3. 选择 [**文件**] » [**配置所有者过滤**]。
4. 选择 **SYS** 和 **dbo**，然后单击 [**确定**]。

♦ 浏览系统表 (SQL)

1. 连接到数据库。
2. 执行 SELECT 语句，通过查询 SYSOBJECT 系统视图来得到对象列表。

示例

以下 SELECT 语句查询 SYSOBJECT 系统视图，并返回由 SYS 和 dbo 所拥有的所有表和视图的列表。会与 SYSTAB 系统视图建立连接，以返回对象名，与 SYSUSER 系统视图建立连接以返回所有者名称。

```
SELECT b.table_name "Object Name",
       c.user_name "Owner",
       b.object_id "ID",
       a.object_type "Type",
       a.status "Status"
  FROM ( SYSOBJECT a JOIN SYSTAB b
         ON a.object_id = b.object_id )
  JOIN SYSUSER c
WHERE c.user_name = 'SYS'
   OR c.user_name = 'dbo'
ORDER BY c.user_name, b.table_name;
```

另请参见

- “SYSOBJECT 系统视图”一节 《SQL Anywhere 服务器 - SQL 参考》
- “SYSTAB 系统视图”一节 《SQL Anywhere 服务器 - SQL 参考》
- “SYSUSER 系统视图”一节 《SQL Anywhere 服务器 - SQL 参考》

使用表

首次创建数据库时，数据库中只有系统表。系统表保存数据库模式。

本节将介绍如何创建、变更和删除表。可以在 Interactive SQL 中执行示例，但 SQL 语句与所使用的管理工具无关。请参见“在 Interactive SQL 中编辑结果集”一节 《SQL Anywhere 服务器 - 数据库管理》。

为了使在必要时重建数据库模式的过程更加简单，可创建命令文件来定义数据库中的表。命令文件应包含 CREATE TABLE 和 ALTER TABLE 语句。

有关组、表和以另一个用户的身份进行连接的详细信息，请参见“引用组拥有的表”一节 《SQL Anywhere 服务器 - 数据库管理》和“数据库对象名和前缀”一节 《SQL Anywhere 服务器 - 数据库管理》。

创建表

首次创建数据库时，数据库中只有系统表，用以保存数据库模式。可以在 Interactive SQL 或 Sybase Central 中使用 SQL 语句来创建用于保存数据的新表。

您可以创建两种类型的表：

- **基表** 保存持久性数据的表。在您显式删除数据或删除该表之前，基表及其数据都一直存在。称之为基表是为了与临时表和视图相区分。
- **临时表** 临时表中的数据只针对一个连接而保存。全局临时表定义（但不包括数据）一直保存在数据库中直到被删除。局部临时表定义和数据的存在时间仅为一个连接的持续时间。请参见“使用临时表”一节第 31 页。

表由行和列组成。每一列保存一类特定的信息（例如电话号码或名称），而每一行指定一个特定的条目。

♦ 创建表 (Sybase Central)

1. 以具有 DBA 权限的用户身份连接到数据库。
2. 在左窗格中，右击 [**表**] 并选择 [**新建**] » [**表**]。
3. 按照 [**创建表向导**] 中的说明操作。
4. 在右窗格中,单击 [**列**] 选项卡，并配置表。
5. 选择 [**文件**] » [**保存**]。

♦ 创建表 (SQL)

1. 以具有 DBA 权限的用户身份连接到数据库。
2. 执行 CREATE TABLE 语句。

示例

以下语句会创建一个新表来描述公司中雇员的资格。表中的列分别保存每种技能的标识号、技能名称和类型（技术或管理）。

```
CREATE TABLE Skills (
   SkillID INTEGER NOT NULL,
   SkillName CHAR( 20 ) NOT NULL,
   SkillType CHAR( 20 ) NOT NULL
);
```

请参见“CREATE TABLE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

变更表

本节将介绍如何变更表的结构或列定义。例如，可以添加列、更改各种列属性或彻底删除列。

在 Sybase Central 中，表变更任务可在右窗格的 [**SQL**] 选项卡上执行，而在 Interactive SQL 中，则可以使用 ALTER TABLE 语句来执行。

有关变更数据库对象属性的信息，请参见“设置数据库对象的属性”一节第 14 页。

有关授予和撤消表权限的信息，请参见“授予针对表的权限”一节 《SQL Anywhere 服务器 - 数据库管理》和“撤消用户权限和特权”一节 《SQL Anywhere 服务器 - 数据库管理》。

表变更与视图依赖性

变更表前，最好使用 sa_dependent_views 系统过程来确定是否存在依赖于该表的视图。请参见“sa_dependent_views 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》。

如果要为带有相关视图的表变更模式，可能需要采取一些如以下各节中所述的附加步骤。

- **相关常规视图** 变更表模式时，数据库中表的定义会被更新。如果有相关常规视图，数据库服务器会在表变更操作执行后自动重新编译这些视图。如果对表模式进行更改后，数据库服务器不能重新编译相关常规视图，这可能是因为您所做的更改使视图定义失效。在这种情况下，必须改正视图定义。请参见“变更常规视图”一节第 40 页。
- **相关实例化视图** 如果有相关实例化视图，必须在变更表前禁用这些视图，然后在变更表后重新启用它们。如果对表模式进行更改后，不能重新启用相关的实例化视图，这可能是因为您所做的更改使实例化视图定义失效。此种情况下，必须删除实例化视图，然后用有效的定义重新创建，或在尝试重新启用实例化视图前对基表进行适当变更。请参见“创建实例化视图”一节第 53 页。

有关变更数据库对象将如何影响视图依赖性的概括说明，请参见“视图依赖性”一节第 34 页。

变更表 (Sybase Central)

可以在 Sybase Central 右窗格的 [**列**] 选项卡上变更表。例如，可以添加或删除列、更改列定义或者更改表或列属性。如果存在任何相关的实例化视图，表的变更即会失败；必须先禁用相关的实例化视图。表变更完毕后，必须重新启用相关实例化视图。请参见“视图依赖性”一节第 34 页。

使用 sa_dependent_views 系统过程来确定是否存在相关实例化视图。请参见“sa_dependent_views 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》。

♦ 变更现有的表 (Sybase Central)

1. 以具有 DBA 权限的用户身份或以表所有者身份连接到数据库。
2. 如果要更改模式，但存在依赖于表的实例化视图，则按如下步骤禁用各视图：
   a. 在左窗格中，双击 [**视图**]。
   b. 右击实例化视图，并选择 [**禁用**]。
3. 双击 [**表**]，并选择要变更的表。
4. 在右窗格中,单击 [**列**] 选项卡，并变更表设置。
5. 选择 [**文件**] » [**保存**]。
6. 如果禁用了实例化视图，则重新启用和初始化每一个视图。请参见“启用和禁用实例化视图”一节第 60 页。

提示

可以选择表的 [**列**] 选项卡并选择 [**文件**] » [**新建列**] 来添加列。

可以选择 [**列**] 选项卡上的列并选择 [**编辑**] » [**删除**] 来删除列。

可以将列复制到某个表中。方法是：在右窗格的 [**列**] 选项卡上选择列，然后单击 [**复制**]。选择表，在右窗格中单击 [**列**] 选项卡，然后单击 [**粘贴**]。

还必须单击 [**保存**] 或选择 [**文件**] » [**保存**]。只有在执行此操作后，所做的更改才会在表中生效。

另请参见

- “启用和禁用实例化视图”一节第 60 页
- “确保数据完整性”第 73 页
- “视图依赖性”一节第 34 页

变更表 (SQL)

可以在 Interactive SQL 中使用 ALTER TABLE 语句来变更表。如果对具有相关实例化视图的表执行 ALTER TABLE 语句时使用了除 ADD FOREIGN KEY 之外的子句，则 ALTER TABLE 语句会失败。对于所有其它子句，必须禁用相关实例化视图，然后在更改完成后重新启用这些视图。请参见“视图依赖性”一节第 34 页。

使用 sa_dependent_views 系统过程来确定是否存在相关实例化视图。请参见“sa_dependent_views 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》。

♦ 变更现有的表 (SQL)

1. 以具有 DBA 权限的用户身份连接到数据库。

2. 如果要对具有相关实例化视图的表执行模式变更操作，并对 ALTER TABLE 语句使用除 ADD FOREIGN KEY 之外的子句，则使用 ALTER MATERIALIZED VIEW ...DISABLE 语句禁用每个相关实例化视图。不需要禁用相关常规视图。
3. 执行 ALTER TABLE 语句来变更表。

   数据库中表的定义会被更新。
4. 如果已禁用了任何实例化视图，则使用 ALTER MATERIALIZED VIEW ...ENABLE 语句来重新启用这些视图。

示例

这些示例说明如何更改数据库的结构。ALTER TABLE 语句几乎可以更改有关表的所有内容，可以使用它来添加或删除外键、将列从一种类型更改为另一种类型，等等等等。在所有这些情况下，一旦进行了更改，引用该表的存储过程、视图和任何其它项都可能无法工作。

以下命令将一列添加到 Skills 表中，用以输入技能的可选说明：

```
ALTER TABLE Skills
ADD SkillDescription CHAR( 254 );
```

还可以使用 ALTER TABLE 语句来变更列属性。以下语句将 SkillDescription 列从最多 254 个字符缩短为最多 80 个字符：

```
ALTER TABLE Skills
ALTER SkillDescription CHAR( 80 );
```

缺省情况下，如果存在长于 80 个字符的条目，将发生错误。可使用 string_rtruncation 选项来更改这一行为。请参见“string_rtruncation 选项 [兼容性]”一节 《SQL Anywhere 服务器 - 数据库管理》。

以下语句将 SkillType 列的名称更改为 Classification：

```
ALTER TABLE Skills
RENAME SkillType TO Classification;
```

以下语句删除 Classification 列。

```
ALTER TABLE Skills
DROP Classification;
```

以下语句更改整个表的名称：

```
ALTER TABLE Skills
RENAME Qualification;
```

另请参见

- “ALTER TABLE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “ALTER VIEW 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “变更常规视图”一节第 40 页
- “启用和禁用实例化视图”一节第 60 页
- “ALTER MATERIALIZED VIEW 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “确保数据完整性”第 73 页
- “视图依赖性”一节第 34 页

删除表

本节将介绍如何从数据库中删除表。可以使用 Sybase Central 或 Interactive SQL 执行此任务。在 Interactive SQL 中，删除 (Delete) 表也称作将其删除 (Drop)。

不能删除在 SQL Remote 发布中用作项目的表。如果在 Sybase Central 中试图执行此操作，则将出现错误。另外，如果要删除带有相关视图的表，可能需要采取一些如以下各节中所述的附加步骤。

表删除与视图依赖性

删除表时，表的定义会从数据库中删除。如果存在相关常规视图，数据库服务器会在执行表变更操作后尝试重新编译并重新启用这些视图。如果不能这样做，很可能是因为表删除操作使视图定义失效。在这种情况下，必须改正视图定义。请参见“变更常规视图”一节第 40 页。

如果存在相关实例化视图，后续的刷新操作将会失败，因为视图的定义不再有效。在这种情况下，必须删除实例化视图，然后以有效的定义重新创建。请参见“创建实例化视图”一节第 53 页。

变更表前，最好使用 sa_dependent_views 系统过程来确定是否存在依赖于该表的视图。请参见“sa_dependent_views 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》。

有关表删除操作将如何影响视图依赖性的概括说明，请参见“视图依赖性”一节第 34 页。

♦ 删除表 (Sybase Central)

1. 以具有 DBA 权限的用户身份或以表所有者身份连接到数据库。
2. 如果要删除的表存在相关实例化视图，则禁用每个实例化视图：
   a. 在左窗格中，双击 [**视图**]。
   b. 右击实例化视图，并选择 [**禁用**]。
3. 双击 [**表**]。
4. 右击表，然后选择 [**删除**]。
5. 单击 [**是**]。

♦ 删除表 (SQL)

1. 以具有 DBA 权限的用户身份或以表所有者身份连接到数据库。
2. 如果要删除的表存在相关实例化视图，则使用 ALTER MATERIALIZED VIEW ...DISABLE 语句来禁用每个相关实例化视图。
3. 执行 DROP TABLE 语句。

示例

以下 DROP TABLE 命令会删除 Skills 表中的所有记录，然后从数据库中删除 Skills 表的定义

```
DROP TABLE Skills;
```


与 CREATE 语句类似，DROP 语句在删除表前后会自动执行 COMMIT 语句。这会使上次执行 COMMIT 或 ROLLBACK 后对数据库的所有更改成为永久更改。DROP 语句还会删除表上的所有索引。请参见“DROP TABLE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

浏览表中的数据

可以使用 Sybase Central 或 Interactive SQL 来浏览保存在数据库的表中的数据。

如果您在使用 Sybase Central，则可以通过在右窗格中选择该表并单击 [**数据**] 选项卡来查看表中的数据。

如果您在使用 Interactive SQL，则执行以下语句：

```
SELECT * FROM table-name;
```

可以通过 Interactive SQL [**结果**] 选项卡或 Sybase Central 中表的 [**数据**] 选项卡来编辑表中的数据。

另请参见

- “使用 Interactive SQL”一节 《SQL Anywhere 服务器 - 数据库管理》

管理主键

主键由一列或组合在一起的一组列组成，其值标识表中的唯一行。主键值不会随着行中数据的有效期发生变化。因为唯一性对良好的数据库设计至关重要，所以在定义表时最好指定主键。

建议不要将近似数据类型（例如 FLOAT 和 DOUBLE）用于主键或具有唯一约束的列。近似数值数据类型在算术运算后容易产生舍入误差。

本节将介绍如何在数据库中创建和编辑主键。可以使用 Sybase Central 或 Interactive SQL 来执行这些任务。

多列主键中的列顺序
主键列的顺序由 CREATE TABLE 语句中的主键和外键子句决定。它与 CREATE TABLE 语句主键声明中列的指定顺序无关。

管理主键 (Sybase Central)

主键是用于标识表中唯一行的一个列或一组列。主键通常在创建表时创建，但以后可进行修改。在 Sybase Central 中，可通过以下两种方法之一访问表的主键：

- 右击表并选择 [**设置主键**]，这将启动 [**设置主键向导**]。[**设置主键向导**] 也可以用来更改现有主键的列顺序。
- 在左窗格中选择表，然后在右窗格中选择 [**约束**] 选项卡。

♦ 配置主键 (Sybase Central)

1. 以具有 DBA 权限的用户身份或以表所有者身份连接到数据库。
2. 在左窗格中，双击 [**表**]。
3. 右击表，然后选择 [**设置主键**]。
4. 按照 [**设置主键向导**] 中的说明操作。

管理主键 (SQL)

可以在 Interactive SQL 中使用 CREATE TABLE 和 ALTER TABLE 语句来创建和变更主键。这些语句可以设置多个表属性，其中包括列约束和检查。

主键中的列不能包含 NULL 值。必须在主键中的各列上指定 NOT NULL。

♦ 添加主键 (SQL)

1. 以具有 DBA 权限的用户身份连接到数据库。
2. 对要配置主键的表执行 ALTER TABLE 语句。

♦ 修改主键 (SQL)

1. 以具有 DBA 权限的用户身份连接到数据库。
2. 执行 ALTER TABLE 语句来删除现有主键。
3. 执行 ALTER TABLE 语句来添加主键。

♦ 删除主键 (SQL)

1. 以具有 DBA 权限的用户身份连接到数据库。
2. 执行带有 DELETE PRIMARY KEY 子句的 ALTER TABLE 语句。

示例 1

以下语句创建名为 Skills 的表，然后将 SkillID 列指定为主键：

```
CREATE TABLE Skills (
   SkillID INTEGER NOT NULL,
   SkillName CHAR( 20 ) NOT NULL,
   SkillType CHAR( 20 ) NOT NULL,
   PRIMARY KEY( SkillID )
);
```

该表中每一行的主键值必须是唯一的，在本示例中，这就意味着您不能给多个行指定同一个 SkillID。表中的每一行由其主键值唯一地标识。

如果要将主键更改为 SkillID 和 Skillname 列的组合，必须首先删除已创建的主键，然后添加新的主键：

```
ALTER TABLE Skills DELETE PRIMARY KEY
ALTER TABLE Skills ADD PRIMARY KEY ( SkillID, SkillName );
```

请参见“ALTER TABLE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》和“管理主键 (Sybase Central)”一节第 22 页。

管理外键

本节将介绍如何在数据库中创建和编辑外键。可以使用 Sybase Central 或 Interactive SQL 来执行这些任务。

外键的作用是将子表（即外表）中的值与父表（即主表）中的值相关联。一个表可以有引用多个父表的多个外键，从而链接多种类型的信息。

示例

SQL Anywhere 示例数据库有一个保存雇员信息的表和一个保存部门信息的表。Departments 表包含以下各列：

- **DepartmentID** 部门的 ID 号。这是该表的主键。
- **DepartmentName** 部门的名称。
- **DepartmentHeadID** 部门经理的雇员 ID。

要查找特定雇员所在部门的名称，不需要将该雇员的部门名称放入 Employees 表中。而是在 Employees 表中包含一个 DepartmentID 列，用以保存与 Departments 表中某一 DepartmentID 值相匹配的值。

Employees 表中的 DepartmentID 列是指向 Departments 表的外键。外键引用包含相应主键的表中的特定行。

因此，Employees 表（其中包含关系中的外键）称为**外表**或**引用表**。Departments 表（其中包含被引用的主键）称为**主表**或**被引用表**。

管理外键 (Sybase Central)

在 Sybase Central 中，选择某表后，其外键会出现在 [**约束**] 选项卡上，该选项卡位于右窗格中。

外键关系在创建子表时（即在子表中插入数据前）创建。然后，外键关系相当于一个约束；针对子表中插入的新行，数据库服务器会检查插入外键列中的值与主表的主键中的值是否匹配。

创建外键后，可以在右窗格中每个表的 [**约束**] 选项卡上跟踪外键；此选项卡会显示引用当前所选表的所有外表。

♦ 创建新外键 (Sybase Central)

1. 以具有 DBA 权限的用户身份或以表所有者身份连接到数据库。
2. 在左窗格中，双击 [**表**]。
3. 右击表，然后选择 [**新建**] » [**外键**]。
4. 按照 [**创建外键向导**] 中的说明操作。

♦ 删除外键 (Sybase Central)：

1. 以具有 DBA 权限的用户身份或以表所有者身份连接到数据库。

2. 在左窗格中，双击 [**表**]。
3. 选择要为其删除外键的表。
4. 在右窗格中，单击 [**约束**] 选项卡。
5. 右击外键，然后选择 [**删除**]。
6. 单击 [**是**]。

对于任何给定表，也可以查看通过外键引用此表的各表的列表。

♦ 显示引用给定表的表的列表 (Sybase Central)

1. 以具有 DBA 权限的用户身份或以表所有者身份连接到数据库。
2. 在左窗格中，双击 [**表**]。
3. 单击表。
4. 在右窗格中，单击 [**引用约束**] 选项卡。

提示
在使用向导创建外键时，可以设置该外键的属性。要在创建外键后查看属性，请在 [**约束**] 选项卡上选择外键，然后选择 [**文件**] » [**属性**]。

可以在 [**引用约束**] 选项卡上选择表，然后选择 [**文件**] » [**属性**]，来查看引用外键的属性。

管理外键 (SQL)

一个表只能定义一个主键，但可以具有多个外键。可以在 Interactive SQL 中使用 CREATE TABLE 和 ALTER TABLE 语句来创建和变更外键。这些语句可以设置多个表属性，其中包括列约束和检查。

♦ 创建外键 (SQL)

1. 以具有 DBA 权限的用户身份连接到数据库。
2. 执行 ALTER TABLE 语句。

创建外键时忽略列名 (SQL)

外键列名和主键列名将根据在两个列表中的位置进行一对一式的配对。如果在定义外键时未指定主表列名，则使用主键列。例如，假设按如下方式创建两个表：

```
CREATE TABLE Table1( a INT, b INT, c INT, PRIMARY KEY ( a, b ) );
CREATE TABLE Table2( x INT, y INT, z INT, PRIMARY KEY ( x, y ) );
```

然后，按如下方式创建外键 fk1，明确指定如何对两个表之间的列进行配对：

```
ALTER TABLE Table2 ADD FOREIGN KEY fk1( x,y ) REFERENCES Table1( a, b );
```

使用下面的语句创建第二个外键 fk2，仅指定外表列。数据库服务器会自动将这两个列与主表主键中的前两列配对。

```
ALTER TABLE Table2 ADD FOREIGN KEY fk2( x, y ) REFERENCES Table1;
```

使用下面的语句创建外键，不指定主表或外表的列：

```
ALTER TABLE Table2 ADD FOREIGN KEY fk3 REFERENCES Table1;
```

由于没有指定引用列，数据库服务器会在外表 (Table2) 中查找与主表 (Table1) 中的列同名的列。如果存在，数据库服务器会确保数据类型匹配，然后使用这些列创建外键。否则，将在 Table2 中创建这些列。在此示例中，Table2 没有名为 a 和 b 的列，因此将创建这两个列且数据类型与 Table1.a 和 Table1.b 相同。这些自动创建的列不会成为外表主键的一部分。

示例

在下例中，将创建一个名为 Skills 的表，其中包含潜在技能列表，然后创建一个与 Skills 表具有外键关系、名为 EmployeeSkills 的表。请注意，EmployeeSkills.SkillID 与 Skills 表的主键列 (Id) 存在外键关系。

```
CREATE TABLE Skills (
   Id INTEGER PRIMARY KEY,
   SkillName CHAR(40),
   Description CHAR(100)
);
CREATE TABLE EmployeeSkills (
   EmployeeID INTEGER NOT NULL,
   SkillId INTEGER NOT NULL,
   SkillLevel INTEGER NOT NULL,
   PRIMARY KEY( EmployeeID ),
   FOREIGN KEY (SkillID) REFERENCES Skills ( Id )
);
```

您也可以在创建完表后使用 ALTER TABLE 语句将外键添加到该表。在下例中，将创建与上例中类似的表，唯一不同之处是在创建表后再添加外键。

```
CREATE TABLE Skills2 (
   Id INTEGER PRIMARY KEY,
   SkillName CHAR(40),
   Description CHAR(100)
);
CREATE TABLE EmployeeSkills2 (
   EmployeeID INTEGER NOT NULL,
   SkillId INTEGER NOT NULL,
   SkillLevel INTEGER NOT NULL,
   PRIMARY KEY( EmployeeID ),
);
ALTER TABLE EmployeeSkills2
   ADD FOREIGN KEY SkillFK ( SkillID )
   REFERENCES Skills2 ( Id );
```

可以在创建外键时指定外键的属性。例如，以下语句将创建与示例 2 中相同的外键，但它将外键定义为 NOT NULL，同时定义更新或删除时的限制。

```
ALTER TABLE Skills2
ADD NOT NULL FOREIGN KEY SkillFK ( SkillID )
REFERENCES Skills2 ( ID )
ON UPDATE RESTRICT
ON DELETE RESTRICT;
```

另请参见

- “管理外键 (Sybase Central)”一节第 24 页
- “CREATE TABLE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “ALTER TABLE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》

使用计算列

计算列是一个以可引用同一行中其它列（称为**相关列**）值的表达式作为值的列。如果要为一个复杂的表达式创建索引而该表达式可能包含一个或多个相关列的值，计算列就特别有用。数据库服务器在发现某个表达式与计算列的 COMPUTE 表达式相匹配时，即会使用计算列；这包括 SELECT 列表及谓语。但是，如果查询表达式包含某个特殊值（如 CURRENT TIMESTAMP），这一匹配则不会发生。有关会阻止匹配发生的特殊值的列表，请参见“特殊值”一节 《SQL Anywhere 服务器 - SQL 参考》。

在查询优化过程中，SQL Anywhere 优化程序会自动尝试将涉及复杂表达式的谓语转换为只引用计算列定义的谓语。例如，假定您要查询包含有关产品发货汇总信息的表：

```
CREATE TABLE Shipments(
   ShipmentID INTEGER NOT NULL PRIMARY KEY,
   ShipmentDate TIMESTAMP,
   ProductCode CHAR(20) NOT NULL,
   Quantity INTEGER NOT NULL,
   TotalPrice DECIMAL(10,2) NOT NULL
);
```

具体地说，查询要返回平均成本介于两美元和四美元之间的发货。可按如下方法编写这一查询：

```
SELECT *
   FROM Shipments
   WHERE ( TotalPrice / Quantity ) BETWEEN 2.00 AND 4.00;
```

但是，在上述查询中，WHERE 子句中的谓语是不可优化搜索的，因为它不引用单个基列。请参见“在查询中使用谓语”一节第 531 页。如果 Shipments 表相对较大，可能更适合采用索引检索而不是顺序扫描。要从索引检索获益，为 Shipments 表创建名为 AverageCost 的计算列，然后在该列上创建索引，如下所示：

```
ALTER TABLE Shipments
   ADD AverageCost DECIMAL(21,13)
   COMPUTE( TotalPrice / Quantity );
 CREATE INDEX IDX_average_cost
   ON Shipments( AverageCost ASC );
```

选择计算列的类型是很重要的；只有在查询中的表达式数据类型与计算列的数据类型完全匹配时，SQL Anywhere 优化程序才会将复杂的表达式替换为计算列。要确定表达式的类型，可以使用内置函数 EXPRTYPE，它会以现成的 SQL 术语返回表达式的类型：

```
SELECT EXPRTYPE(
 'SELECT ( TotalPrice/Quantity ) AS X FROM Shipments', 1 )
   FROM DUMMY;
```

对于 Shipments 表，上述查询会返回 decimal(21,13)。在优化过程中，SQL Anywhere 优化程序按如下所示重写上述查询：

```
SELECT *
   FROM Shipments
   WHERE AverageCost
   BETWEEN 2.00 AND 4.00;
```

在此例中，WHERE 子句中的谓语现在是可优化搜索的，使优化程序能使用新的 IDX_average_cost 索引为该查询的访问计划选择索引扫描。

变更计算列表达式

可以使用 ALTER TABLE 语句更改计算列中所用的表达式。下面的语句会更改计算列所基于的表达式。

```
ALTER TABLE table-name
ALTER column-name
SET COMPUTE ( new-expression );
```

在执行上述语句时，将重新计算该列。如果新表达式无效，ALTER TABLE 语句将会失败。

下面的语句使某个列不再成为计算列。

```
ALTER TABLE
table-name
ALTER column-name
DROP COMPUTE;
```

该列中现有的值在此语句执行时不会发生变化，但它们不会再自动更新。

插入和更新计算列

有关计算列值插入与更新的注意事项包括以下几项：

- **直接插入和更新** 不应使用 INSERT 或 UPDATE 语句在计算列中放入值，因为这些值不能反映所需的计算。此外，在计算列中手工插入或更新的数据在以后重新计算列时可能会发生变化。
- **列依赖性** 强烈建议您不要使用触发器设置计算列的定义中引用的列值（例如，将 NULL 值更改为非 NULL 值），因为这会导致计算列的值不反映所需的计算。
- **列出列名** 您必须始终在 INSERT 语句中显式指定具有计算列的表的列名。
- **触发器** 如果对计算列定义触发器，任何影响该列的 INSERT 或 UPDATE 语句都将触发触发器。

虽然可以使用 INSERT、UPDATE 或 LOAD TABLE 语句在计算列中插入值，但既不建议使用该功能，也不需要使用该功能。

LOAD TABLE 语句允许对计算列进行 *可选* 计算。加载操作期间取消计算会使复杂的卸载/重装序列执行得更快。当计算列的值必须保持不变的情况下，这样做也很有用，即使 COMPUTE 表达式引用不确定值（如 CURRENT TIMESTAMP）。

避免更改触发器中相关列的值，因为这会导致计算列的值与列定义不一致。

如果计算列 x 依赖于已声明非 NULL 的列 y，则如果尝试将 y 设置为 NULL，将在触发器触发之前被拒绝，并生成错误消息。

重新计算计算列

随着行的插入和更新，数据库服务器会自动维护计算列的值。大多数应用程序根本不需要直接更新或插入计算列值。

计算列在出现以下情况时会重新计算：

- 删除、添加或重命名了列。
- 重命名了表。
- 修改了任何列的数据类型或 COMPUTE 子句。
- 插入了行。
- 更新了行。

计算列在出现以下情况时*不*会重新计算：

- 查询计算列。
- 计算列所依赖的列中的值发生更改。

在数据库中或数据库之间复制表或列

使用 Sybase Central，您可以复制现有的表或列并将它们插入同一个数据库中的另一个位置或完全不同的数据库中。请参见“在 SQL Anywhere 插件中复制数据库对象”一节 《SQL Anywhere 服务器 - 数据库管理》。

如果不使用 Sybase Central：

- 要将 SELECT 语句结果插入指定位置，请参见“SELECT 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。
- 要将一行或多行从数据库的其它位置插入表中，请参见“INSERT 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

使用临时表

临时表存储在临时文件中。就如任何其它 dbspace 的页一样，临时文件的页也可以缓存。对临时表的操作决不会写入事务日志。临时表分为两类：**局部临时**表和**全局临时**表。

另请参见

- “CREATE TABLE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “DECLARE LOCAL TEMPORARY TABLE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》

局部临时表

局部临时表仅存在于某个连接存续期间或者某个复合语句存续期间（如果该表是在复合语句中定义的）。

全局临时表会在数据库中保留，直至通过 DROP TABLE 语句显式删除。"全局" 一词用于指示来自相同或不同应用程序的多个连接可以同时使用此表。全局临时表具有如下特性：

- 表的定义记录在目录中，并将保留到表被显式删除。
- 对表进行的插入、更新和删除操作不会记录在事务日志中。
- 表的列统计信息由数据库服务器在内存中维护。

全局临时表

全局临时表分为两类：**非共享**和**共享**。通常情况下，全局临时表是非共享的，也就是说，每个连接只能在表中看到其自己的行。连接结束时，该连接的行即从表中删除。

如果全局临时表为共享的，表的所有数据均在所有连接间共享。要创建共享的全局临时表，创建表时应指定 SHARE BY ALL 子句。除了全局临时表的一般特性外，共享的全局临时表还具有以下特性：

- 表的内容会保留到表被显式删除或数据库被关闭。
- 数据库启动后，表是空的。
- 表的行锁定行为与基表相同。

非事务性临时表

使用 CREATE TABLE 语句的 NOT TRANSACTIONAL 子句可以将临时表声明为非事务性表。在某些情况下，NOT TRANSACTIONAL 子句可以提高性能，因为对非事务性临时表执行的操作不会记入回退日志中。例如，如果反复调用使用临时表的过程而不会干预到 COMMIT 或 ROLLBACK，或者表中包含许多行，则 NOT TRANSACTIONAL 会很有用。对非事务性临时表的更改不受 COMMIT 或 ROLLBACK 的影响。

创建临时表

可以使用 SQL 语句或 Sybase Central 创建临时表。

♦ **创建表 (Sybase Central)**

1. 以具有 DBA 权限的用户身份或以表所有者身份连接到数据库。
2. 右击 [**表**]，然后选择 [**新建**] » [**全局临时表**]。
3. 按照 [**创建全局临时表向导**] 中的说明操作。
4. 在右窗格中,单击 [**列**] 选项卡，并配置表。
5. 选择 [**文件**] » [**保存**]。

♦ **创建表 (SQL)**

1. 以具有 DBA 权限的用户身份连接到数据库。
2. 执行 CREATE TABLE 语句或 DECLARE LOCAL TEMPORARY TABLE 语句。

另请参见

- “CREATE TABLE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “DECLARE LOCAL TEMPORARY TABLE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》

在过程中引用临时表

如果临时表定义不一致，则在过程之间共享该表会导致出现问题。例如，假设有两个过程 procA 和 procB，这两个过程均定义临时表 temp_table，并调用称为 sharedProc 的另一过程。即未调用 procA，也未调用 procB，所以临时表还不存在。

现在假设 temp_table 在 procA 中的定义与在 procB 中的定义略有不同—两个过程均使用相同的列名称和类型，但列顺序不同。

调用 procA 时，它返回预期结果。但调用 procB 时，它返回不同结果。

这是因为，调用 procA 时，它创建 temp_table，然后调用 sharedProc。调用 sharedProc 时，会解析并验证其中的 SELECT 语句，然后缓存该语句解析后的表示形式，以便再次执行 SELECT 语句。缓存的形式反映 procA 中表定义的列顺序。

调用 procB 时将重新创建 temp_table，但列顺序不同。procB 调用 sharedProc 时，数据库服务器使用 SELECT 语句的缓存表示形式。因此，结果不同。

您可以通过执行以下操作之一避免此问题发生：

- 确保以此方式使用的临时表定义一致
- 考虑改用全局临时表

使用视图

视图是由视图定义的结果集定义的计算表，表示为 SQL 查询。您可以使用视图以一种可以控制的格式向数据库用户显示所要提供的信息。SQL Anywhere 支持两种类型的视图：**常规视图**和**实例化视图**。

数据库中每个视图的定义都存储在 ISYSVIEW 系统表中。请参见“SYSVIEW 系统视图”一节《SQL Anywhere 服务器 - SQL 参考》。

文档约定

在 SQL Anywhere 文档中，术语**常规视图**用于描述每次引用视图时都重新计算并且结果集不存储在磁盘上的视图。这是最常用的视图类型。大多数文档是指常规视图。

术语**实例化视图**用于描述其结果集预先计算并在磁盘上实例化的视图（与基表内容类似）。

单独的术语**视图**在文档中的含义视上下文而定。在讨论常规视图和实例化视图的通用方面的章节中使用时，它既指常规视图也指实例化视图。如果该术语用在实例化视图的文档中，是指实例化视图，用在常规视图的文档中，则是指常规视图。

比较常规视图、实例化视图和基表

下表比较常规视图、实例化视图和基表：

功能	常规视图	实例化视图	基表
允许访问权限	是	是	是
允许 SELECT	是	是	是
允许 UPDATE	有些	否	是
允许 INSERT	有些	否	是
允许 DELETE	有些	否	是
允许相关视图	是	是	是
允许索引	否	是	是
允许完整性约束	否	否	是
允许键	否	否	是

使用视图的优点

视图允许您定制对数据库中数据的访问。定制访问有以下几个好处：

- **有效地利用资源** 常规视图不需要额外的数据存储空间；每次调用这些视图时都会对其重新计算。实例化视图需要磁盘空间，但每次调用时不必重新计算。当数据库很大并且数据库服务器要处理频繁、重复的连接相同表的请求时，实例化视图可以缩短响应时间。
- **提高安全性** 因为可以只允许访问相关的信息。
- **提高可用性** 因为可以以一种比基表中的数据更易于理解的格式向用户和应用程序开发人员提供数据。
- **提高一致性** 因为可以在数据库中集中一般查询的定义。

视图依赖性

视图定义可以引用其它对象（包括列、表和其它视图）。当视图引用另一个对象时，该视图称为**引用对象**，其所引用的对象称为**被引用对象**。进一步而言，可以说引用对象**依赖于**其所引用的对象。

给定视图的被引用对象集包括该视图直接或间接引用的所有对象。例如，视图可以通过引用另一个引用该表的视图间接引用表。

假设有以下一组表和视图：

```
CREATE TABLE t1 ( c1 INT, c2 INT );
CREATE TABLE t2( c3 INT, c4 INT );
CREATE VIEW v1 AS SELECT * FROM t1;
CREATE VIEW v2 AS SELECT c3 FROM t2;
CREATE VIEW v3 AS SELECT c1, c3 FROM v1, v2;
```

通过上面的定义可以确定以下视图依赖性：

- 视图 v1 依赖于 t1 的各个列及 t1 本身。
- 视图 v2 依赖于 t2.c3 及 t2 本身。
- 视图 v3 依赖于列 t1.c1 和 t2.c3、表 t1 和 t2 以及视图 v1 和 v2。

数据库服务器会跟踪给定视图所引用的列、表和视图。数据库服务器使用这些依赖性信息来确保对被引用对象的模式更改不会使引用视图处于不能使用的状态。下表说明视图依赖性如何影响常规视图和实例化视图。

依赖性和模式变更更改

尝试变更为表或视图定义的模式时，要求数据库服务器考虑是否有受更改影响的相关视图。举例来说，模式变更操作包括：

- 删除表、视图、实例化视图或列
- 重命名表、视图、实例化视图或列

- 添加、删除或变更列
- 变更列的数据类型、大小或为空性
- 禁用视图或表的视图依赖性

当尝试进行模式变更操作时，会发生以下事件：

1. 数据库服务器生成直接或间接依赖于所变更表或视图的视图的列表。忽略状态为 DISABLED 的视图。

   如果有任何相关视图为实例化视图，请求即会失败并返回错误，剩余的事件也就不会发生。您必须显式禁用相关实例化视图，然后才能继续模式变更操作。请参见“启用和禁用实例化视图”一节第 60 页。
2. 数据库服务器获得所变更的对象以及所有相关常规视图的独占模式锁。
3. 数据库服务器将所有相关常规视图的状态设置为 INVALID。
4. 数据库服务器执行模式变更操作。如果操作失败，锁会被释放，相关常规视图的状态重置为 VALID，返回错误，下一步骤也不会发生。
5. 数据库服务器重新编译相关常规视图，编译成功后将各视图的状态设置为 VALID。如果有任何常规视图的编译失败，该视图会保持 INVALID 状态。后续对 INVALID 常规视图的请求会使数据库服务器尝试重新编译该视图。如果后续尝试也失败，可能需要变更 INVALID 视图或其所依赖的对象。

依赖性和模式变更更改（常规视图）

- 常规视图可引用表或视图，包括实例化视图。
- 更改表或视图的模式时，数据库将自动尝试重新编译所有引用常规视图。
- 禁用或删除视图或表时，也自动禁用所有相关常规视图。
- 可以使用 ALTER TABLE 语句的 DISABLE VIEW DEPENDENCIES 子句禁用相关常规视图。

依赖性和模式变更更改（实例化视图）

- 实例化视图只能引用基表。
- 如果有任何启用的实例化视图引用基表，则不允许更改基表模式。但是，可以向表中添加外键（例如，ALTER TABLE ADD FOREIGN KEY）。
- 删除表之前，必须禁用或删除所有相关实例化视图。
- ALTER TABLE 语句的 DISABLE VIEW DEPENDENCIES 子句不影响实例化视图。要禁用实例化视图，则必须使用 ALTER MATERIALIZED VIEW ...DISABLE 语句来禁用每个相关实例化视图。
- 禁用实例化视图后，必须显式重新启用它，例如使用 ALTER MATERIALIZED VIEW ...ENABLE 语句。

目录中的视图依赖性信息

数据库服务器会跟踪直接依赖性。直接依赖性是指某个对象在其定义中直接引用另一对象。数据库服务器也使用直接依赖性信息来确定间接依赖性。例如，假定视图 A 引用视图 B，而视图 B 引用表 C。这种情况下，视图 A 直接依赖于视图 B，间接依赖于表 C。

SYSDEPENDENCY 系统视图会存储依赖性信息。SYSDEPENDENCY 系统视图中的各行用来描述两个数据库对象间的依赖性。请参见“SYSDEPENDENCY 系统视图”一节 《SQL Anywhere 服务器 - SQL 参考》。

也可以使用 sa_dependent_views 系统过程来返回依赖于给定表或视图的视图列表。请参见“sa_dependent_views 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》。

使用常规视图

浏览数据时，查询对一个或多个数据库对象进行操作，并产生一个结果集。与基表类似，查询的结果集具有列和行。视图为特定的查询指定一个名称，并在数据库系统表中保存定义。

假设您经常需要列出每个部门的雇员数。您可以使用以下语句得到该列表：

```
SELECT DepartmentID, COUNT(*)
FROM Employees
GROUP BY DepartmentID;
```

您可以使用 Sybase Central 或 Interactive SQL 创建包含此语句结果的视图。

常规视图 SELECT 语句的限制

对可用作常规视图的 SELECT 语句有一些限制。特别是您不能在 SELECT 查询中使用 ORDER BY 子句。关系表的一个特性是行或列的顺序没有特殊意义，而使用 ORDER BY 子句将在视图的行上强加顺序。可以在视图定义中使用 GROUP BY 子句、子查询和连接。

要规划一个视图，可以调整 SELECT 查询本身，直到其以所需格式提供完全符合需要的结果。调整好 SELECT 语句之后，即可在查询前面添加一个短语来创建视图：

```
CREATE VIEW view-name AS query;
```

更新常规视图

如果定义视图的查询说明是可更新的，则可以使用 UPDATE、INSERT 或 DELETE 语句对视图进行更新。如果在视图的查询说明中其定义包括以下内容中的任何一个，则将视图视为固有*不可更新*。

- UNION 子句
- DISTINCT 子句
- GROUP BY 子句
- FIRST 或 TOP 子句
- 集合函数
- 当 ansi_update_constraints 选项设置为 'Strict' 或 'Cursor' 时，FROM 子句中有多个表。请参见“ansi_update_constraints 选项 [兼容性]”一节 《SQL Anywhere 服务器 - 数据库管理》。
- 当 ansi_update_constraints 选项设置为 'Strict' 或 'Cursor' 时，存在 ORDER BY 子句。请参见“ansi_update_constraints 选项 [兼容性]”一节 《SQL Anywhere 服务器 - 数据库管理》。
- 所有 *select-list* 项都不是基表列

复制常规视图

在 Sybase Central 中，您可以在数据库之间复制视图。若要执行此操作，请在 Sybase Central 的右窗格中选择视图，然后将其拖到另一个已连接数据库的 [**视图**] 文件夹中。这样即会创建一个新视图，原始视图的定义会复制到其中。请注意，只有视图定义会复制到新视图中，其它视图属性（例如权限）不会随之复制。

使用 WITH CHECK OPTION 选项

WITH CHECK OPTION 子句可用于控制在通过视图更新基表或在其中插入内容时所更改的数据。以下示例对此进行说明。

执行以下语句以创建带有 WITH CHECK OPTION 子句的 SalesEmployees 视图。

```
CREATE VIEW SalesEmployees AS
   SELECT EmployeeID, GivenName, Surname, DepartmentID
   FROM Employees
   WHERE DepartmentID = 200
   WITH CHECK OPTION;
```

选择查看此视图的内容，如下所示：

```
SELECT * FROM SalesEmployees;
```

EmployeeID	GivenName	Surname	DepartmentID
129	Philip	Chin	200
195	Marc	Dill	200
299	Rollin	Overbey	200
467	James	Klobucher	200
...	...	...	...

接下来，尝试将 Philip Chin 的 DepartmentID 更新为 400：

```
UPDATE SalesEmployees
SET DepartmentID = 400
WHERE EmployeeID = 129;
```

由于指定了 WITH CHECK OPTION，数据库服务器会计算更新操作是否违反视图定义中的规则（在本例中，为 WHERE 子句中的表达式）。该语句将失败（DepartmentID 必须为 200），数据库服务器将返回错误："在基本表 'Employees' 中插入/更新时 WITH CHECK OPTION 违规。"

如果未在视图定义中指定 WITH CHECK OPTION，更新操作则会继续，用新值修改 Employees 表，进而使 Philip Chin 从视图中消失。

如果创建一个引用 SalesEmployees 视图的视图（例如 View2），那么即使在定义 View2 时未使用 WITH CHECK OPTION 子句，在 View2 上所做的任何更新或插入只要导致 SalesEmployees 上的 WITH CHECK OPTION 条件失败就都会被拒绝。

另请参见

- “SELECT 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “对查询结果进行汇总、分组和排序”第 339 页
- “使用实例化视图”一节第 46 页

常规视图状态

常规视图都与某种状态相关联。状态反映了视图对数据库服务器的可用性。您可以在 Sybase Central 的左窗格中选择 [**视图**] 并检查右窗格中 [**状态**] 列的值，来查看所有视图的状态。如果要查看单个视图的状态，请在 Sybase Central 中右击该视图，然后选择 [**属性**] 以查看 [**状态**] 值。

以下是常规视图的可能状态的说明：

- **有效** 视图有效并保证与其定义一致。数据库服务器无需任何附加工作即可利用此视图。已启用的视图具有 [有效] 状态。

  在 SYSOBJECT 系统视图中，值 1 指示 [有效] 状态。请参见“SYSOBJECT 系统视图”一节《SQL Anywhere 服务器 - SQL 参考》。

- **无效** 如果对被引用对象的模式更改导致不能启用该视图，则视图会在模式更改后变为 INVALID 状态。例如，假定视图 v1 引用表 t 中的列 c1。如果通过变更语句删除 t 中的 c1，当数据库服务器在用于删除列的 ALTER 操作期间尝试重新编译视图时，v1 的状态会被设置为 [无效]。这种情况下，只有将 c1 添加回 t 或将 v1 改为不再引用 c1，v1 才能重新编译。如果视图所引用的某个表或视图被删除，该视图的状态也可能会变为 INVALID。

  INVALID 视图与 DISABLED 视图的区别在于：每次引用 INVALID 视图时（例如，由某个查询引用），数据库服务器都会尝试重新编译该视图。如果编译成功，查询会继续进行。视图的状态会保持为 [无效]，直到显式启用该视图。如果编译失败，则返回错误。

  数据库服务器在内部启用 [无效] 视图时，会发出性能警告。

  在 SYSOBJECT 系统视图中，值 2 指示 [无效] 状态。请参见“SYSOBJECT 系统视图”一节《SQL Anywhere 服务器 - SQL 参考》。

- **已禁用** 数据库服务器不能使用处于 DISABLE 状态的视图来应答查询。任何尝试使用禁用视图的查询都会返回错误。

  以下情况下常规视图会处于此状态：

  - 您以某种方式显式禁用了该视图，例如通过执行 ALTER VIEW ...DISABLE 语句来禁用每个相关实例化视图。
  - 您禁用了该视图所依赖的某个实例化或非实例化视图。
  - 您以某种方式禁用了某个表的视图依赖性，例如通过执行 ALTER TABLE ...DISABLE VIEW DEPENDENCIES 语句。

  有关启用和禁用常规视图的信息，请参见“启用和禁用常规视图”一节第 42 页。

  在 SYSOBJECT 系统视图中，值 4 指示 DISABLED 状态。请参见“SYSOBJECT 系统视图”一节 《SQL Anywhere 服务器 - SQL 参考》。

创建常规视图

创建常规视图时，数据库服务器将视图定义存储在数据库中；但该视图的任何数据都不会存储下来。而视图定义也仅在其被引用时、在使用视图的时间段内被执行。这意味着创建视图并不需要在数据库中存储重复数据。

♦ **创建新常规视图 (Sybase Central)**

1. 以具有 DBA 权限的用户身份连接到数据库。
2. 在左窗格中，右击 [**视图**] 并选择 [**新建**] » [**视图**]。
3. 按照 [**创建视图向导**] 中的说明操作。
4. 在右窗格中，单击 [**SQL**] 选项卡以编辑视图定义。要保存更改，请选择 [**文件**] » [**保存**]。

♦ **创建新常规视图 (SQL)**

1. 连接到数据库。
2. 执行 CREATE VIEW 语句。

示例

创建一个名为 DepartmentSize 的视图，其中包含本节前面部分中给出的 SELECT 语句的结果：

```
CREATE VIEW DepartmentSize AS
   SELECT DepartmentID, COUNT(*)
   FROM Employees
   GROUP BY DepartmentID;
```

请参见“CREATE VIEW 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

变更常规视图

可以使用 Sybase Central 或 Interactive SQL 变更常规视图。

在 Sybase Central 中，可以在右窗格中对象的 [**SQL**] 选项卡上变更视图、过程和函数的定义。通过选择视图然后选择 [**文件**] » [**在新建窗口中编辑**]，可以在单独的窗口中编辑视图。在 Interactive SQL 中，可以使用 ALTER VIEW 语句变更视图。ALTER VIEW 语句会用新视图定义替换原定义，但保留视图上的权限。

不能重命名现有视图。而必须以新名称创建一个新视图，将以前的定义复制到新视图中，然后删除旧视图。

如果使用 ALTER VIEW 语句变更另一个用户拥有的视图，必须通过包含该所有者来限定名称（例如，GROUPO.EmployeeConfidential）。如果不限定名称，数据库服务器就会查找您拥有的该名称的视图，并对其进行变更。如果没有这样的视图，服务器将返回错误。

视图变更与视图依赖性

如果要变更某个常规视图的定义，但有其它视图依赖于该视图，则变更完成后可能需要采取一些附加步骤。例如，变更某个视图后，数据库服务器会自动重新进行编译，使数据库服务器能够使用。如果存在相关常规视图，数据库服务器也会禁用并重新启用这些视图。如果无法启用相关视图，则其状态被指定为 INVALID，您必须使所变更常规视图的定义与相关常规视图的定义一致，或者使相关常规视图的定义与所变更常规视图的定义一致。

要确定某个常规视图是否存在相关视图，请使用 sa_dependent_views 系统过程。请参见“sa_dependent_views 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》。

有关基础对象的模式变更如何影响视图的信息，请参见“视图依赖性”一节第 34 页。

♦ 变更常规视图 (Sybase Central)

1. 以具有 DBA 权限的用户身份或以常规视图所有者身份连接到数据库。
2. 在左窗格中，双击 [**视图**]。
3. 选择视图。
4. 在右窗格中，单击 [**SQL**] 选项卡，然后编辑视图的定义。

> **提示**
> 如果要编辑多个视图，可以为每个视图单独打开一个窗口，而不是在右窗格的 [**SQL**] 选项卡上编辑每个视图。打开单独窗口的方法是：选择视图，然后选择 [**文件**] » [**在新建窗口中编辑**]。

5. 选择 [**文件**] » [**保存**]。

♦ 变更常规视图 (SQL)

1. 以具有 DBA 权限的用户身份或以常规视图所有者身份连接到数据库。
2. 执行 ALTER VIEW 语句。

示例

此示例说明当变更常规视图时，实际上是在替换视图的定义。在本例中，通过更改视图定义，列名称更具信息性。

```
CREATE VIEW DepartmentSize ( col1, col2 ) AS
   SELECT DepartmentID, COUNT( * )
   FROM Employees GROUP BY DepartmentID;
ALTER VIEW DepartmentSize ( DepartmentNumber, NumberOfEmployees ) AS
   SELECT DepartmentID, COUNT( * )
   FROM Employees GROUP BY DepartmentID;
```

下面的示例说明在仅更改常规视图的某个属性时无需重新定义视图。在本例中，视图被设置为隐藏其定义。

```
ALTER VIEW DepartmentSize SET HIDDEN;
```

请参见“ALTER VIEW 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

删除常规视图

使用 Sybase Central 和 Interactive SQL 都可以删除常规视图。

如果删除具有相关视图的常规视图，则在删除操作过程中相关视图会变成 INVALID 状态。相关视图会一直保持不可用状态，直到进行了更改或者重新创建最初所删除的视图。请参见“变更常规视图”一节第 40 页。

要确定某个常规视图是否存在相关视图，请使用 sa_dependent_views 系统过程。请参见“sa_dependent_views 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》。

有关基础对象的更改如何影响常规视图的信息，请参见“视图依赖性”一节第 34 页。

- **删除常规视图 (Sybase Central)**

1. 以具有 DBA 权限的用户身份或以常规视图所有者身份连接到数据库。
2. 在左窗格中，双击 [**视图**]。
3. 右击视图，然后选择 [**删除**]。
4. 单击 [**是**]。

- **删除常规视图 (SQL)**

1. 以具有 DBA 权限的用户身份或以常规视图所有者身份连接到数据库。
2. 执行 DROP VIEW 语句。

示例

删除名为 DepartmentSize 的常规视图。

```
DROP VIEW DepartmentSize;
```

请参见“DROP VIEW 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

启用和禁用常规视图

本节说明启用和禁用常规视图。有关启用和禁用实例化视图的信息，请参见“启用和禁用实例化视图”一节第 60 页。

可以通过启用或禁用常规视图来控制数据库服务器能否使用该视图。禁用某个常规视图时，数据库服务器将该视图的定义保留在数据库中；但该视图不能用来满足某个查询的条件。如果某个查询显式地引用禁用的视图，则查询会失败并返回错误。视图一旦被禁用后，必须显式重新启用，数据库服务器才能使用它。

如果禁用某个视图，直接或间接引用该视图的其它视图会自动被禁用。因此，重新启用视图后，必须重新启用在其被禁用时与其相关的所有其它视图。禁用视图前，可以使用 sa_dependent_views 系统过程来确定相关视图的列表。请参见“sa_dependent_views 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》。

启用常规视图时，数据库服务器会使用数据库中存储的该视图的定义来将其重新编译。如果编译成功，视图状态更改为 VALID。如果重新编译未成功，说明模式可能在一个或多个被引用对象中被更改。如果是这样，必须更改视图定义或更改被引用对象，直到它们彼此一致，然后再启用视图。

注意
启用常规视图前，必须重新启用和禁用其所引用的视图。

可以对禁用的对象授予权限。对禁用对象的权限存储在数据库中，在对象被启用时生效。

♦ 禁用常规视图 (Sybase Central)

1. 以具有 DBA 权限的用户身份或以常规视图所有者身份连接到数据库。
2. 在左窗格中，双击 [**视图**]。
3. 右击视图，然后选择 [**禁用**]。

♦ 禁用常规视图 (SQL)

1. 以具有 DBA 权限的用户身份或以常规视图所有者身份连接到数据库。
2. 执行 ALTER VIEW ...DISABLE 语句来禁用每个相关实例化视图。

示例

以下示例会禁用 GROUPO 拥有的名为 ViewSalesOrders 的常规视图。

```
ALTER VIEW GROUPO.ViewSalesOrders DISABLE;
```

♦ 启用常规视图 (Sybase Central)

1. 以具有 DBA 权限的用户身份或以常规视图所有者身份连接到数据库。
2. 在左窗格中，双击 [**视图**]。
3. 右击视图，然后选择 [**重新编译和启用**]。

♦ 启用常规视图 (SQL)

1. 以具有 DBA 权限的用户身份或以常规视图所有者身份连接到数据库。
2. 执行 ALTER VIEW ...ENABLE 语句。

示例

以下示例重新启用 GROUPO 拥有的名为 ViewSalesOrders 的常规视图。

```
ALTER VIEW GROUPO.ViewSalesOrders ENABLE;
```

另请参见

- “sa_dependent_views 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》
- “ALTER VIEW 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “SYSDEPENDENCY 系统视图”一节 《SQL Anywhere 服务器 - SQL 参考》

浏览常规视图中的数据

要浏览视图中保存的数据，可以使用 Interactive SQL。Interactive SQL 允许通过执行查询来确定要查看的数据。请参见“查询数据”第 253 页。

使用 Sybase Central 时，可以选择您拥有权限的视图，然后选择 [**文件**] » [**在 Interactive SQL 中查看数据**]。此命令将打开 Interactive SQL，视图的内容会显示在 [**结果**] 窗格中的 [**结果**] 选项卡上。要浏览视图，Interactive SQL 执行 `SELECT * FROM owner.view` 语句。

另请参见

- “使用 Interactive SQL”一节 《SQL Anywhere 服务器 - 数据库管理》

查看系统表数据

只能通过查询系统视图查看系统表中的数据；不能直接查询系统表。除了少数系统表外，每个系统表都有相应视图。

系统视图的命名类似于系统表，只是开头没有 I。例如，要查看 ISYSTAB 系统表中的数据，可以查询 SYSTAB 系统视图。

有关 SQL Anywhere 所提供视图的列表及各视图所包含信息类型的说明，请参见“系统视图”一节 《SQL Anywhere 服务器 - SQL 参考》。

可以使用 Sybase Central 或 Interactive SQL 浏览系统视图数据。

♦ 通过系统视图查看系统表数据 (Sybase Central)

1. 以具有 DBA 权限的用户身份连接到数据库。
2. 在左窗格中，双击 [**视图**]。
3. 选择与系统表对应的视图。
4. 在右窗格中，单击 [**数据**] 选项卡。

♦ 通过系统视图查看系统表数据 (SQL)

1. 以具有 DBA 权限的用户身份连接到数据库。
2. 执行一个引用与系统表对应的系统视图的 SELECT 语句。

示例

假定您要查看 ISYSTAB 系统表中的数据。因为不能直接查询该表，以下语句将显示相应的 SYSTAB 系统视图中的所有数据：

```
SELECT * FROM SYS.SYSTAB;
```

有时，在系统表中存在的列在相应的系统视图中并不存在。要抽取包含特定视图定义的文本文件，请使用如下语句：


```
SELECT viewtext
FROM SYS.SYSVIEWS
WHERE viewname = 'SYSTAB';
OUTPUT TO viewtext.sql
FORMAT TEXT
ESCAPES OFF
QUOTE '';
```

使用实例化视图

实例化视图 是指其结果集已经计算出来并像基表一样存储在磁盘上的视图。从概念上讲，实例化视图既是视图（在目录中存储有查询说明）又是表（有持久实例化的行）。因此，许多对表执行的操作也可以对实例化视图执行。例如，可以在实例化视图上建立索引，也可以从实例化视图卸载。

考虑对频繁执行且开销庞大的查询（如涉及集中的集合和连接操作的查询）使用实例化视图。实例化视图提供了用以存储集合化、连接化数据的可查询结构。对于数据库规模大、频繁的查询引发大量数据的重复性集合和连接操作的环境而言，使用实例化视图可以提高性能。例如，实例化视图非常适于在数据仓库应用程序中使用。

实例化视图使用它们所引用的基表的数据预先计算。实例化视图为只读；不能对它们使用数据变更操作，如 INSERT、LOAD、DELETE 和 UPDATE。

实例化视图的列统计信息的生成和维护方式与表完全相同。请参见“优化程序估计值和列统计信息”一节第 526 页。

在实例化视图上虽然可以创建索引，但不能创建键、约束、触发器或项目。

另请参见

- “使用实例化视图提高性能”一节第 536 页
- “CREATE MATERIALIZED VIEW 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “ALTER MATERIALIZED VIEW 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “REFRESH MATERIALIZED VIEW 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “sa_materialized_view_info 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》
- “sa_materialized_view_can_be_immediate 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》
- “sa_refresh_materialized_views 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》

手动和快速实例化视图

实例化视图有两种类型：手动和快速，这两种类型也表明了实例化视图的**刷新类型**。

- **手动视图** 手动实例化视图或**手动视图** 是刷新类型定义为 MANUAL REFRESH 的实例化视图。手动视图中的数据会变为失效，因为在明确请求刷新前，不刷新手动视图，例如通过使用 REFRESH MATERIALIZED VIEW 语句或 sa_refresh_materialized_views 系统过程。缺省情况下，创建的实例化视图是手动视图。

  基础表有任何更改时，手动视图即视为失效，即使该更改不影响实例化视图中的数据。可以通过检查 sa_materialized_view_info 系统过程返回的 DataStatus 值确定手动视图是否视为失效。如果返回 S，则手动视图失效。

- **快速视图** 快速实例化视图或**快速视图**是刷新类型定义为 IMMEDIATE REFRESH 的实例化视图。基础表更改影响视图中的数据时，会自动刷新快速视图中的数据。如果基础表更改不影响视图中的数据，则不刷新该视图。

  此外，刷新快速视图时，只刷新需要更改的行。这与刷新手动视图不同，刷新手动视图时，会删除所有数据然后重新创建。

可以将手动视图更改为快速视图，也可以将快速视图更改为手动视图。但是，从手动视图更改为快速视图需要更多步骤。请参见“将手动视图更改为快速视图”一节第 57 页。

更改实例化视图的刷新类型会影响视图的状态和属性，特别是将手动视图更改为快速视图时。请参见“实例化视图状态和属性”一节第 48 页。

从数据库检索实例化视图信息

- **状态和属性信息** 可以使用 sa_materialized_view_info 系统过程来请求实例化视图的信息（例如状态）。请参见“sa_materialized_view_info 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》。

  另请参见“实例化视图状态和属性”一节第 48 页。

- **数据库选项信息** 可以通过查询 SYSMVOPTION 系统视图检索创建实例化视图时与实例化视图一起存储的数据库选项。以下语句创建实例化视图，然后查询数据库，以找出在创建视图时所用的数据库选项。

```
CREATE MATERIALIZED VIEW EmployeeConfid15 AS
   SELECT EmployeeID, Employees.DepartmentID, SocialSecurityNumber,
Salary, ManagerID,
      Departments.DepartmentName, Departments.DepartmentHeadID
   FROM Employees, Departments
   WHERE Employees.DepartmentID=Departments.DepartmentID;

SELECT option_name, option_value
FROM SYSMVOPTION JOIN SYSMVOPTIONNAME
WHERE SYSMVOPTION.view_object_id=(
   SELECT object_id FROM SYSTAB
   WHERE table_name='EmployeeConfid15' )
ORDER BY option_name;
```

> **小心**
> 处理该示例时，应删除所创建的实例化视图。否则，在试验其它示例时，将无法对其基础表 Employees 和 Departments 执行模式更改。无法变更具有已启用相关实例化视图的表的模式。请参见“删除实例化视图”一节第 64 页。

- **依赖性信息** 要确定依赖于实例化视图的视图列表，请使用 sa_dependent_views 系统过程。请参见“sa_dependent_views 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》。

  此信息也可以在 SYSDEPENDENCY 系统视图中找到。请参见“SYSDEPENDENCY 系统视图”一节 《SQL Anywhere 服务器 - SQL 参考》。

何时使用实例化视图

使用实例化视图前，应认真考虑以下需求和设置：

- **磁盘空间需求** 由于实例化视图包含基表数据的副本，因此可能需要为数据库分配额外的磁盘空间来容纳所创建的实例化视图。需要认真考虑额外的空间需求，以便在使用实例化视图的优势与开销之间找到平衡点。

- **维护开销和数据更新度需求** 基础表中的数据更改时需要刷新实例化视图中的数据。需要通过考虑如下潜在的冲突因素来确定刷新实例化视图的频率：
  - **基础数据的更改频率** 频繁或大量的数据更改会使手动视图失效。如果数据更新度很重要，考虑使用快速视图。
  - **刷新的开销** 根据每个实例化视图的基础查询的复杂程度和涉及数据量的大小，刷新所需的计算可能会需要庞大的开销，频繁刷新实例化视图可能会给数据库服务器带来无法承受的负荷。此外，刷新操作期间实例化视图不可用。
  - **应用程序的数据更新度需求** 如果数据库服务器使用失效的实例化视图，则它将失效的数据提供给应用程序。失效数据是不再表示基础表中数据的当前状态的数据。过期失效程度受实例化视图的刷新频率制约。应用程序必须能够确定其可以容许的失效程度，以提高性能。有关管理实例化视图中数据失效的详细信息，请参见“设置优化程序的实例化视图失效程度阈值”一节第 63 页。
  - **数据一致性需求** 刷新实例化视图时，必须确定刷新实例化视图时所应保持的一致性。请参见“REFRESH MATERIALIZED VIEW 语句”一节 《SQL Anywhere 服务器 - SQL 参考》的 WITH ISOLATION LEVEL 子句。
- **在优化中使用** 您应验证优化程序在执行查询时是否考虑到了实例化视图。通过在 Interactive SQL 中查看查询的图形式计划的 [**高级详细信息**] 窗口，可以看到用于特定查询的实例化视图的列表。请参见“读取执行计划”一节第 569 页和“使用实例化视图提高性能”一节第 536 页。

  通过查看由优化程序枚举的访问计划，也可以在 Sybase Central 中使用 [应用程序分析] 模式来确定查询的枚举阶段是否将实例化视图考虑在内。要查看优化程序所枚举的访问计划，跟踪功能必须打开并且必须配置为包括 OPTIMIZATION_LOGGING 跟踪类型。请参见“应用程序分析”一节第 161 页和“选择诊断跟踪级别”一节第 173 页。

另请参见

- “手动和快速实例化视图”一节第 46 页
- “使用实例化视图提高性能”一节第 536 页

实例化视图状态和属性

实例化视图用状态和属性的结合来实现特征化。实例化视图的**状态**反映视图是否可由数据库服务器使用。实例化视图的**属性**反映视图中数据的状态。

确定现有实例化视图的状态和属性的最佳方法是使用 sa_materialized_view_info 系统过程。请参见“sa_materialized_view_info 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》。

还可以通过以下方法查看有关实例化视图的信息：在 Sybase Central 中选择 [**视图**] 文件夹，然后查看为各个视图提供的详细信息。或者通过查询 SYSTAB 和 SYSVIEW 系统视图。请参见“SYSTAB 系统视图”一节 《SQL Anywhere 服务器 - SQL 参考》和“SYSVIEW 系统视图”一节 《SQL Anywhere 服务器 - SQL 参考》。

实例化视图状态

实例化视图有两个可能状态：

- **已启用** 如果实例化视图已成功编译，可供数据库服务器使用，则实例化视图状态为**已启用**。已启用实例化视图中可能没有数据。例如，如果截断已启用实例化视图中的数据，它将更改为已启用和未初始化。如果在指定实例化视图的定义的基础表中没有数据，实例化视图可以初始化但为空。这与实例化视图没有数据不同，因为它未初始化。
- **已禁用** 实例化视图只有在以某种方式显式禁用后才会具有**已禁用**状态，例如使用 ALTER MATERIALIZED VIEW ...DISABLE 语句来禁用每个相关实例化视图。如果禁用实例化视图，将删除视图的数据和索引。此外，禁用快速视图时，它将更改为手动视图。

要确定视图是已启用还是已禁用，请使用 sa_materialized_view_info 系统过程返回视图的 Status 属性。请参见“sa_materialized_view_info 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》。

有关启用和禁用实例化视图的信息，请参见“启用和禁用实例化视图”一节第 60 页。

实例化视图属性

优化程序计算是否使用视图时使用实例化视图属性。下表介绍 sa_materialized_view_info 系统过程返回的实例化视图的属性：

- **Status** Status 属性指示视图是已启用还是已禁用。
- **DataStatus** DataStatus 属性反映视图中数据的状态。例如，通过该属性可了解视图是否已初始化以及视图是否已失效。如果自上次刷新实例化视图以后基础表中的数据已更改，则手动视图失效。快速视图从不失效。
- **ViewLastRefreshed** ViewLastRefreshed 属性指出上次刷新视图的时间。
- **DateLastModified** DateLastModified 属性指出视图失效时，任何基础表中数据的最新修改时间。
- **AvailForOptimization** AvailForOptimization 属性反映视图是否可供优化程序使用。
- **RefreshType** RefreshType 属性指出视图是手动视图还是快速视图。

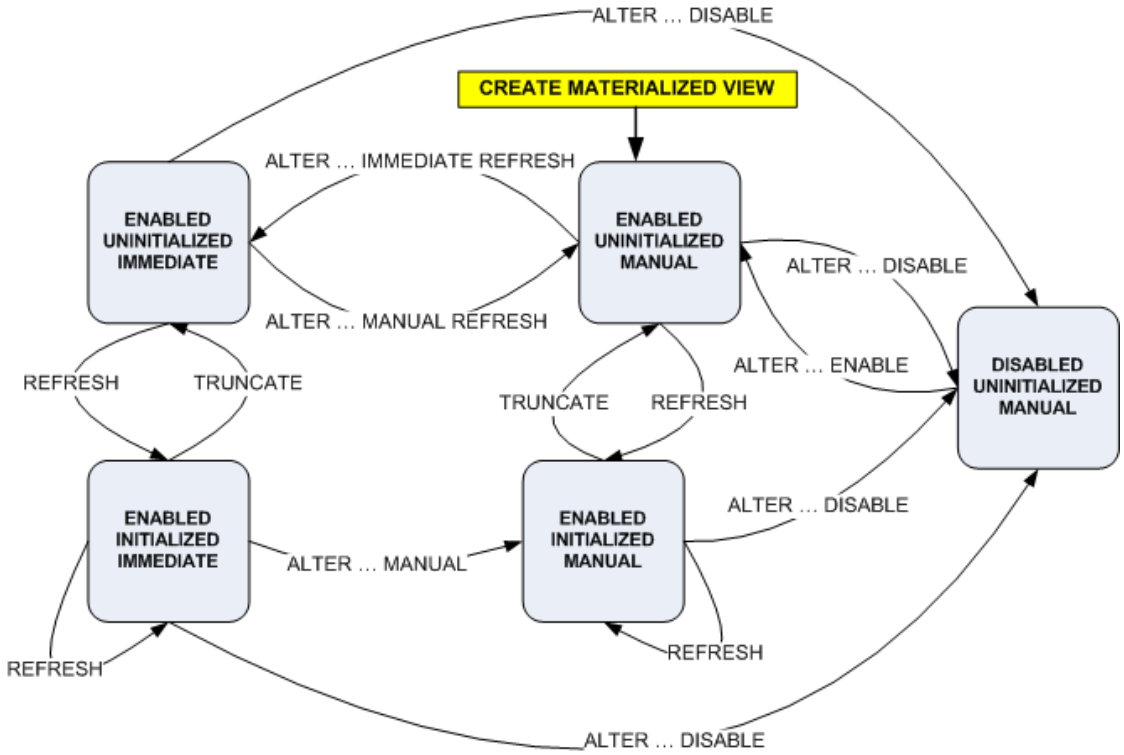
有关每个属性的可能值的列表，请参见“sa_materialized_view_info 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》。

虽然没有属性能让您了解是否能够将手动视图转换为快速视图，但您可以使用 sa_materialized_view_can_be_immediate 系统过程确定此问题。请参见“sa_materialized_view_can_be_immediate 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》。

变更、刷新和截断实例化视图时状态和属性的更改

对实例化视图执行的操作（如变更、刷新和截断）影响视图的状态和属性。下图显示这些任务如何影响实例化视图的状态和某些属性。

在该图中，每个灰色方块是一个实例化视图；快速视图由术语 IMMEDIATE 标识，手动视图由术语 MANUAL 标识。灰色方框之间的连接符中的术语 ALTER 是 ALTER MATERIALIZED VIEW 的简写。尽管图中显示了用于更改实例化视图状态的 SQL 语句，但您也可以使用 Sybase Central 执行这些活动。



图中需要注意的一些重要概念如下：

- 创建实例化视图时，视图是已启用手动视图，并且未初始化（不包含数据）。
- 刷新未初始化视图时，视图变为已初始化（填入了数据）。
- 从手动视图更改为快速视图需要几个步骤，对于快速视图还有更多限制。请参见“将手动视图更改为快速视图”一节第 57 页和“快速视图的附加限制”一节第 52 页。
- 禁用实例化视图时：
  - 数据被删除
  - 视图转为未初始化
  - 索引被删除
  - 快速视图转为手动视图

另请参见

- “使用实例化视图”一节第 46 页
- “CREATE MATERIALIZED VIEW 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “ALTER MATERIALIZED VIEW 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “REFRESH MATERIALIZED VIEW 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “TRUNCATE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “DROP MATERIALIZED VIEW 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “实例化视图的限制”一节第 51 页
- “手动和快速实例化视图”一节第 46 页
- “SYSOBJECT 系统视图”一节 《SQL Anywhere 服务器 - SQL 参考》

实例化视图的限制

创建、初始化、刷新和查看匹配的实例化视图时，以下限制适用：

- 创建实例化视图时，实例化视图的定义必须明确定义列名称；列定义中不能包括 `SELECT *` 结构。
- 创建实例化视图时，其定义中不能包含：
  - 对其它实例化或非实例化视图的引用
  - 对远程表或临时表的引用
  - 变量，如 CURRENT USER；所有表达式必须是确定性表达式
  - 对存储过程、用户定义的函数或外部函数的调用
  - T-SQL 外连接
  - FOR XML 子句
- 创建实例化视图时，以下数据库选项必须具有指定的设置；否则将返回错误。要使优化程序能够使用该视图，也需要这些数据库选项：
  - ansinull=On
  - conversion_error=On
  - sort_collation=Null
  - string_rtruncation=On

- 创建实例化视图时，将存储每个视图的以下数据库选项设置。只有连接的当前选项值与为实例化视图存储的值相匹配，该视图才能在优化中使用：
  - date_format
  - date_order
  - default_timestamp_increment
  - first_day_of_week
  - nearest_century
  - precision
  - scale
  - time_format
  - timestamp_format
- 刷新视图时，将忽略上面列出的所有选项的连接设置。改用数据库选项设置（这些设置必须与视图的存储设置匹配）。

在实例化视图定义中指定 ORDER BY 子句

实例化视图的行不会以任何特定顺序存储，这一点与基表相似；数据库服务器会在计算数据时以最有效的方式对行进行排序。因此，在实例化视图定义中指定 ORDER BY 子句并不会对视图在实现后的行顺序产生必然影响。此外，执行视图匹配时，优化程序也会忽略视图定义中的 ORDER BY 子句。

有关实例化视图和优化程序执行的视图匹配的信息，请参见“使用实例化视图提高性能”一节 第 536 页。

快速视图的附加限制

手动视图更改为快速视图时，会检查以下限制。如果视图违反任何限制，则会返回错误：

注意
可以使用 sa_materialized_view_can_be_immediate 系统过程检查手动视图是否可以转变为快速视图。请参见“sa_materialized_view_can_be_immediate 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》。

- 视图必须未初始化。请参见“实例化视图状态和属性”一节第 48 页。
- 视图必须在不可为空的列上有唯一索引。如果没有，则必须添加一个。请参见“创建索引”一节第 68 页。
- 如果视图定义是分组查询，唯一索引列必须与非集合函数的选择列表项对应。

- 视图定义不能包含：
  - 表达式可为空的 SUM 函数
  - GROUPING SETS 子句
  - CUBE 子句
  - ROLLUP 子句
  - LEFT OUTER JOIN 子句
  - RIGHT OUTER JOIN 子句
  - FULL OUTER JOIN 子句
  - DISTINCT 子句
  - 行限制子句
  - 非确定型表达式
  - 自连接和递归连接
- 视图定义必须为单一选择项目连接或分组选择项目连接查询块，分组选择项目连接查询块不能包含 HAVING 子句。
- 分组选择项目连接查询块的选择列表中必须包含 COUNT(*)，并且只允许 SUM 和 COUNT 集合函数。

  有关这些结构的说明，请参见“实例化视图评估”一节第 539 页。
- 复杂表达式中不能引用选择列表中的集合函数。例如，在选择列表中不允许 `SUM(expression) + 1`。

创建实例化视图

创建实例化视图时，其定义存储在数据库中。数据库服务器会验证其定义，以确保编译正确。数据库服务器会对所有列和表引用进行完全限定，以确保对视图具有访问权的所有用户看到相同的定义。成功创建实例化视图后，使用 REFRESH MATERIALIZED VIEW 语句在视图中填入数据，也称为**初始化**视图。请参见“REFRESH MATERIALIZED VIEW 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

创建、初始化或刷新实例化视图前，应确保所有实例化视图的限制均已得到满足。请参见“实例化视图的限制”一节第 51 页。

要获得数据库中所有实例化视图的列表（包括各视图状态），请使用 sa_materialized_view_info 系统过程。请参见“sa_materialized_view_info 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》。

创建完实例化视图的定义后，它会出现在 Sybase Central 的 [**视图**] 文件夹中。

另请参见

- “CREATE MATERIALIZED VIEW 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “REFRESH MATERIALIZED VIEW 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “SQL Anywhere 示例数据库” 《SQL Anywhere 11 - 简介》

♦ **创建实例化视图 (Sybase Central)**

1. 以具有 DBA 或 RESOURCE 权限的用户身份连接到数据库。
2. 在左窗格中，右击 [**视图**] 并选择 [**新建**] » [**实例化视图**]。
3. 按照 [**创建实例化视图向导**] 中的说明操作。
4. 初始化实例化视图，以使它包含数据。请参见“初始化实例化视图”一节第 54 页。

小心
处理该示例时，应删除所创建的实例化视图。否则，在试验其它示例时，将无法对其基础表 Employees 和 Departments 执行模式更改。无法变更具有已启用相关实例化视图的表的模式。请参见“删除实例化视图”一节第 64 页。

♦ **创建实例化视图 (SQL)**

1. 以具有 DBA 或 RESOURCE 权限的用户身份连接到数据库。
2. 执行 CREATE MATERIALIZED VIEW 语句。数据库服务器即创建视图定义并将其存储在数据库中，并且将视图的状态设置为 [已启用]。请参见“CREATE MATERIALIZED VIEW 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。
3. 必须初始化实例化视图，以使其包含数据。请参见“初始化实例化视图”一节第 54 页。

示例

以下语句创建一个包含雇员相关信息的实例化视图 EmployeeConfid16，然后对其初始化以在其中填充数据。

```
CREATE MATERIALIZED VIEW EmployeeConfid16 AS
   SELECT EmployeeID, Employees.DepartmentID, SocialSecurityNumber, Salary,
ManagerID,
      Departments.DepartmentName, Departments.DepartmentHeadID
   FROM Employees, Departments
   WHERE Employees.DepartmentID=Departments.DepartmentID;
REFRESH MATERIALIZED VIEW EmployeeConfid16;
```

小心
使用完该示例后，应删除所创建的实例化视图。否则，在试验其它示例时，将无法对其基础表 Employees 和 Departments 执行模式更改。无法变更具有已启用相关实例化视图的表的模式。请参见“删除实例化视图”一节第 64 页。

初始化实例化视图

实例化视图必须经过初始化，才能供数据库服务器使用。要初始化实例化视图，可刷新该视图。如果刷新尝试失败，实例化视图会恢复为未初始化状态。

创建、初始化或刷新实例化视图前，应确保所有实例化视图的限制均已得到满足。请参见“实例化视图的限制”一节第 51 页。

注意
也可以使用 sa_refresh_materialized_views 系统过程同时初始化所有未初始化的实例化视图。请参见“sa_refresh_materialized_views 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》。

♦ 初始化实例化视图 (Sybase Central)

1. 以具有 DBA 权限的用户身份或以具有实例化视图的 INSERT 权限的用户身份连接到数据库。
2. 在左窗格中，双击 [**视图**]。
3. 右击实例化视图，然后选择 [**刷新数据**]。
4. 选择隔离级别并单击 [**确定**]。

小心
使用完该示例后，应删除所创建的实例化视图。否则，在试验其它示例时，将无法对其基础表 Employees 和 Departments 执行模式更改。无法变更具有已启用相关实例化视图的表的模式。请参见“删除实例化视图”一节第 64 页。

♦ 初始化实例化视图 (SQL)

1. 以具有 DBA 权限的用户身份或以具有实例化视图的 INSERT 权限的用户身份连接到数据库。
2. 执行 REFRESH MATERIALIZED VIEW 语句。

示例

以下语句创建一个实例化视图 EmployeeConfid6，然后对其初始化：

```
CREATE MATERIALIZED VIEW EmployeeConfid6 AS
   SELECT EmployeeID, Employees.DepartmentID, SocialSecurityNumber, Salary,
ManagerID,
      Departments.DepartmentName, Departments.DepartmentHeadID
   FROM Employees, Departments
   WHERE Employees.DepartmentID=Departments.DepartmentID;
REFRESH MATERIALIZED VIEW EmployeeConfid6;
```

小心
使用完该示例后，应删除所创建的实例化视图。否则，在试验其它示例时，将无法对其基础表 Employees 和 Departments 执行模式更改。无法变更具有已启用相关实例化视图的表的模式。请参见“删除实例化视图”一节第 64 页。

另请参见

- “CREATE MATERIALIZED VIEW 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “REFRESH MATERIALIZED VIEW 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “启用和禁用实例化视图”一节第 60 页

刷新手动视图

如果基表发生变化，手动视图即会失效。刷新手动视图是指数据库服务器重新执行视图的查询定义，并用查询的新结果集替换视图数据。刷新可使视图数据与基础数据保持一致。应考虑手动视图数据失效的可接受程度，并制订刷新策略。您的策略应考虑完成刷新所花费的时间，因为在刷新操作期间视图不能用于查询。

还可以设置策略，在该策略中使用事件刷新视图。例如，可以创建事件，以定期刷新。

除了未初始化（不包含数据）之外，快速视图不需要刷新，例如截断之后。

您也可以使用 sa_refresh_materialized_views 系统过程刷新视图。请参见“sa_refresh_materialized_views 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》。

可以使用 materialized_view_optimization 数据库选项来配置一个失效程度阈值，如果超过该阈值，优化程序在处理查询时就不会使用该实例化视图。请参见“设置优化程序的实例化视图失效程度阈值”一节第 63 页。

使用 REFRESH MATERIALIZED VIEW 语句时，可以使用 WITH ISOLATION LEVEL 子句覆盖连接隔离级别。有关刷新实例化视图时如何控制并发性的详细信息，请参见“REFRESH MATERIALIZED VIEW 语句”一节 《SQL Anywhere 服务器 - SQL 参考》的 WITH 子句。

升级具有实例化视图的数据库
建议您在升级数据库服务器后，或重建或升级数据库（以与升级的数据库服务器一起工作）后，刷新实例化视图。

♦ 刷新手动视图 (Sybase Central)

1. 以具有 DBA 权限的用户身份或以具有实例化视图的 INSERT 权限的用户身份连接到数据库。您还必须拥有基础表的 SELECT 权限。
2. 在左窗格中，双击 [**视图**]。
3. 右击实例化视图，然后选择 [**刷新数据**]。
4. 选择隔离级别并单击 [**确定**]。

♦ 刷新手动视图 (SQL)

1. 以具有 DBA 权限的用户身份或以具有实例化视图的 INSERT 权限的用户身份连接到数据库。您还必须拥有基础表的 SELECT 权限。
2. 执行 REFRESH MATERIALIZED VIEW 语句。

示例

以下语句创建 EmployeeConfid33 实例化视图，然后对其刷新。

```
CREATE MATERIALIZED VIEW EmployeeConfid33 AS
   SELECT EmployeeID, Employees.DepartmentID, SocialSecurityNumber, Salary,
ManagerID,
      Departments.DepartmentName, Departments.DepartmentHeadID
   FROM Employees, Departments
```



```
   WHERE Employees.DepartmentID=Departments.DepartmentID;
REFRESH MATERIALIZED VIEW EmployeeConfid33;
```

小心
使用完该示例后，应删除所创建的实例化视图。否则，在试验其它示例时，将无法对其基础表 Employees 和 Departments 执行模式更改。无法变更具有已启用相关实例化视图的表的模式。请参见“删除实例化视图”一节第 64 页。

另请参见

- “将手动视图更改为快速视图”一节第 57 页
- “使用调度和事件自动完成任务” 《SQL Anywhere 服务器 - 数据库管理》
- “materialized_view_optimization 选项 [数据库]”一节 《SQL Anywhere 服务器 - 数据库管理》

将手动视图更改为快速视图

创建实例化视图时，视图的刷新类型为手动。但可将其更改为快速。要从手动更改为快速，视图必须处于未初始化状态（不包含数据）。如果视图刚创建，还未刷新，则它未初始化。如果其中包含数据，则必须截断数据。视图还必须有唯一索引，必须符合快速视图要求的限制。请参见“快速视图的附加限制”一节第 52 页。

只需更改视图的刷新类型即可随时将快速视图转换为手动视图。

以下过程说明如何将手动视图更改为快速视图。在执行其中一个过程之前，请验证手动视图是否有唯一索引且未初始化。然后还可以使用 sa_materialized_view_can_be_immediate 系统过程检查它是否符合立即刷新类型的条件。请参见“sa_materialized_view_can_be_immediate 系统过程”一节《SQL Anywhere 服务器 - SQL 参考》。

♦ 将手动视图更改为快速视图 (Sybase Central)

1. 以具有 DBA 权限的用户身份，或视图及其引用的所有表的所有者身份连接到数据库。
2. 在左窗格中，双击 [**视图**]。
3. 右击实例化视图，并选择 [**属性**]。
4. 在 [**刷新类型**] 字段中，选择 [**立即**]。
5. 单击 [**确定**]。

♦ 将手动视图更改为快速视图 (SQL)

1. 以具有 DBA 权限的用户身份，或视图及其引用的所有表的所有者身份连接到数据库。
2. 通过执行 ALTER MATERIALIZED VIEW ...IMMEDIATE REFRESH 语句将刷新类型更改为立即。

以下过程说明如何将快速视图更改为手动视图。

♦ 将快速视图更改为手动视图 (Sybase Central)

1. 以视图所有者身份或以具有 DBA 权限的用户身份连接到数据库。
2. 在左窗格中，双击 [**视图**]。
3. 右击实例化视图，并选择 [**属性**]。
4. 在 [**刷新类型**] 字段中，选择 [**手工**]。
5. 单击 [**确定**]。

♦ 将快速视图更改为手动视图 (SQL)

1. 以视图所有者身份或以具有 DBA 权限的用户身份连接到数据库。
2. 通过执行 ALTER MATERIALIZED VIEW ...MANUAL REFRESH 语句将刷新类型更改为手动。

示例

以下示例创建一个实例化视图，然后对其初始化。随后，添加唯一索引，因为快速视图必须具有唯一索引。由于在更改刷新类型时，视图内不得包含数据，因而需截断视图。最后，刷新类型得以更改。

```
CREATE MATERIALIZED VIEW EmployeeConfid44 AS
   SELECT EmployeeID, Employees.DepartmentID,
          SocialSecurityNumber, Salary, ManagerID,
          Departments.DepartmentName, Departments.DepartmentHeadID
     FROM Employees, Departments
     WHERE Employees.DepartmentID=Departments.DepartmentID;
REFRESH MATERIALIZED VIEW EmployeeConfid44;
CREATE UNIQUE INDEX EmployeeIDIdx
   ON EmployeeConfid44 ( EmployeeID );
TRUNCATE MATERIALIZED VIEW EmployeeConfid44;
ALTER MATERIALIZED VIEW EmployeeConfid44
   IMMEDIATE REFRESH;
```

以下语句将刷新类型重新更改回手动：

```
ALTER MATERIALIZED VIEW EmployeeConfid44
   MANUAL REFRESH;
```

小心
使用完该示例后，应删除所创建的实例化视图。否则，在试验其它示例时，将无法对其基础表 Employees 和 Departments 执行模式更改。无法变更具有已启用相关实例化视图的表的模式。请参见“删除实例化视图”一节第 64 页。

另请参见

- “ALTER MATERIALIZED VIEW 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “实例化视图状态和属性”一节第 48 页
- “创建索引”一节第 68 页
- “初始化实例化视图”一节第 54 页

加密和解密实例化视图

为了更加安全起见，可以将实例化视图加密。例如，如果实例化视图包含在基础表中已经加密的数据，最好也将实例化视图加密。在数据库中必须已启用表加密，实例化视图才能加密。加密实例化视图时使用在创建数据库时所指定的加密算法和密钥。要查看数据库中当前使用的加密设置（包括是否启用了表加密），请使用 DB_PROPERTY 函数查询 Encryption 数据库属性，如下所示：

```
SELECT DB_PROPERTY( 'Encryption' );
```

与表加密一样，实例化视图的加密也会影响性能，因为数据库服务器必须将从该视图中所检索的数据解密。

♦ 加密实例化视图 (Sybase Central)

1. 以视图所有者身份或以具有 DBA 权限的用户身份连接到数据库。
2. 在左窗格中，双击 [**视图**]。
3. 右击实例化视图，并选择 [**属性**]。
4. 单击 [**杂项**] 选项卡。
5. 选中 [**实例化视图数据已加密**] 复选框。
6. 单击 [**确定**]。

♦ 加密实例化视图 (SQL)

1. 以具有 DBA 权限的用户身份，或以实例化视图所有者身份连接到数据库。
2. 执行带有 ENCRYPTED 子句的 ALTER MATERIALIZED VIEW 语句。

示例

以下语句创建 EmployeeConfid443 实例化视图，并对其初始化，然后进行加密。数据库必须已配置为允许使用加密表，此语句才有效：

```
CREATE MATERIALIZED VIEW EmployeeConfid44 AS
   SELECT EmployeeID, Employees.DepartmentID, SocialSecurityNumber, Salary,
ManagerID,
      Departments.DepartmentName, Departments.DepartmentHeadID
   FROM Employees, Departments
   WHERE Employees.DepartmentID=Departments.DepartmentID;
REFRESH MATERIALIZED VIEW EmployeeConfid44;
ALTER MATERIALIZED VIEW GROUPO.EmployeeConfid44 ENCRYPTED;
```

小心
使用完该示例后，应删除所创建的实例化视图。否则，在试验其它示例时，将无法对其基础表 Employees 和 Departments 执行模式更改。无法变更具有已启用相关实例化视图的表的模式。请参见“删除实例化视图”一节第 64 页。

♦ 解密实例化视图 (Sybase Central)

1. 以视图所有者身份或以具有 DBA 权限的用户身份连接到数据库。

2. 在左窗格中，双击 [**视图**]。
3. 右击实例化视图，并选择 [**属性**]。
4. 单击 [**杂项**] 选项卡。
5. 清除 [**实例化视图数据已加密**] 复选框。
6. 单击 [**确定**]。

♦ 解密实例化视图 (SQL)

1. 以具有 DBA 权限的用户身份，或以实例化视图所有者身份连接到数据库。
2. 执行带有 NOT ENCRYPTED 子句的 ALTER MATERIALIZED VIEW 语句。

示例

以下语句解密 EmployeeConfid44 实例化视图。

```
ALTER MATERIALIZED VIEW GROUPO.EmployeeConfid44 NOT ENCRYPTED;
```

另请参见

- “启用数据库中的表加密”一节 《SQL Anywhere 服务器 - 数据库管理》
- “ALTER MATERIALIZED VIEW 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “DB_PROPERTY 函数 [System]”一节 《SQL Anywhere 服务器 - SQL 参考》

启用和禁用实例化视图

可以通过启用或禁用实例化视图来控制数据库服务器能否使用该视图。优化期间，优化程序也不会考虑禁用的实例化视图。如果某个查询显式地引用禁用的实例化视图，则查询会失败并返回错误。禁用实例化视图时，数据库服务器会删除视图的数据，但保留数据库中的定义。实例化视图在重新启用后处于未初始化状态，必须进行刷新才能在其中填入数据。

禁用实例化视图时，数据库服务器会自动禁用依赖于实例化视图的常规视图。因此，重新启用实例化视图后，必须重新启用所有相关视图。为此，禁用实例化视图前，最好确定出其相关视图的列表。为此可使用 sa_dependent_views 系统过程。此过程会检查 ISYSDEPENDENCY 系统表并返回相关视图的列表（如果有）。

禁用实例化视图时，将删除数据和索引。如果视图是快速视图，会更改为手动视图。因此，重新启用实例化视图时，需要刷新它，重建索引，并将其更改回快速视图（如果需要）。

可以对禁用的对象授予权限。对禁用对象的权限存储在数据库中，在对象被启用时生效。

♦ 禁用实例化视图 (Sybase Central)

1. 以具有 DBA 权限的用户身份，或以实例化视图所有者身份连接到数据库。
2. 在左窗格中，双击 [**视图**]。
3. 右击实例化视图，并选择 [**禁用**]。

♦ 禁用实例化视图 (SQL)

1. 以具有 DBA 权限的用户身份，或以实例化视图所有者身份连接到数据库。
2. 执行 ALTER MATERIALIZED VIEW ...DISABLE 语句来禁用每个相关实例化视图。

示例

以下示例创建 EmployeeConfid55 实例化视图，并对其初始化，然后将其禁用。禁用该实例化视图后，实例化视图的数据将被删除，实例化视图的定义会保留在数据库中，实例化视图不能由数据库服务器使用，并且相关视图将被禁用（如果有）。

```
CREATE MATERIALIZED VIEW EmployeeConfid55 AS
   SELECT EmployeeID, Employees.DepartmentID, SocialSecurityNumber, Salary,
ManagerID,
      Departments.DepartmentName, Departments.DepartmentHeadID
   FROM Employees, Departments
   WHERE Employees.DepartmentID=Departments.DepartmentID;
REFRESH MATERIALIZED VIEW EmployeeConfid55;
ALTER MATERIALIZED VIEW EmployeeConfid55 DISABLE;
```

♦ 启用实例化视图 (Sybase Central)

1. 以具有 DBA 权限的用户身份，或以实例化视图所有者身份连接到数据库。
2. 在左窗格中，双击 [**视图**]。
3. 右击实例化视图，然后选择 [**重新编译和启用**]。
4. 或者，右击视图，然后选择 [**刷新数据**] 以初始化该视图并在其中填充数据。

♦ 启用实例化视图 (SQL)

1. 以具有 DBA 权限的用户身份，或以实例化视图所有者身份连接到数据库。
2. 执行 ALTER MATERIALIZED VIEW ...ENABLE 语句。
3. 或者，执行 REFRESH MATERIALIZED VIEW 以初始化视图并在其中填入数据。

示例

以下两条语句分别重新启用 EmployeeConfid5 实例化视图，然后在其中填入数据。

```
ALTER MATERIALIZED VIEW EmployeeConfid55 ENABLE;
REFRESH MATERIALIZED VIEW EmployeeConfid55;
```

小心
使用完该示例后，应删除所创建的实例化视图。否则，在试验其它示例时，将无法对其基础表 Employees 和 Departments 执行模式更改。无法变更具有已启用相关实例化视图的表的模式。请参见“删除实例化视图”一节第 64 页。

另请参见

- “将手动视图更改为快速视图”一节第 57 页
- “REFRESH MATERIALIZED VIEW 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “CREATE INDEX 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “sa_dependent_views 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》
- “ALTER MATERIALIZED VIEW 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “ALTER VIEW 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “视图依赖性”一节第 34 页
- “SYSDEPENDENCY 系统视图”一节 《SQL Anywhere 服务器 - SQL 参考》

允许和禁止优化程序使用实例化视图

优化程序会维护优化过程中可用实例化视图的列表。如果实例化视图的定义中包含优化程序拒绝的某些元素，或者实例化视图的数据被认为更新度不够而不适于使用，则将不会考虑在优化中使用该视图。有关实例化视图需满足哪些条件才能在优化过程中使用的信息，请参见“使用实例化视图提高性能”一节第 536 页。

缺省情况下，实例化视图可以由优化程序使用。但是，您可以禁止优化程序使用实例化视图，除非该视图在查询中被显式引用。

要确定允许还是禁止优化程序使用实例化视图，请使用 sa_materialized_view_info 系统过程。请参见“sa_materialized_view_info 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》。

♦ 允许在优化中使用实例化视图 (Sybase Central)

1. 以具有 DBA 权限的用户身份连接到数据库。
2. 在左窗格中，双击 [**视图**]。
3. 右击实例化视图，并选择 [**属性**]。
4. 单击 [**常规**] 选项卡，然后选择 [**在优化中使用**]。
5. 单击 [**确定**]。

♦ 允许在优化中使用实例化视图 (SQL)

1. 以具有 DBA 权限的用户身份，或以实例化视图所有者身份连接到数据库。
2. 执行带有 ENABLE USE IN OPTIMIZATION 子句的 ALTER MATERIALIZED VIEW 语句。

示例

以下语句启用 EmployeeConfid77 视图以供优化使用：

```
ALTER MATERIALIZED VIEW GROUPO.EmployeeConfid77 ENABLE USE IN OPTIMIZATION;
```

♦ 禁止在优化中使用实例化视图 (Sybase Central)

1. 以具有 DBA 权限的用户身份连接到数据库。

2. 在左窗格中，双击 [**视图**]。
3. 右击实例化视图，并选择 [**属性**]。
4. 单击 [**常规**] 选项卡，然后清除 [**在优化中使用**]。
5. 单击 [**确定**]。

♦ 禁止在优化中使用实例化视图 (SQL)

1. 以具有 DBA 权限的用户身份，或以实例化视图所有者身份连接到数据库。
2. 执行带有 DISABLE USE IN OPTIMIZATION 子句的 ALTER MATERIALIZED VIEW 语句。

示例

以下语句创建 EmployeeConfid77 实例化视图，并对其刷新，然后将其禁用以供优化使用。

```
CREATE MATERIALIZED VIEW EmployeeConfid77 AS
   SELECT EmployeeID, Employees.DepartmentID, SocialSecurityNumber, Salary,
ManagerID,
      Departments.DepartmentName, Departments.DepartmentHeadID
   FROM Employees, Departments
   WHERE Employees.DepartmentID=Departments.DepartmentID;
REFRESH MATERIALIZED VIEW EmployeeConfid77;
ALTER MATERIALIZED VIEW EmployeeConfid77 DISABLE USE IN OPTIMIZATION;
```

另请参见

- “ALTER MATERIALIZED VIEW 语句”一节 《SQL Anywhere 服务器 - SQL 参考》

设置优化程序的实例化视图失效程度阈值

当实例化视图所引用的表中的数据发生更改时，实例化视图中的数据就会失效。materialized_view_optimization 数据库选项允许您配置一个失效程度阈值，如果实例化视图超过该阈值，优化程序在处理查询时就不会使用该实例化视图。materialized_view_optimization 数据库选项不影响实例化视图的刷新频率。

如果某个查询显式引用了一个实例化视图，则该视图用于处理查询，而无论视图中数据的更新度如何。还可以使用 SELECT 语句的 OPTION 子句覆盖 materialized_view_optimization 数据库选项的设置，强制使用实例化视图。请参见“SELECT 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

如果您发现优化程序未考虑某个实例化视图，可能是由于该视图数据失效的缘故。调整为负责刷新视图的事件或触发器所指定的时间间隔。

注意
使用快照隔离时，如果实例化视图在事务的快照开始后被刷新，优化程序就会避免使用该视图。

有关如何使用 materialized_view_optimization 数据库选项的信息，请参见“materialized_view_optimization 选项 [数据库]”一节 《SQL Anywhere 服务器 - 数据库管理》。

有关使用事件和触发器的信息，请参见“使用调度和事件自动完成任务” 《SQL Anywhere 服务器 - 数据库管理》。

有关确定优化程序是否考虑实例化视图的信息，请参见“读取执行计划”一节第 569 页和“监控查询性能”一节第 211 页。

隐藏实例化视图

可以对用户隐藏实例化视图的定义。隐藏实例化视图时，将数据库中所存储的视图定义进行模糊处理，使目录中不显示该视图。隐藏后的视图仍可直接引用，也仍可在查询处理中使用。实例化视图隐藏后，使用调试程序进行调试时将会不显示其定义，也无法通过过程分析获得其定义，但视图仍可卸载并重新装载到其它数据库中。

实例化视图的隐藏操作是不可逆的，并且只能使用 SQL 语句执行。

♦ **隐藏实例化视图 (SQL)**

1. 以具有 DBA 权限的用户身份，或以实例化视图所有者身份连接到数据库。
2. 执行带有 SET HIDDEN 子句的 ALTER MATERIALIZED VIEW 语句。

示例

以下语句创建一个实例化视图 EmployeeConfid3，并对其刷新，然后对其视图定义进行模糊处理。

```
CREATE MATERIALIZED VIEW EmployeeConfid3 AS
   SELECT EmployeeID, Employees.DepartmentID, SocialSecurityNumber, Salary,
ManagerID,
      Departments.DepartmentName, Departments.DepartmentHeadID
   FROM Employees, Departments
   WHERE Employees.DepartmentID=Departments.DepartmentID;
REFRESH MATERIALIZED VIEW EmployeeConfid3;
ALTER MATERIALIZED VIEW EmployeeConfid3 SET HIDDEN;
```

小心
使用完该示例后，应删除所创建的实例化视图。否则，在试验其它示例时，将无法对其基础表 Employees 和 Departments 执行模式更改。无法变更具有已启用相关实例化视图的表的模式。请参见“删除实例化视图”一节第 64 页。

另请参见

- “ALTER MATERIALIZED VIEW 语句”一节 《SQL Anywhere 服务器 - SQL 参考》

删除实例化视图

某个实例化视图不再需要时，可将其删除。

实例化视图的删除与视图依赖性

删除实例化视图之前，必须删除或禁用所有相关视图。要确定某个实例化视图是否存在相关视图，请使用 sa_dependent_views 系统过程。请参见“sa_dependent_views 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》。

另请参见“视图依赖性”一节第 34 页。

♦ **删除实例化视图 (Sybase Central)**

1. 以具有 DBA 权限的用户身份或以视图所有者身份连接到数据库。
2. 在左窗格中，双击 [**视图**]。
3. 右击实例化视图，然后选择 [**删除**]。
4. 单击 [**是**]。

♦ **删除实例化视图 (SQL)**

1. 以具有 DBA 权限的用户身份或以视图所有者身份连接到数据库。
2. 执行 DROP MATERIALIZED VIEW 语句。

示例

以下语句创建 EmployeeConfid4 实例化视图，并对其初始化（在其中填充数据），然后将其删除。

```
CREATE MATERIALIZED VIEW EmployeeConfid4 AS
   SELECT EmployeeID, Employees.DepartmentID, SocialSecurityNumber, Salary,
ManagerID,
      Departments.DepartmentName, Departments.DepartmentHeadID
   FROM Employees, Departments
   WHERE Employees.DepartmentID=Departments.DepartmentID;
REFRESH MATERIALIZED VIEW EmployeeConfid4;
DROP MATERIALIZED VIEW EmployeeConfid4;
```

另请参见

- “DROP MATERIALIZED VIEW 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “sa_dependent_views 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》
- “视图依赖性”一节第 34 页

使用索引

在设计和创建数据库时，性能是一个需要重点考虑的因素。索引可以显著提高用于搜索特定行或特定行子集的语句的性能。而另一方面，索引会占用额外的磁盘空间，可能减慢插入、更新和删除操作的速度。

何时使用索引

索引会按表中的一个或多个列将行排序。索引像一个电话号码簿，电话号码簿先按照姓进行排序，然后按照名对相同的姓进行排序。这种排序加快了对特定的姓搜索其电话号码的速度，但它对查找位于特定地址的电话号码未提供任何帮助。同样，数据库索引只对一个或多个特定列的搜索有帮助。

随着表的大小不断增大，索引的用途也越来越大。随着电话号码簿内容的增多，查找某个给定地址的电话号码所需的平均时间也将会增加。但是，与在一个较小的电话号码簿中查找某个人（比方说 K. Kaminski）的电话号码相比，在一个较大的电话号码簿中查找所花费的时间不会长很多。

当存在合适的索引并且使用该索引将提高性能时，数据库服务器查询优化程序将自动使用索引。

创建索引也有一些负面影响。特别是当修改列中的数据时，所有索引都必须与表本身保持一致，从而导致索引可能会影响插入、更新和删除的性能。因此，应该删除不必要的索引。请使用索引顾问来识别不必要的索引。请参见“获取对查询的 [索引顾问] 建议”一节第 167 页。

确定要创建的索引

为数据库选择一组适当的索引是性能优化工作的一个重点。确定一组适当的索引也是一个要求很高的问题。有些索引可能会极大地提高性能，但在存储空间和变更数据时的系统开销方面，索引也会带来相关的开销。

使用索引顾问工具可协助您正确选择索引。它会分析一个查询或一组操作，然后建议您给数据库添加哪些索引。如有未使用的索引，它也会告诉您。请参见“获取对查询的 [索引顾问] 建议”一节第 167 页。

经常被搜索的列的索引

SQL Anywhere 会自动为主键和外键列建立索引。因此，没有必要在键列上手工创建索引，通常也不建议这样做。如果某个列只是键的一部分，则创建索引可能会有帮助。

索引会需要额外的空间，并且可能会略微降低某些用于修改表中数据的语句（例如 INSERT、UPDATE 和 DELETE 语句）的性能。但是，索引可以显著提高搜索性能，因此强烈建议您在频繁执行数据搜索时使用索引。要进一步了解有关索引如何提高性能的信息，请参见“使用索引”一节第 221 页。

只要可能，优化程序便自动使用索引提高数据库语句的性能。此外，当删除、更新或插入行后，索引会自动更新。虽然构建查询时可以使用索引提示显式引用索引，但没必要这样做。

索引提示

构建查询时可提供索引提示。索引提示通过强制使用一个或多个特定索引来覆盖优化程序的查询访问计划选择。索引提示通常只在计算优化程序的计划选择时使用，并且只应由高级用户和数据库管理员使用。不正确地应用索引提示可导致查询性能低下。

使用 FROM 子句的子句指定索引提示。例如，INDEX 子句最多允许指定四个索引。优化程序必须能够使用所有指定的索引，否则会返回错误。

指定 NO INDEX 会禁止查询使用索引，而改为强制顺序扫描表。但是，顺序扫描开销非常大，需要的执行时间也更长。计算优化程序的索引选择时，此子句仅用于比较。

缺省情况下，如果仅使用索引数据（也就是不必访问表中的行）便可以满足查询，则数据库服务器执行仅索引检索。但是，您可能要指定 INDEX ONLY ON，以便索引不再可用于仅索引检索时（例如索引更改或删除）能返回错误。

有关可以在 FROM 子句中指定的索引提示子句的详细信息，请参见“FROM 子句”一节 《SQL Anywhere 服务器 - SQL 参考》。

使用聚簇索引

尽管对于那些搜索指定范围键值的语句而言，使用索引可以极大提高性能，但在索引中相继出现的两行在数据库中却不必出现在同一个表页上。

通过声明聚簇索引可以进一步提高大规模索引扫描的性能。使用聚簇索引会提高相邻索引条目中的两行出现在数据库中同一页上的可能性。这样就减少了表页需要被读到缓冲池中的次数，从而提高了性能。

如果存在具有聚簇特性的索引，数据库服务器在存储表行时会尝试使用与表行在聚簇索引中的顺序大致相同的顺序。但是，尽管数据库服务器会尝试保持键顺序，但聚簇只是大致的，并不能保证完全的聚簇。因此，数据库服务器不能按聚簇索引键顺序来顺序地扫描表并检索所有行。要确保表中的行按排序顺序返回，需要一个访问计划，要么通过索引来访问行，要么执行物理排序。

优化程序可以利用具有聚簇属性的索引，它会修改索引检索的预期开销，通过匹配或相邻索引键值将表行的预期物理相邻性考虑进去。

随着时间的推移，给定表中有越来越多的行被插入或更新，表的聚簇量可能会随之减少。数据库服务器自动跟踪 ISYSPHYSIDX 系统表中各聚簇索引的聚簇量。如果数据库服务器检测到表中的行很大程度上未聚簇，优化程序会调整其预期索引检索开销。

决定使表的一个索引聚簇时需要考虑预期查询负荷。通常需要进行一些实验。通常，当指定查询具备以下条件时，数据库服务器可以使用聚簇索引提高性能：

- 回应查询所需的很多表页还不在内存中。当表页已在内存中时，服务器不需要读取这些页，此类聚簇不相关。
- 可通过执行预计会返回无关紧要的行数的索引检索来回应查询。例如，对于简单主键搜索，聚簇通常不相关。
- 与执行仅索引检索相反，数据库服务器需要实际读取表页。

使用 SQL 语句聚簇索引

可以随时使用 SQL 语句添加或删除索引的聚簇属性。任何主键索引、外键索引、UNIQUE 约束索引或次级索引都可以声明 CLUSTERED 属性。但每个表最多只能声明一个聚簇索引。为此可以使用以下任一语句：

- “CREATE TABLE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “ALTER DATABASE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “CREATE INDEX 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “DECLARE LOCAL TEMPORARY TABLE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》

可以联合使用若干语句来维护和恢复聚簇效果：

- UNLOAD TABLE 语句可用于按照聚族索引键的顺序卸载表。请参见“UNLOAD 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。
- LOAD TABLE 语句可用于按照聚族索引键的顺序在表中插入行。请参见“LOAD TABLE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。
- INSERT 语句尝试按照聚族索引键顺序在包含相邻行的同一表页上放置新行。请参见“INSERT 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。
- REORGANIZE TABLE 语句通过按照聚簇索引重新排列行来恢复表的聚簇。如果在未指定聚族的情况下对表执行 REORGANIZE TABLE 语句，则这些表会按主键重新排序。请参见“REORGANIZE TABLE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

在 Sybase Central 中创建聚簇索引

也可以在 Sybase Central 中使用 [创建索引向导]，并在出现提示时选择 [创建聚簇索引]，来创建聚簇索引。请参见“创建索引”一节第 68 页。

重新排序行以匹配聚簇索引

要重新排序表中的行以匹配聚簇索引，请使用 REORGANIZE TABLE 语句。请参见“REORGANIZE TABLE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

创建索引

索引在指定表的一个或多个列上创建。您可以在基表或临时表上创建索引，但不能在视图上创建索引。要创建单独的索引，请使用 Sybase Central 或 Interactive SQL。可以在索引顾问的指导下为数据库选择适当的索引。

创建索引时，列会按您指定的顺序出现在索引中。索引定义中不允许重复引用列名称。

♦ 创建新索引 (Sybase Central)

1. 以具有 DBA 权限的用户身份连接到数据库。

2. 在左窗格中，双击 [**表**]，并选择要为其创建索引的表。
3. 在右窗格中，单击 [**索引**] 选项卡。
4. 在左窗格中，右击表，并选择 [**新建**] » [**索引**]。
5. 按照 [**创建索引向导**] 中的说明操作。

   新索引随即出现在表的 [**索引**] 选项卡上。它也出现在 [**索引**] 中。

♦ 创建新索引 (SQL)

1. 以具有 DBA 权限的用户身份，或以要创建索引的表的所有者身份连接到数据库。
2. 执行 CREATE INDEX 语句。

除了在表中一个或多个列上创建索引外，还可以使用计算列在内置函数上创建索引。请参见“CREATE INDEX 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

示例

下例使用 Surname 和 GivenName 列在 Employees 表上创建称为 EmployeeNames 的索引：

```
CREATE INDEX EmployeeNames
ON Employees (Surname, GivenName);
```

请参见“CREATE INDEX 语句”一节 《SQL Anywhere 服务器 - SQL 参考》和“提高数据库性能” 第 159 页。

校验索引

可以对索引进行校验，以确保索引中引用的每一行在表中都确实存在。对于外键索引，通过校验检查还会确保相应行存在于主表中。这种检查是对 VALIDATE TABLE 语句所执行的有效性检查的补充。

小心
只有在所有连接都没有更改数据库的时候，才能校验表或整个数据库。

♦ 校验索引 (Sybase Central)

1. 以具有 DBA 权限的用户身份，或以要创建索引的表的所有者身份连接到数据库。
2. 在左窗格中，双击 [**索引**]。
3. 右击索引并选择 [**校验**]。
4. 单击 [**确定**]。

♦ **校验索引 (SQL)**

1. 以具有 DBA 权限的用户身份，或以要创建索引的表的所有者身份连接到数据库。
2. 执行 VALIDATE INDEX 语句。

♦ **校验索引（dbvalid 实用程序）**

● 运行 dbvalid 命令时指定 -i 选项。

示例 1

校验名为 EmployeeNames 的索引。如果提供的是表名而不是索引名，则将校验主键索引。

```
VALIDATE INDEX EmployeeNames;
```

请参见“VALIDATE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

示例 2

校验名为 EmployeeNames 的索引。-I 选项指定每个给定的对象名称是一个索引。

```
dbvalid -I EmployeeNames
```

请参见“校验实用程序 (dbvalid)”一节 《SQL Anywhere 服务器 - 数据库管理》。

重建索引

有时需要重建索引，因为由于对表执行大量的插入和删除操作，索引会形成碎片或有偏差。重建索引时，重建的是物理索引。重建操作会为使用物理索引的所有逻辑索引带来好处。不必对逻辑索引执行重建。请参见“使用逻辑索引共享索引”一节第 596 页。

可以在 Sybase Central 中或通过执行 ALTER INDEX ...REBUILD 语句来重建索引。还可以通过使用 REORGANIZE TABLE 语句重建索引来删除表碎片。本节介绍如何使用 Sybase Central 和 ALTER INDEX ...REBUILD 语句来重建索引。有关使用 REORGANIZE TABLE 语句的详细信息，请参见“REORGANIZE TABLE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

♦ **重建索引 (Sybase Central)**

1. 以具有 DBA 权限的用户身份，或以要创建索引的表的所有者身份连接到数据库。
2. 在左窗格中，双击 [**索引**]。
3. 右击索引并选择 [**重建**]。
4. 单击 [**确定**]。

♦ **重建索引 (SQL)**

1. 以具有 DBA 权限的用户身份，或以与索引相关联的表的所有者身份连接到数据库。
2. 执行 ALTER INDEX ...REBUILD 语句。REBUILD 语句。

示例

以下语句会重建 Customers 表的 IX_customer_name 索引：

```
ALTER INDEX IX_customer_name ON Customers REBUILD;
```

有关 ALTER INDEX 语句的语法的详细信息，请参见“ALTER INDEX 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

另请参见

有关索引碎片和偏差以及如何减少它们的详细信息，请参见“减少索引碎片和分布偏差”一节 第 216 页。

有关如何检测索引碎片和偏差的详细信息，请参见“应用程序分析向导”一节第 161 页 和“sa_index_density 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》。

删除索引

如果不再需要某个索引，可以在 Sybase Central 或 Interactive SQL 中将其从数据库中删除。

- **删除索引 (Sybase Central)**

1. 以具有 DBA 权限的用户身份，或以要创建索引的表的所有者身份连接到数据库。
2. 在左窗格中，双击 [**索引**]。
3. 右击索引，然后选择 [**删除**]。
4. 单击 [**是**]。

- **删除索引 (SQL)**

1. 以具有 DBA 权限的用户身份，或以与索引相关联的表的所有者身份连接到数据库。
2. 执行 DROP INDEX 语句。

示例

以下语句会从数据库中删除 EmployeeNames 索引：

```
DROP INDEX EmployeeNames;
```

请参见“DROP INDEX 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

目录中的索引信息

ISYSIDX 系统表提供了数据库中所有索引（包括主键索引和外键索引）的列表。有关索引的附加信息可在 ISYSPHYSIDX、ISYSIDXCOL 和 ISYSFKEY 系统视图中找到。可以使用 Sybase Central 或 Interactive SQL 浏览这些表的视图，以查看其中包含的数据。

下面简要概括了索引信息在系统表中的存储方式：

- **ISYSIDX 系统表** 用于跟踪索引的中心表，ISYSIDX 系统表中的每一行定义数据库中的一个逻辑索引（PKEY、FKEY、UNIQUE 约束、次级索引）。请参见“SYSIDX 系统视图”一节《SQL Anywhere 服务器 - SQL 参考》和“使用逻辑索引共享索引”一节第 596 页。
- **ISYSPHYSIDX 系统表** ISYSPHYSIDX 系统表中的每一行定义数据库中的一个物理索引。请参见“SYSPHYSIDX 系统视图”一节 《SQL Anywhere 服务器 - SQL 参考》和“使用逻辑索引共享索引”一节第 596 页。
- **ISYSIDXCOL 系统表** 就像 SYSIDX 系统视图中的每一行描述数据库中一个索引一样，SYSIDXCOL 系统视图中的每一行描述 SYSIDX 系统视图中所描述的一个索引的一列。请参见“SYSIDXCOL 系统视图”一节 《SQL Anywhere 服务器 - SQL 参考》。
- **ISYSFKEY 系统表** 数据库中的每个外键由 ISYSFKEY 系统表中的一行和 ISYSIDX 系统表中的一行来定义。请参见“SYSFKEY 系统视图”一节 《SQL Anywhere 服务器 - SQL 参考》。

确保数据完整性

目录

如果数据具有完整性，则表示数据有效—正确并且精确—而且数据库的关系结构是完整的。参照完整性约束可以加强数据库的关系结构。这些规则使数据在各个表之间保持一致。在数据库中构建完整性约束是确保数据保持一致的最好方法。

可以实施多种类型的完整性约束。例如，您可以通过对表和列施加约束和 CHECK 约束来确保各个条目正确。您还可以通过选择合适的数据类型或设置特殊缺省值来配置列属性。

SQL Anywhere 支持存储过程，这样可以对数据输入数据库的方式进行具体的控制。您还可以创建触发器或在发生特定操作（例如，更新特定的列）时会自动调用的自定义存储过程。

有关过程和触发器的详细信息，请参见“使用过程、触发器和批处理”第 777 页。

哪些情况会导致数据无效

如果不进行适当的检查，可能会导致数据库中的数据无效。可以使用本章中介绍的功能防止这些情况发生。

信息不正确

- 操作员键入的交易日期不正确。
- 由于操作员在输入数据时丢失了一位数字，导致雇员的工资变为实际工资的十分之一。

数据重复

- 两名不同雇员在公司数据库的 Departments 表中添加相同的新部门（DepartmentID 为 200）。

外键关系失效

- 由 DepartmentID 300 标识的部门关闭，并且由于疏忽而导致一条雇员记录未分配到新部门。

在数据库中构建完整性约束

为确保数据库中数据的有效性，需要创建检查语句以定义有效数据和无效数据，并设计数据必须遵守的规则（亦称作业务规则）。检查语句和规则一起构成**约束**。

构建数据库本身的约束比构建客户端应用程序的约束或将约束形成文字说明提供给数据库用户更为可靠。构建数据库的约束将成为数据库本身定义的一部分，并且数据库将在所有应用程序中一致地实施这些约束。只需在数据库中设置一次约束，对以后与数据库进行的所有交互都将实施该约束。

相反，构建客户端应用程序的约束在每次软件更改时都会受到影响，而且可能需要在多个应用程序，或一个客户端应用程序的多个位置中实施。

哪些情况会更改数据库的内容

从客户端应用程序提交 SQL 语句时，会更改数据库表中的信息。实际上，只有几个 SQL 语句会修改数据库中的信息。您可以：

- 使用 UPDATE 语句可以更新表中某一行的信息。
- 使用 DELETE 语句可以删除现有行。
- 使用 INSERT 语句可以在表中插入新行。

用于保持数据完整性的工具

要保持数据完整性，您可以使用缺省值、数据约束和保持数据库参照结构的约束。

缺省值

可以为列指派缺省值以确保某类数据条目更可靠。例如：

- 可以使用当前日期作为某一列的缺省值，来记录任何用户或客户端应用程序操作的事务日期。
- 使用其它类型的缺省值还可以使列值在输入一行新数据时自动递增，无需执行任何其它特定用户操作。利用此功能，可以确保项目（例如，订购单）具有唯一的连续编号。

有关这些列缺省值和其它列缺省值的详细信息，请参见“使用列缺省值”一节第 79 页。

约束

可以对各列或各表中的数据应用多种类型的约束。例如：

- NOT NULL 约束可以防止列包含 NULL 条目。
- 指派给某列的 CHECK 约束可以确保该列中的每一项都满足特定的条件。例如，可以确保 Salary 列的条目在指定的范围内，从而避免在输入新值时出现用户错误。
- 可以针对不同列中的相对值设置 CHECK 约束。例如，您可以确保图书馆数据库中的 [返还日期] 条目晚于 [借阅日期] 条目。
- 触发器可以实施更复杂的 CHECK 条件。请参见“使用过程、触发器和批处理”第 777 页。

列约束还可以从域继承。有关这些约束以及其它表约束和列约束的详细信息，请参见“使用表和列约束”一节第 85 页。

实体和参照完整性

由主键和外键定义的关系将关系数据库表中的信息相关联。必须将这些关系构建数据库设计。以下完整性规则用于保持数据库的结构：

- **实体完整性** 跟踪主键。确保给定表中的每一行都由一个非 NULL 主键唯一地标识。
- **参照完整性** 跟踪定义表间关系的外键。确保所有外键值与相应主键中的值匹配，或包含 NULL 值（如果定义为允许 NULL 值）。

有关实施参照完整性的详细信息，请参见“实施实体完整性和参照完整性”一节第 92 页。有关设计合适的主键和外键关系的详细信息，请参见“在 SQL Anywhere 中创建数据库”第 3 页。

使用触发器实施高级完整性规则

也可以使用触发器来保持数据完整性。**触发器**是指一个存储在数据库中的过程，只要指定的表中的信息发生更改，就会自动执行此过程。触发器是一个强大的机制，数据库管理员和开发人员可以用它来确保数据的可靠性。

有关触发器的详细信息，请参见“使用过程、触发器和批处理”第 777 页。

用于实施完整性约束的 SQL 语句

以下 SQL 语句用于实施完整性约束：

- **CREATE TABLE 语句** 此语句可在创建表的过程中实现完整性约束。
- **ALTER TABLE 语句** 此语句可将完整性约束添加到现有表，或修改现有表的约束。
- **CREATE TRIGGER 语句** 使用这个语句可以创建触发器，实施更复杂的业务规则。
- **CREATE DOMAIN 语句** 此语句创建用户定义的数据类型。数据类型的定义可以包括约束。

有关这些语句的语法的详细信息，请参见“SQL 语句”《SQL Anywhere 服务器 - SQL 参考》。

使用列缺省值

只要在数据库表中输入一行新数据，列缺省值就会自动为特定列指派一个指定的值。指派的缺省值不需要客户端应用程序执行任何操作，但是如果客户端应用程序确实为列指定了值，新值将替换列的缺省值。

使用列缺省值可以自动在列中快速填入信息，例如插入某一行的日期或时间，或输入信息者的用户 ID。使用列缺省值有助于保持数据完整性，但不能实施数据完整性。客户端应用程序总是可以替换缺省值。

支持的缺省值

SQL 支持以下缺省值：

- CREATE TABLE 语句或 ALTER TABLE 语句中指定的字符串
- CREATE TABLE 语句或 ALTER TABLE 语句中指定的数字
- AUTOINCREMENT：自动递增的数字：每增加一行，新行中该列的值将在原有最大值的基础上加一
- 缺省的 GLOBAL AUTOINCREMENT，确保多个数据库间的主键的唯一性。
- 使用 NEWID 函数生成的通用唯一标识符 (UUID)。
- 当前日期、时间或时间戳
- 当前数据库用户的用户 ID
- NULL 值
- 不参照任何数据库对象的常数表达式

创建列缺省值

可以使用 CREATE TABLE 语句在创建表时创建列缺省值，或使用 ALTER TABLE 语句在创建了表之后添加列缺省值。

示例

以下语句将缺省值添加到 SalesOrders 表中名为 ID 的现有列中，使其自动递增（除非客户端应用程序指定一个值）。请注意，在 SQL Anywhere 示例数据库中，此列已设置为 AUTOINCREMENT。

```
ALTER TABLE SalesOrders
ALTER ID DEFAULT AUTOINCREMENT;
```

有关详细信息，请参见“ALTER TABLE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》和“CREATE TABLE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

变更和删除列缺省值

可以使用 ALTER TABLE 语句更改或删除列缺省值，方式与用此语句创建缺省值时相同。以下语句将名为 OrderDate 的列的缺省值从其当前设置更改为 CURRENT DATE：

```
ALTER TABLE SalesOrders
ALTER OrderDate DEFAULT CURRENT DATE;
```

要删除列缺省值，可将其修改为 NULL。以下语句将删除 OrderDate 列中的缺省值：

```
ALTER TABLE SalesOrders
ALTER OrderDate DEFAULT NULL;
```

在 Sybase Central 中管理列缺省值

在 Sybase Central 中，可以使用 [**列属性**] 窗口中的 [**值**] 选项卡来添加、变更和删除列缺省值。

♦ **显示列的 [属性] 窗口**

1. 以具有 DBA 特权的用户身份连接到数据库。
2. 在左窗格中，双击 [**表**]。
3. 单击表。
4. 单击 [**列**] 选项卡。
5. 右击列并选择 [**属性**]。

当前日期和时间缺省值

对于数据类型为 DATE、TIME 或 TIMESTAMP 的列，可以使用当前日期、当前时间或当前时间戳作为缺省值。您选择的缺省值必须与此列的数据类型兼容。

当前日期缺省值的有用示例

使用当前日期作为缺省值可以：

- 在联系人数据库中记录通话日期
- 在销售记录数据库中记录订单日期
- 在图书馆数据库中记录借阅人的借书日期

当前时间戳

当前时间戳与当前日期缺省值相似，但精度更高。例如，联系管理应用程序的用户在一天中可能要与同一个客户联系多次：当前时间戳缺省值在区分这些联系时将发挥作用。

由于时间戳记录的日期和时间可以精确到百万分之一秒，因此，当事件的顺序在数据库中非常重要时，可能需要使用当前时间戳。

缺省时间戳

缺省时间戳提供了一种指示表中各行的上次修改时间的方法。当用 DEFAULT TIMESTAMP 声明列时，会提供一个缺省的插入值，每当更新行时，该值都用当前日期和时间更新。要提供插入时的缺省值，而在每次更新时不改变，请使用 DEFAULT CURRENT TIMESTAMP 而不是 DEFAULT TIMESTAMP。请参见“CREATE TABLE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》中的 DEFAULT 子句。

有关时间戳、时间和日期的详细信息，请参见“SQL 数据类型” 《SQL Anywhere 服务器 - SQL 参考》。

用户 ID 缺省值

将 DEFAULT USER 指派给列可作为一种简单又可靠的方法来标识在数据库中输入信息的人。例如，若销售人员从事按业绩提成的工作，可能需要此信息。

为表的主键生成用户 ID 缺省值对偶尔连接的用户来说非常有用，并有助于避免在更新信息时发生冲突。这些用户可以将与他们的工作相关的表复制到便携式计算机上，在未连接到多用户数据库的情况下进行更改，等到他们回来后，再将事务日志应用于服务器。

LAST USER 特殊值指定上次修改行的用户的名称。与 DEFAULT TIMESTAMP 组合时，LAST USER 缺省值可用于记录（在单独的列中）上次更改行的用户和日期及时间。请参见“LAST USER 特殊值”一节 《SQL Anywhere 服务器 - SQL 参考》。

自动增量缺省值

AUTOINCREMENT 缺省值适用于那些数字值本身并没有含义的数字数据字段。此功能可为每个新行指派大于列中任何其它值的唯一值。可以使用 AUTOINCREMENT 列记录订购单编号、标识客户服务呼叫或其它需要标识号的条目。

自动增量列通常是主键列，或限制为包含唯一值的列（请参见“实施实体完整性”一节 第 92 页）。

您可以使用 @@identity 全局变量来检索插入到自动增量列中的最新值。有关详细信息，请参见“@@identity 全局变量”一节 《SQL Anywhere 服务器 - SQL 参考》。

自动增量和负数

自动增量设计用于正整数。

创建表时，自动增量的初始值设置为 0。执行完在该列中显式插入负值的插入操作后，初始值仍保持为分配给该列的最大值。如果插入操作未提供任何值，AUTOINCREMENT 将生成值 1，强制生成的所有其它值都为正数。

自动增量和 IDENTITY 列

具有自动增量缺省值的列在 Transact-SQL 应用程序中称为 IDENTITY 列。

有关 IDENTITY 列的信息，请参见“特殊 IDENTITY 列”一节第 624 页。

另请参见

- “重装带有自动增量列的表”一节 《SQL Anywhere 11 - 更改和升级》

GLOBAL AUTOINCREMENT 缺省值

GLOBAL AUTOINCREMENT 缺省值专用于在 SQL Remote 复制或 MobiLink 同步环境中使用多个数据库的情况。它可确保多个数据库间的主键的唯一性。

此选项类似于 AUTOINCREMENT，只不过要对域进行分区。每个分区都包含相同数目的值。为每个数据库副本指定一个唯一全局数据库标识号。SQL Anywhere 只从用数据库编号唯一标识的分区中提供数据库中的缺省值。

分区大小可以为任意正整数，但分区大小的选择通常要保证任何一个分区内的编号资源尽量不被用尽（如果曾有过这种情况）。

对于 BIGINT 或 UNSIGNED BIGINT 类型的列，缺省分区大小是 $2^{32} = 4294967296$；对于所有其它类型的列，缺省分区大小是 $2^{16} = 65536$。由于这些缺省值可能不合适（尤其当列不是 INT 或 BIGINT 类型时），因此最好显式指定分区大小。

使用此选项时，每个数据库中的公共选项 global_database_id 的值必须设置为唯一的非负整数。该值唯一标识数据库，并指示从哪个分区分配缺省值。允许的值范围是 $np + 1$ 到 $(n + 1)p$，其中 n 是公共选项 global_database_id 的值，p 是分区大小。例如，如果将分区大小定义为 1000 并将 global_database_id 设置为 3，则该范围将是 3001 到 4000。

如果上一个值小于 $(n + 1)p$，则下一个缺省值将比列中的上一个最大值大一。如果列不包含任何值，则第一个缺省值为 $np + 1$。缺省列值不受当前分区之外的列值的影响，即不受小于 $np + 1$ 或大于 $p(n + 1)$ 的数的影响。如果通过 MobiLink 同步从另一个数据库中复制了这些值，则这些值就可能存在。

由于不能将公共选项 global_database_id 设置为负值，因此所选值始终是正数。最大标识号仅受列数据类型和分区大小的限制。

如果将公共选项 global_database_id 设置为缺省值 2147483647，则在列中插入 NULL 值。如果不允许使用 NULL 值，则尝试插入行时将出错。例如，如果列包含在表的主键中，便会发生这种情况。

当分区内的可用值用完时，也会生成 NULL 缺省值。在这种情况下，应为数据库指派一个新的 global_database_id 值，以便可以从另一个分区中选择缺省值。如果列不允许使用 NULL 值，则尝试插入 NULL 值时将出错。若要检测提供的未用值是否过小并处理此情况，请创建一个 GlobalAutoincrement 类型的事件。请参见“了解事件”一节 《SQL Anywhere 服务器 - 数据库管理》。

全局自动增量列通常是主键列，或限制为包含唯一值的列（请参见“实施实体完整性”一节 第 92 页）。

虽然在其它情况下可以使用全局自动增量缺省值，但这样做可能反而导致数据库性能降低。例如，在各列的下一个值存储为 64 位有符号整数的情况下，使用大于 $2^{31} - 1$ 的值或较大的双精度值或数字值可能会导致返转为负值。

您可以使用 @@identity 全局变量来检索插入到自动增量列中的最新值。有关详细信息，请参见“@@identity 全局变量”一节 《SQL Anywhere 服务器 - SQL 参考》。

另请参见

- “使用全局自动增量”一节 《MobiLink - 服务器管理》
- “全局自动增量列”一节 《SQL Remote》
- “CREATE TABLE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “重装带有自动增量列的表”一节 《SQL Anywhere 11 - 更改和升级》

NEWID 缺省值

通用唯一标识符（Universally Unique Identifier，简称 UUID），也称为全局唯一标识符（Globally Unique Identifier，简称 GUID）可用于标识在表中唯一的行。在一台计算机上所生成的值与在其它计算机上生成的值不相同。因此，可将它们用作复制和同步环境中的键。

相比较而言，将 UUID 值与 GLOBAL AUTOINCREMENT 值用作主键各有优缺点。例如：

- UUID 比 GLOBAL AUTOINCREMENT 更易于设置，因为无需为每个远程数据库都分配一个唯一的数据库 ID。也无需考虑系统中数据库的数目或单个表中的行数。可以使用抽取实用程序 (dbxtract) 来进行数据库 ID 的指派。如果使用 BIGINT 数据类型，使用 GLOBAL AUTOINCREMENT 时通常不用考虑这个问题，但使用较小的数据类型时需要考虑这一点。
- UUID 值比 GLOBAL AUTOINCREMENT 所需的值大得多，并且要求主表和外表有更多的表空间。使用 UUID 时，这些列中的索引的效率较低。简言之，GLOBAL AUTOINCREMENT 的表现可能更好些。
- UUID 不具有隐式排序。例如，如果 A 和 B 是 UUID 值，A > B 并不意味着 A 在 B 之后生成（即使 A 和 B 在同一台计算机上生成）。如果您需要此行为，可能需要额外的列和索引。

另请参见

- “NEWID 函数 [Miscellaneous]”一节 《SQL Anywhere 服务器 - SQL 参考》
- “UNIQUEIDENTIFIER 数据类型”一节 《SQL Anywhere 服务器 - SQL 参考》

NULL 缺省值

对于允许使用 NULL 值的列，指定 NULL 缺省值与不指定任何缺省值完全相同。如果插入行的客户端未显式指派值，此行会自动获得一个 NULL 值。

某些列的信息可选或并非始终可用时，可以使用 NULL 缺省值。

有关 NULL 值的详细信息，请参见“NULL 值”一节 《SQL Anywhere 服务器 - SQL 参考》。

字符串和数字缺省值

只要某一列的数据类型为字符串或数字，就可以指定一个特定字符串或数字作为缺省值。必须确保指定的缺省值可以转换为该列的数据类型。

缺省字符串和数字适用于具有典型条目的指定列。例如，如果公司有两个办事处，总部位于 city_1，而较小的办事处位于 city_2，则可能需要将地点列的缺省条目设置为 city_1，以便更容易地输入数据。

常数表达式缺省值

可以使用未引用任何数据库对象的常数表达式作为缺省值。使用常数表达式可以设置像 *自今天起第十五天的日期* 这样的列缺省值。要设置上述缺省值，应输入

```
... DEFAULT ( DATEADD( day, 15, GETDATE() ) );
```


使用表和列约束

除基本的表结构（列的数量、名称和数据类型，该表的名称和位置）外，CREATE TABLE 语句和 ALTER TABLE 语句还可以指定许多不同的表属性，这些属性可用于控制数据完整性。约束允许您对可显示在某个列中的值或不同列中的值之间的关系施加限制。约束可以是表范围内的约束，也可以应用于各个列。

本节介绍如何使用约束来帮助确保表中数据的正确性。

对列使用 CHECK 约束

使用 CHECK 条件可以确保列中的值满足某个条件或规则。这些规则或条件可能需要用来验证数据是否正确，或者可能需要使用更严格的规则来反映公司的政策和程序。如果某列的有效值限制在一定范围内，则可以单独在该列上使用 CHECK 条件。

CHECK 条件就位后，修改行之前将依据此条件来评估将来值。更新具有检查约束的值时，将检查该值的约束以及该行其它值的约束。

也可以在域中附加 CHECK 约束。请参见“从域继承列 CHECK 约束”一节第 86 页。

注意

如果条件返回 FALSE 值，则列 CHECK 测试失败。如果条件返回 UNKNOWN 值（此行为同返回 TRUE 值一样），则允许使用此值。

有关有效条件的详细信息，请参见“搜索条件”一节 《SQL Anywhere 服务器 - SQL 参考》。

示例 1

可以实施特定的格式要求。例如，如果表中有一列用于存储电话号码，您可能希望确保用户以相同的格式输入电话号码。对于北美洲电话号码，可以使用类似以下约束：

```
ALTER TABLE Customers
ALTER Phone
CHECK ( Phone LIKE '(___) ___-____' );
```

CHECK 条件就位后，如果您尝试将 Phone 值设置为 9835，将不允许进行此更改。

示例 2

可以确保输入的值与一组有限数量的值中的一个相匹配。例如，要确保 City 列只包含几个允许城市（例如，公司设有办事处的那些城市）之一，可以使用以下约束：

```
ALTER TABLE Customers
ALTER City
CHECK ( City IN ( 'city_1', 'city_2', 'city_3' ) );
```

缺省情况下，比较字符串时不区分大小写，除非创建数据库时显式指定需要区分大小写。

示例 3

可以确保日期或数字在特定的范围内。例如，您可能需要使用以下约束确保雇员的 StartDate 列在公司成立日期和当前日期之间：

```
ALTER TABLE Employees
ALTER StartDate
CHECK ( StartDate BETWEEN '1983/06/27'
                  AND CURRENT DATE );
```

有多种日期格式可以使用。不管当前选项如何设置，数据库始终都可以识别本例中使用的 YYYY/MM/DD 格式。

对表使用 CHECK 约束

作为约束应用于表上的 CHECK 条件通常确保正在添加或修改的行中的两个值之间具有正确的关系。

为约束指定名称时，约束单独保存在系统表中，您可以单独替换或删除它们。由于这种情况比较灵活，因此建议您尽可能命名 CHECK 约束或者使用单个的列约束。

例如，可以在 Employees 表上添加一个约束以确保 TerminationDate 始终晚于或等于 StartDate：

```
ALTER TABLE Employees
   ADD CONSTRAINT valid_term_date
   CHECK( TerminationDate >= StartDate );
```

有关详细信息，请参见“ALTER TABLE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

从域继承列 CHECK 约束

您可以在域中附加 CHECK 约束。使用这些域定义的列将继承 CHECK 约束。为列显式指定的 CHECK 约束将替换域中的 CHECK 约束。例如，此域定义中的 CHECK 子句要求插入到列中的值只能为正整数。

```
CREATE DATATYPE posint INT
CHECK ( @col > 0 );
```

任何使用 posint 域定义的列都只接受正整数，除非为列本身显式指定了 CHECK 约束。由于任何前缀为 @ 符号的变量都会在评估 CHECK 约束时被列名替换，因此，可以使用任何前缀为 @ 的变量名称来代替 @col。

带有 DELETE CHECK 子句的 ALTER TABLE 语句将从表定义中删除所有 CHECK 约束，包括从域继承的那些 CHECK 约束。

在域上定义列后，对此域定义中的约束所做的任何更改都不会应用到该列。创建列时，它会从该域获取约束，但之后两者之间就不再有关联。

另请参见

- “域”一节 《SQL Anywhere 服务器 - SQL 参考》
- “对列使用 CHECK 约束”一节第 85 页

管理约束

在 Sybase Central 中，可以在表或 [**列属性**] 窗口的 [**约束**] 选项卡上添加、变更和删除列约束。

♦ **管理约束 (Sybase Central)**

1. 以具有 DBA 特权的用户身份连接到数据库。
2. 在左窗格中，双击 [**表**]。
3. 单击要变更的表。
4. 在右窗格中，单击 [**约束**] 选项卡，然后修改现有约束或添加新约束。

管理 UNIQUE 约束

对于列，UNIQUE 约束可指定列中的值必须唯一。对于表，UNIQUE 约束可标识一个或多个用于标识表中唯一行的列。表中任何两行的值在所有指定的列中不能相同。一个表可以有一个以上的 UNIQUE 约束。

♦ **管理唯一约束 (Sybase Central)**

1. 以具有 DBA 特权的用户身份连接到数据库。
2. 在左窗格中，双击 [**表**]。
3. 单击要变更的表。
4. 在右窗格中，单击 [**约束**] 选项卡。
5. 右击 [**约束**] 选项卡，然后选择 [**新建**] » [**唯一约束**]。
6. 请按照 [**创建唯一约束向导**] 中的说明完成操作。

变更和删除 CHECK 约束

变更表可能会影响到数据库的其他用户。虽然可以在其它连接处于活动状态时执行 ALTER TABLE 语句，但如果任何其它连接正在使用要变更的表，则不能执行 ALTER TABLE 语句。对于大型表，ALTER TABLE 操作非常耗时。在处理该语句的过程中，所有其它引用正被变更的表的请求都将被禁止。请参见“ALTER TABLE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

有几种方法可以改变表中现有的 CHECK 约束集。

- 您可以给表或某个列添加新 CHECK 约束。
- 通过将列的 CHECK 约束设置为 NULL，可以删除该约束。例如，以下语句将删除 Customers 表中 Phone 列的 CHECK 约束：

```
ALTER TABLE Customers
ALTER Phone CHECK NULL;
```

- 您可以用与添加 CHECK 约束相同的方法来替换列的 CHECK 约束。例如，以下语句将添加或替换 Customers 表中 Phone 列的 CHECK 约束：

```
ALTER TABLE Customers
ALTER Phone
CHECK ( Phone LIKE '___-___-____' );
```

- 您可以变更在表中定义的 CHECK 约束：
  - 您可以使用带有 ADD *table-constraint* 子句的 ALTER TABLE 添加新的 CHECK 约束。
  - 如果已经定义了约束名，则可以变更各个约束。
  - 如果尚未定义约束名，则可以使用 ALTER TABLE DELETE CHECK 删除所有现有的 CHECK 约束（包括列 CHECK 约束和从域继承的 CHECK 约束），然后添加新的 CHECK 约束。

    使用带有 DELETE CHECK 子句的 ALTER TABLE 语句：

```
ALTER TABLE table-name
DELETE CHECK;
```

Sybase Central 允许您添加、变更和删除表 CHECK 约束和列 CHECK 约束。有关详细信息，请参见“管理约束”一节第 87 页。

从表中删除列并不会删除表约束中保存的与该列相关联的 CHECK 约束。如果不删除这些约束，尝试在该表中插入数据时（甚至在只查询数据时）会出现以下错误消息：[**未找到列**]。

注意
如果返回 FALSE 值，则表 CHECK 约束失败。如果条件返回 UNKNOWN 值（此行为同返回 TRUE 值一样），则允许使用此值。

使用域

域为用户定义的数据类型，与其它属性一起使用时，可以限制可接受值的范围或提供缺省值。域继承自内部数据类型之一。通常情况下，可允许值的范围由一个检查约束限制。此外，域可以指定缺省值，并确定其是否使用 NULL 值。

定义自己的域具有许多优点，包括：

- 避免因输入不适当的值而产生的常见错误。在域中设置的约束可以确保所有用于在某范围内或以某种格式保存值的列和变量只能保存所需的值。例如，使用一种数据类型可以确保只能在数据库中键入位数正确的信用卡号。
- 使应用程序和数据库结构更便于理解。
- 便利性。例如，您可能希望所有表标识符都是正整数，并且在缺省情况下自动递增。您可以通过在每次定义新表时输入适当的约束和缺省值来实施此限制，但如果定义一个新域，然后简单地声明标识符只能使用属于指定域的值，则更为简便。

有关域的详细信息，请参见“SQL 数据类型”《SQL Anywhere 服务器 - SQL 参考》。

创建域 (Sybase Central)

可以使用 Sybase Central 创建域，或将域指派到一列。

♦ 创建新域 (Sybase Central)

1. 以具有 DBA 特权的用户身份连接到数据库。
2. 在左窗格中，右击 [**域**] 并选择 [**新建**] » [**域**]。
3. 请按照 [**创建域向导**] 中的说明进行操作。

♦ 将域指派到列 (Sybase Central)

1. 以具有 DBA 特权的用户身份连接到数据库。
2. 在左窗格中，双击 [**表**]。
3. 单击表。
4. 在右窗格中，单击 [**列**] 选项卡。
5. 选择列，然后在 [**数据类型**] 字段中单击省略号（三个句点）按钮。
6. 单击 [**数据类型**] 选项卡，然后选择 [**域**]。
7. 在 [**域**] 列表中，选择一个域。
8. 单击 [**确定**]。

创建域 (SQL)

可以使用 CREATE DOMAIN 语句创建和定义域。请参见“CREATE DOMAIN 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

SQL Anywhere 附带了一些预定义的域。例如，货币域 MONEY。

♦ 创建新域 (SQL)

1. 连接到数据库。
2. 执行 CREATE DOMAIN 语句。

示例 1：简单的域

数据库中的一些列用于存储雇员姓名，其它列用于存储地址。那么，您可以定义以下域。

```
CREATE DOMAIN persons_name CHAR(30)
CREATE DOMAIN street_address CHAR(35);
```

定义了这些域后，则可以按照类似使用内置数据类型的方式使用它们。例如，可以用这些定义来定义表，如下所示。

```
CREATE TABLE Customers (
   ID INT  DEFAULT AUTOINCREMENT  PRIMARY KEY,
   Name persons_name,
   Street street_address);
```

示例 2：缺省值、检查约束和标识符

在上例中，表的主键指定为整数类型。实际上，许多表都需要相似的标识符。创建一个标识符域供这些应用程序使用，要比将主键指定为整数更为方便。

创建域时，可以指定一个缺省值，并提供检查约束，以确保不能将不适当的值键入任何属于这种类型的列。

通常使用整数值作为表标识符。正整数非常适合于用作唯一标识符。由于可能在许多表中使用这样的标识符，因此可以定义以下域。

```
CREATE DOMAIN identifier UNSIGNED INT
DEFAULT AUTOINCREMENT;
```

可以使用此定义重写 Customers 表的定义，如上所示。

```
CREATE TABLE Customers2 (
   ID identifier PRIMARY KEY,
   Name persons_name,
   Street street_address
);
```

删除域

可以使用 Sybase Central 或 DROP DOMAIN 语句来删除域。

只有拥有 DBA 特权的用户或创建域的用户可以删除域。此外，如果数据库中的任何变量或列使用域，则不能删除该域，因此，首先需要删除所有属于该类型的列或变量，才能删除该域。

- **删除域 (Sybase Central)**

1. 以具有 DBA 特权的用户身份连接到数据库。
2. 在左窗格中，双击 [**域**]。
3. 在右窗格中，右击域然后选择 [**删除**]。
4. 单击 [**是**]。

- **删除域 (SQL)**

1. 以具有 DBA 特权的用户身份连接到数据库。
2. 执行 DROP DOMAIN 语句。

示例

以下语句将删除 persons_name 域。

```
DROP DOMAIN persons_name;
```

有关详细信息，请参见“DROP DOMAIN 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

实施实体完整性和参照完整性

数据库的关系结构使得数据库服务器可以识别数据库中的信息，并确保各表中的所有行都能保持表之间的关系（在数据库模式中描述）。

实施实体完整性

用户插入或更新行后，数据库服务器将确保表的主键仍有效：表中的每一行都由主键唯一标识。

示例 1

SQL Anywhere 示例数据库中的 Employees 表使用雇员 ID 作为主键。在表中添加新雇员后，数据库服务器将检查新雇员 ID 值是否唯一，以及是否非 NULL。

示例 2

SQL Anywhere 示例数据库中的 SalesOrderItems 表使用两个列来定义主键。

此表保存有关订购项目的信息。一列包含指定订单的 ID，但每个订单可能涉及几个项目，因此此列本身不能作为主键。另一 LineID 列标识与项目对应的行。将 ID 和 LineID 列结合起来可以唯一地指定项目，并形成主键。

如果客户端应用程序破坏了实体完整性

实体完整性要求主键的每个值在表中都是唯一的，并且不能有 NULL 值。如果客户端应用程序尝试插入或更新一个主键值，但提供的值不唯一，则会破坏实体完整性。如果实体完整性被破坏，将导致无法将新信息添加到数据库中，而会向客户端应用程序发送一条错误消息。

应用程序编程人员应确定如何将有关完整性被破坏的信息传达给用户，以便用户采取适当的措施。适当的措施通常很简单，只需要求用户为主键提供另外一个唯一的值。

主键实施实体完整性

为每个表指定了主键后，客户端应用程序开发人员或数据库管理员不需要采取任何其它措施即可保持数据完整性。

表的所有者在创建表时定义了表的主键。如果他们以后要修改表的结构，也可以重新定义主键。

有关创建主键的详细信息，请参见“管理主键”一节第 22 页。

有关 CREATE TABLE 语句的详细语法，请参见“CREATE TABLE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

有关更改表结构的信息，请参见“ALTER TABLE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

实施参照完整性

外键（由一个特定列或列组合构成）将一个表（**外**表）中的信息与另一个表（**参照**表或**主**表）中的信息进行关联。要使外键关系有效，外键中的条目必须与参照表中一行的主键值相对应。有时，可能会参照其它某个唯一的列组合，而不参照主键。

示例 1

SQL Anywhere 示例数据库包含一个 Employees 表和一个 Departments 表。Employees 表的主键是雇员 ID，而 Departments 表的主键是部门 ID。在 Employees 表中，部门 ID 称作 Departments 表的**外键**，因为 Employees 表中的每个部门 ID 都严格对应于 Departments 表中的部门 ID。

外键关系是多对一关系。Employees 表中的几个条目都具有相同的部门 ID 条目，但部门 ID 是 Departments 表的主键，因此在 Departments 表中是唯一的。如果外键引用 Departments 表中包含重复条目或 NULL 值条目的列，则无法获知 Departments 表中的哪一行是适当的参照。这是一个强制外键。

示例 2

假设数据库中还包含一个列出了办事处地点的办事处表。Employees 表中可能有一个办事处表的外键，显示雇员工作的办事处位于哪座城市。该数据库的设计人员可以选择在聘用雇员时不指派办事处地点，例如，因为尚未将新雇员分配到办事处，或者因为他们不在外面的办事处工作。在这种情况下，该外键可以允许空值，因此是可选外键。

外键实施参照完整性

与主键一样，可以使用 CREATE TABLE 或 ALTER TABLE 语句创建外键。一旦创建了外键，该键中的列就只能包含通过该外键关联的表中的主键值。

有关创建外键的详细信息，请参见“管理主键”一节第 22 页。

丢失参照完整性

如果有人执行了以下操作，数据库可能会丢失参照完整性：

- 更新或删除主键值。所有引用该主键的外键都将变为无效。
- 在外表中添加一个新行，并为外键输入一个没有相应主键值的值。该数据库将不满足完整性要求。

SQL Anywhere 对这两类完整性丢失都提供保护。

如果客户端应用程序破坏了参照完整性

如果客户端应用程序在一个表中更新或删除了一个主键值，而数据库中其它的地方有一个外键引用该主键值，则会有破坏参照完整性的危险。

示例

如果服务器允许更新或删除该主键，并且未对引用该主键的外键进行任何变更，该外键引用将变为无效。任何使用该外键引用的尝试（例如，在 SELECT 语句中使用 KEY JOIN 子句）都将失败，因为被引用表中没有相应的值。

由于 SQL Anywhere 通常以一种比较直接的方式来处理对实体完整性的破坏，只是简单地拒绝输入数据并返回一条错误消息，因此对参照完整性的潜在破坏变得更为复杂。有几种方式（称作参照完整性操作）可以帮助您保持参照完整性。

参照完整性动作

在更新或删除被引用的主键时保持参照完整性的操作非常简单，只需拒绝进行更新或删除。但常常也可以对每个外键执行具体的操作来保持参照完整性。数据库管理员和表的所有者可以使用 CREATE TABLE 和 ALTER TABLE 语句指定，在完整性被破坏时，对引用被修改的主键的外键执行哪些操作。

可以为更新和删除主键的情况分别指定各个可用参照完整性操作：

- **RESTRICT** 生成一条错误消息，并在尝试变更被引用的主键值时，拒绝进行修改。这是缺省的参照完整性操作。
- **SET NULL** 将所有参照被修改的主键的外键设置为 NULL。
- **SET DEFAULT** 将所有引用被修改的主键的外键设置为该列的缺省值（在表定义中指定的缺省值）。
- **CASCADE** 与 ON UPDATE 一起使用时，此动作将所有引用已更新主键的外键更新为新值。与 ON DELETE 一起使用时，此动作将删除所有包含引用已删除主键的外键的行。

参照完整性动作由系统触发器执行。该触发器在主表上定义，但使用辅助表的所有者权限执行。这一行为意味着可以在所有者不同的表之间进行级联操作，而无需获得额外权限。

参照完整性检查

对于定义为 RESTRICT 操作的外键，如果其破坏了参照完整性，则将在执行语句时进行缺省检查动作。如果指定了 CHECK ON COMMIT 子句，则只会在提交事务时进行该检查操作。

使用数据库选项控制检查时间

对于定义为 RESTRICT 操作的外键，如果其破坏了参照完整性，设置 wait_for_commit 数据库选项可以控制该行为。CHECK ON COMMIT 子句可以替换此选项。

将 wait_for_commit 设置为 Off（缺省设置），将不允许执行会导致数据库不一致的操作。例如，尝试删除仍具有雇员的部门将会失败。以下语句将出错：

```
DELETE FROM Departments
WHERE DepartmentID = 200;
```

把 wait_for_commit 设置为 On 将导致执行提交之前不对参照完整性进行检查。如果数据库处于不一致状态，将不允许执行提交操作，并报告一个错误。在此模式下，虽然数据库用户可以删除仍含有雇员的部门，但在执行以下操作之前该用户不能将更改提交到数据库：

- 删除或重新分配属于该部门的雇员。
- 将 DepartmentID 行插回到 Departments 表中。
- 回退事务，撤消 DELETE 操作。

执行 INSERT 时检查完整性

SQL Anywhere 在执行 INSERT 语句时可执行完整性检查。例如，假设您尝试创建一个部门，但所提供的 DepartmentID 值已被使用：

```
INSERT
INTO Departments ( DepartmentID, DepartmentName, DepartmentHeadID )
VALUES ( 200, 'Eastern Sales', 902 );
```

由于该表的主键不再唯一，因而 INSERT 语句将被拒绝。因为 DepartmentID 列是主键，所以不允许出现重复的值。

插入违反关系的值

以下语句在 SalesOrders 表中插入一个新行，但错误地提供了在 Employees 表中不存在的 SalesRepresentative ID。

```
INSERT
INTO SalesOrders ( ID, CustomerID, OrderDate, SalesRepresentative)
VALUES ( 2700, 186, '2000-10-19', 284 );
```

根据 SalesOrders 表的 SalesRepresentative 列和 Employees 表的 EmployeeID 列，Employees 表和 SalesOrders 表之间存在一个一对多关系。只有在主表 (Employees) 中输入一个记录后，才能在外表 (SalesOrders) 中插入相应记录。

外键

Employees 表的主键是雇员 ID 号。SalesRepresentative 表中的销售代表 ID 号是 Employees 表的外键，这意味着，SalesOrders 表中的每一销售代表号都必须与 Employees 表中某些雇员的雇员 ID 号匹配。

在您试图为销售代表 284 添加订单时，将收到类似于以下的错误消息：**表 'SalesOrders' 中的外键 'FK_SalesRepresentative_EmployeeID' 没有主键值**

在 Employees 表中没有具有该 ID 号的雇员。以此防止您在不具备有效销售代表 ID 的情况下插入订单。

另请参见

- “表间的关系”一节 《SQL Anywhere 11 - 简介》

执行 DELETE 或 UPDATE 时检查完整性

在执行更新或删除操作时也可能引起外键错误。例如，假设您尝试从 Departments 表中删除研发部。DepartmentID 字段作为 Departments 表的主键，构成了一对多关系中的 "一" 方（Employees 表的 DepartmentID 字段是相应的外键，构成了 "多" 方）。在关系中的 [多] 方上的所有相应记录未全部删除前，不可以删除 [一] 方上的记录。

执行 DELETE 时的参照完整性错误

假设尝试删除 Departments 表中的研发部 (DepartmentID 100)。将报告一个错误，指出在数据库中有其它记录引用了研发部，因此未执行删除操作。若要删除研发部，需要先删除该部门中的所有雇员，操作如下：

```
DELETE
FROM Employees
WHERE DepartmentID = 100;
```

由于已删除了属于研发部的所有雇员，现在可以删除研发部了：

```
DELETE
FROM Departments
WHERE DepartmentID = 100;
```

输入 ROLLBACK 语句取消对数据库的这些更改：

```
ROLLBACK;
```

执行 UPDATE 时的参照完整性错误

现在，假设您尝试更改 Employees 表的 DepartmentID 字段。DepartmentID 字段作为 Employees 表的外键，构成了一对多关系中的 "多" 方（Departments 表的 DepartmentID 字段是相应的主键，构成了 "一" 方）。关系的 [多] 方上的记录不能更改，除非它对应于 [一] 方上的记录。也即，除非它有一个用于引用的主键。

例如，以下 UPDATE 语句会导致完整性错误：

```
UPDATE Employees
SET DepartmentID = 600
WHERE DepartmentID = 100;
```

这会引发错误 [**表 'Employees' 中的外键 'FK_DepartmentID_DepartmentID' 没有主键值**]，因为 Departments 表中没有 DepartmentID 为 600 的部门。

若要更改 Employees 表中 DepartmentID 字段的值，它必须对应于 Departments 表中已经存在的值。例如：

```
UPDATE Employees
SET DepartmentID = 300
WHERE DepartmentID = 100;
```

此语句可以执行，因为字段 DepartmentID 的值 300 对应于已经存在的财务部。

输入 ROLLBACK 语句取消对数据库的这些更改：

```
ROLLBACK;
```

在提交时检查完整性

在上述示例中，都在执行每一命令时检查数据库的完整性。任何会导致数据库不一致的操作都不会执行。

可以使用 wait_for_commit 选项将数据库配置为在提交前不检查完整性。如果在发生更改时您需要进行可能会导致数据暂时不一致的更改，这会非常有用。例如，假设您要删除 Employees 和 Departments 表中的研发部。由于这些表互相引用且必须一次对一个表进行删除，因此在删除期间表之间会出现不一致。这种情况下，数据库在完成删除前无法执行 COMMIT。将 wait_for_commit 选项设置为 On，以在执行提交前允许存在数据不一致情况。请参见“wait_for_commit 选项 [数据库]”一节 《SQL Anywhere 服务器 - 数据库管理》。

您也可以将外键定义为能自动进行修改以与主键的更改保持一致。在上例中，如果已使用 ON DELETE CASCADE 定义从 Employees 到 Departments 的外键，则若删除部门 ID，将自动删除 Employees 表中的对应条目。

在前面的几种情况中，没有任何方法可以将不一致的数据库作为永久数据库提交。SQL Anywhere 还支持在更改造成数据库不一致时使用替代操作。请参见“确保数据完整性”第 73 页。

系统表中的完整性规则

有关数据库完整性检查和规则的所有信息都存储在系统表中。使用如下的对应系统视图访问此信息：

系统视图	说明
SYS.SYSCONSTRAINT	SYS.SYSCONSTRAINT 系统视图中的每一行描述数据库中的一个约束。目前支持的约束包括表和列检查、主键、外键和唯一约束。请参见“SYSCONSTRAINT 系统视图”一节 《SQL Anywhere 服务器 - SQL 参考》。 对于表和列检查约束，实际的 CHECK 条件包含在 SYS.ISYSCHECK 系统表中。请参见“SYSCHECK 系统视图”一节 《SQL Anywhere 服务器 - SQL 参考》。
SYS.SYSCHECK	SYS.SYSCHECK 系统视图中的每一行均定义 SYS.SYSCONSTRAINT 系统视图中列出的一个检查约束。请参见“SYSCHECK 系统视图”一节 《SQL Anywhere 服务器 - SQL 参考》。
SYS.SYSFKEY	SYS.SYSFKEY 系统视图中的每一行描述了一个外键，包括为该键定义的匹配类型。请参见“SYSFKEY 系统视图”一节 《SQL Anywhere 服务器 - SQL 参考》。
SYS.SYSIDX	SYS.SYSIDX 系统视图中的每一行均定义数据库中的一个索引。请参见“SYSIDX 系统视图”一节 《SQL Anywhere 服务器 - SQL 参考》。
SYS.SYSTRIGGER	SYS.SYSTRIGGER 系统视图中的每一行都描述了数据库中的一个触发器，包括自动为具有参照触发操作（例如 ON DELETE CASCADE）的外键约束创建的触发器。 referential_action 列包含一个字符，该字符指出操作是级联 (C)、删除 (D)、设置为空值 (N)，还是限制 (R)。 事件列包含一个字符，指定引起操作发生的事件：A=插入和删除、B=插入和更新、C=更新、D=删除、E=删除和更新、I=插入、U=更新、M=插入、删除和更新。 trigger_time 列显示是在发生触发事件之后 (A) 还是之前 (B) 执行该操作。请参见“SYSTRIGGER 系统视图”一节 《SQL Anywhere 服务器 - SQL 参考》。

使用事务和隔离级别

目录

为确保数据完整性，必须能确定数据库中的信息在哪些状态下是**一致的**。下面的示例可以很好地说明一致性的概念：

一致性示例

假设您使用数据库来管理财务帐户，并且要将资金从一位客户的帐户转到另一位客户的帐户上。转帐前后数据库都处于一致状态；但在将资金从一个帐户中划出尚未存入另一个帐户之前，数据库处于不一致状态。转帐过程中，当客户帐户上的总金额与转帐前的总金额相同时，数据库就处于一致状态。但在转帐过程进行到一半时，数据库处于不一致状态。取款和存款操作必须都执行，或都不执行。

事务是一个逻辑工作单元

事务是一个逻辑工作单元。每个事务都是一系列在逻辑上相关的命令，用于执行一项任务，并将数据库从一种一致状态转换到另一种一致状态。什么样的状态是一致状态取决于您的数据库。

事务内的语句被视为不可分的单元：或者所有语句都执行，或者都不执行。在每个事务的末尾，需要**提交**所做更改，将这些更改永久应用到数据库中。如果由于某种原因事务中的一些语句未正确处理，则任何中间更改都会被撤消，或**回退**。这种特性有另外一个说法，即事务是**原子的**。

将语句组合成事务对于保护数据的一致性（即使在出现介质故障或系统故障时）和管理并发数据库操作都非常关键。可以安全地交错执行事务，并且每个事务的完成都标志着一个时间点，在该时间

点数据库中的信息处于一致状态。每个事务的设计宗旨都是执行一项将数据库从一个一致状态更改为另一个一致状态的任务。

在正常运行期间，如果出现系统故障或数据库崩溃，SQL Anywhere 将在下次启动数据库时自动恢复数据。自动恢复过程将恢复所有已完成的事务，并回退在发生故障时任何尚未提交的事务。事务的原子特性可以确保将数据库恢复到一个一致状态。

有关数据库备份和数据恢复的详细信息，请参见“备份和数据恢复”《SQL Anywhere 服务器 - 数据库管理》。

有关并发数据库使用的详细信息，请参见“并发简介”一节第 103 页。

使用事务

SQL Anywhere 期望您将命令组合到事务中。您提交事务以使对数据库进行的更改成为永久更改。更改数据时，所做的更改会记录在事务日志中，直到输入 COMMIT 命令后才成为永久更改。

事务起始于以下事件之一：

- 在与数据库连接后执行的第一个语句。
- 在事务结束后执行的第一个语句。

以下事件之一将完成事务：

- 执行 COMMIT 语句使对数据库进行的更改成为永久更改。
- 执行 ROLLBACK 语句撤消由事务执行的所有更改。
- 执行具有自动提交副作用的语句：数据定义命令（如 ALTER、CREATE、COMMENT 和 DROP）都具有自动提交副作用。
- 与数据库断开连接将执行隐式回退。
- ODBC 和 JDBC 具有一个自动提交设置，可以在每个语句后强制执行 COMMIT 命令。缺省情况下，ODBC 和 JDBC 要求将自动提交设置为 ON，并且要求每个语句都是一个事务。如果要利用事务设计的各种可能性，则应关闭自动提交设置。

  有关自动提交的详细信息，请参见“设置自动提交或手工提交模式”一节 《SQL Anywhere 服务器 - 编程》。
- 将 chained 数据库选项设置为 Off 与在每个语句后强制自动提交类似。缺省情况下，那些使用 jConnect 或 Open Client 应用程序的连接已将 chained 设置为 Off。

  有关详细信息，请参见“设置自动提交或手工提交模式”一节 《SQL Anywhere 服务器 - 编程》和“chained 选项 [兼容性]”一节 《SQL Anywhere 服务器 - 数据库管理》。

Interactive SQL 中的选项

Interactive SQL 提供两个选项，可用来控制事务结束的时间和方式：

- 如果您将 auto_commit 选项设置为 On，则 Interactive SQL 将自动在成功执行每一语句后提交您的结果，并且自动在每一失败的语句后执行 ROLLBACK。请参见“auto_commit 选项 [Interactive SQL]”一节 《SQL Anywhere 服务器 - 数据库管理》。
- commit_on_exit 选项的设置控制退出 Interactive SQL 时对未提交的更改执行的操作。如果该选项设置为 On（缺省设置），则 Interactive SQL 执行 COMMIT 语句；否则，它会用 ROLLBACK 语句撤消未提交的更改。请参见“commit_on_exit 选项 [Interactive SQL]”一节 《SQL Anywhere 服务器 - 数据库管理》。

在 Interactive SQL 中使用数据源
缺省情况下，ODBC 在自动提交模式下工作。即使已在 Interactive SQL 中将 auto_commit 选项设置为 Off，ODBC 的设置也将替换 Interactive SQL 的设置。可以使用 SQL_ATTR_AUTOCOMMIT 连接属性更改 ODBC 的设置。ODBC 自动提交与 chained 选项无关。

SQL Anywhere 也支持 Transact-SQL 命令（例如 BEGIN TRANSACTION），以便和 Sybase Adaptive Server Enterprise 兼容。有关详细信息，请参见“Transact-SQL 兼容性”一节第 612 页。

确定事务的开始时间

TransactionStartTime 数据库属性返回数据库在 COMMIT 或 ROLLBACK 操作后第一次修改的时间。可以使用此属性查找所有活动连接的最早事务的开始时间。例如：

```
BEGIN
  DECLARE connid int;
  DECLARE earliest char(50);
  DECLARE connstart char(50);
  SET connid=next_connection(null);
  SET earliest = NULL;
  lp: LOOP
  IF connid IS NULL THEN LEAVE lp END IF;
    SET connstart = CONNECTION_PROPERTY('TransactionStartTime',connid);
    IF connstart <> '' THEN
      IF earliest IS NULL
      OR CAST(connstart AS TIMESTAMP) < CAST(earliest AS TIMESTAMP) THEN
        SET earliest = connstart;
      END IF;
    END IF;
    SET connid=next_connection(connid);
  END LOOP;
  SELECT earliest
END
```


并发简介

并发是指数据库服务器同时处理多个事务的能力。如果数据库服务器中没有特殊的机制，并发事务可能会相互干扰，导致信息不一致或不正确。例如，百货商店中的数据库系统必须允许多个店员同时更新顾客帐户。各个店员在协助每位顾客购物时必须能及时更新帐户的状态：他们不能等到没有其他人使用该数据库时再进行更新。

谁需要了解并发

并发是所有数据库管理员和开发人员都应考虑的问题。即使对于单用户数据库，如果要处理来自多个应用程序（甚至一个应用程序的多个连接）的请求，也必须考虑并发问题。这些应用程序和连接之间也会发生干扰，其方式与网络设置中的多个用户完全相同。

事务大小影响并发

将 SQL 语句组合成事务的方式对数据完整性和系统性能有很大影响。如果事务太短，不包含完整的逻辑工作单元，则可能会导致数据库中出现不一致。如果将事务编写得太长并包含几个不相关的操作，则在执行 ROLLBACK 时很可能会不必要地撤消本可以安全提交到数据库中的工作。

如果事务很长，则会降低并发性能，因为这些事务会导致无法并发处理其它事务。

决定事务的长度的因素有许多，具体取决于应用程序的类型和环境。

要进一步了解有关运行并发 SQL Anywhere 数据库服务器的信息，请参见“运行 SQL Anywhere 数据库服务器简介”一节 《SQL Anywhere 服务器 - 数据库管理》。

事务内的保存点

使用**保存点**将各组相关语句分开可以在事务内标识重要的状态，并且可以有选择地返回到这些状态。

SAVEPOINT 语句定义事务执行过程中的中间点。使用 ROLLBACK TO SAVEPOINT 语句可以撤消该点后的所有更改。执行了 RELEASE SAVEPOINT 语句或事务结束后，则不能再使用该保存点。请注意，保存点对提交没有影响。当执行 COMMIT 时，该事务内的所有更改都在数据库中永久生效。

RELEASE SAVEPOINT 或 ROLLBACK TO SAVEPOINT 命令不会释放任何锁：只有在事务结束后才会释放锁。

命名和嵌套保存点

如果对保存点进行命名和嵌套，一个事务内就可以有多个活动保存点。SAVEPOINT 和 RELEASE SAVEPOINT 之间的更改可以通过回退到前一个保存点或回退事务本身来取消。在提交事务之前，事务内的更改还没有永久地应用到数据库中。事务结束后，所有保存点都将被释放。

不能在批量操作模式下使用保存点。使用保存点的附加开销很少。

隔离级别和一致性

使用 SQL Anywhere 可以控制一个事务中的操作对其它并发事务中的操作的可见程度。方法是设置名为**隔离级别**的数据库选项。

SQL Anywhere 还允许您使用相应的表提示控制查询中单个表的隔离级别。请参见“FROM 子句”一节 《SQL Anywhere 服务器 - SQL 参考》。

SQL Anywhere 提供以下隔离级别：

此隔离级别……	具有这些特征……
0—读取未提交数据	● 允许读取有写锁定或无写锁定的行 ● 未应用读锁定 ● 无法确保并发事务将不会修改行或回退对行所做的更改 ● 对应于表提示 NOLOCK 和 READUNCOMMITTED ● 允许脏读、非可重复读取和幻像行
1—读取已提交数据	● 只允许读取没有写锁定的行 ● 仅为读取当前行获取并保持读锁定，但当游标离开该行时释放读锁定 ● 无法确保数据在事务执行过程中不发生更改 ● 对应于表提示 READCOMMITTED ● 阻止进行脏读 ● 允许非可重复读取和幻像行
2—可重复读取	● 只允许读取没有写锁定的行 ● 读取结果集中的每一行时获取读锁定，并一直保持到事务结束为止 ● 对应于表提示 REPEATABLEREAD ● 阻止进行脏读和非可重复读取 ● 允许幻像行
3—可序列化	● 只允许读取结果中没有写锁定的行 ● 打开游标时获取读锁定，并一直保持到事务结束为止 ● 对应于表提示 HOLDLOCK 和 SERIALIZABLE ● 阻止脏读、非可重复读取和幻像行

此隔离级别……	具有这些特征……
快照 [1]	● 未应用读锁定 ● 允许读取任何行 ● 事务读取或更新第一行时，将使用已提交数据的数据库快照
语句快照 [1]	● 未应用读锁定 ● 允许读取任何行 ● 语句读取第一行时，将使用已提交数据的数据库快照
只读语句快照 [1]	● 未应用读锁定 ● 允许读取任何行 ● 只读语句读取第一行时，将使用已提交数据的数据库快照 ● 对可更新语句使用 updatable_statement_isolation 选项指定的隔离级别（0、1、2 或 3）

[1] 必须为数据库启用快照隔离，方法是将数据库的 allow_snapshot_isolation 选项设置为 On。请参见“启用快照隔离”一节第 109 页。

缺省隔离级别为 0，但 Open Client、jConnect 和 TDS 连接除外，它们的缺省隔离级别为 1。

有关 MobiLink 隔离级别的信息，请参见“MobiLink 隔离级别”一节 《MobiLink - 服务器管理》。

基于锁的隔离级别可以防止部分或全部干扰。级别 3 是最高隔离级别。较低的级别允许的不一致也较多，但通常性能会更高。级别 0（读取未提交数据）为缺省设置。

快照隔离级别可以防止读取和写入操作之间的所有干扰。但写入操作之间仍可能会相互干扰。此级别几乎不会出现不一致的情况，其争用性能与隔离级别 0 相同。由于需要保存和使用行版本，因此与争用无关的性能会变差。

注意
所有隔离级别都确保每个事务或者完全执行，或者根本不执行，并且不会丢失任何更新。

隔离仅在事务之间执行：同一事务中的多个游标可能互相干扰。

快照隔离

多个用户同时读取和写入相同数据时，可能发生阻塞和死锁。快照隔离旨在通过维护数据的不同版本提高并发性和一致性。在事务中使用快照隔离时，数据库服务器会返回数据的已提交版本，以响应任何读请求。它在进行此操作时不获取读锁定，因此不会对正在写入数据的用户造成干扰。

快照是数据库中已提交的一组数据。使用快照隔离时，事务中的所有查询都会使用同一组数据。数据库表不会获取任何锁，因此其它事务可以无阻塞地访问并修改数据。SQL Anywhere 支持以下三种快照隔离级别，允许您控制何时使用快照：

- **快照** 从事务读取、插入、更新或删除第一行时开始，使用已提交数据的快照。
- **语句快照** 从语句读取第一行开始，使用已提交数据的快照。事务内的每个语句看到的都是不同时间的数据快照。
- **只读语句快照** 对于只读语句，从读取第一行时开始，使用已提交数据的快照。事务内的每个只读语句看到的都是不同时间的数据快照。对于插入、更新和删除语句，使用由 updatable_statement_isolation 选项指定的隔离级别（可以是 0（缺省值）、1、2 或 3）。

您还可以使用 BEGIN SNAPSHOT 语句，指定何时启动事务的快照。请参见“BEGIN SNAPSHOT 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

快照隔离在许多情况下都很有用，例如：

- **执行大量读取操作和很少更新操作的应用程序** 快照事务仅为那些修改数据库的语句获取写锁定。如果事务主要执行读取操作，则快照事务不会获取可能干扰其他用户的事务的读锁定。
- **其他用户需要访问数据时执行长时间运行的事务的应用程序** 快照事务不获取读锁定，因此快照事务发生时，其他用户可以读取和更新数据。
- **必须从数据库读取一致的数据集的应用程序** 快照显示特定时间点的已提交数据集，因此您可以使用快照隔离来查看在整个事务中未更改的一致数据，即使其他用户在您的事务运行时对数据进行更改。

快照隔离仅影响所有用户共享的基表和全局临时表。对其它表类型的读取操作决不会看到数据的旧版本，且从不启动快照。仅当将 isolation_level 选项设置为快照，且更新启动一个事务时，对其它表类型的更新才启动快照。

当存在使用 WITH HOLD 子句打开的使用语句或事务快照的游标时，以下语句无法执行。

- ALTER INDEX
- ALTER TABLE
- CREATE INDEX
- DROP INDEX
- REFRESH MATERIALIZED VIEW
- REORGANIZE TABLE
- CREATE TEXT INDEX
- REFRESH TEXT INDEX

当使用 WITH HOLD 子句打开游标时，可看到在快照启动时所提交的所有行的快照。还可看到从在其内打开游标的事务启动以来，由当前连接完成的所有修改。

仅当未执行快速截断时允许执行 TRUNCATE TABLE，因为在此情况下，单独的 DELETE 操作随后会被记录在事务日志中。请参见“TRUNCATE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

此外，如果从非快照事务中执行这些语句中的任何语句，则已运行的快照事务在随后尝试使用表时，会返回一条错误，指示模式已更改。

如果视图在事务的快照开始后进行刷新，则实例化视图匹配将避免使用该视图。

所有编程接口都支持快照隔离级别。可以使用 SET OPTION 语句设置隔离级别。有关使用快照隔离的信息，请参见：

- “isolation_level 选项 [数据库] [兼容性]”一节 《SQL Anywhere 服务器 - 数据库管理》
- ADO 和 OLE DB： “使用事务”一节 《SQL Anywhere 服务器 - 编程》
- ADO.NET： “IsolationLevel 属性”一节 《SQL Anywhere 服务器 - 编程》

行版本

为数据库启用快照隔离后，每次更新行时，数据库服务器会将原始行的副本添加到存储在临时文件内的版本中。原始行版本条目会一直存储，直到可能需要访问原始行值的所有活动的快照事务完成。使用快照隔离的事务只能看到已提交的值，因此，如果在快照事务开始之前未提交或回退对行的更新，则快照事务需要能够访问原始行值。这就允许使用快照隔离的事务在查看数据时，不会在基础表上放置任何锁。

VersionStorePages 数据库属性返回当前用于版本存储的临时文件中的页数。要获得此值，请执行以下查询：

```
SELECT DB_PROPERTY ( 'VersionStorePages' );
```

当不再需要旧行版本条目时，会将它们删除。旧版本的 BLOB 会一直存储在原始表（不是临时表）中，直到不再需要它们，旧行版本的索引条目也会一直存储在原始索引中，直到不再需要它们。

可以使用 sa_disk_free_space 系统过程检索临时文件中可用空间的大小。请参见“sa_disk_free_space 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》。

如果触发了更新行值的触发器，则那些行的原始值也会存储在临时文件中。

将应用程序设计成使用更短的事务和更短的快照，会减少临时文件的空间需求。

如果您担心临时文件的增大，则可建立一个 GrowTemp 系统事件，指定临时文件达到特定大小时要采取的操作。请参见“了解系统事件”一节 《SQL Anywhere 服务器 - 数据库管理》。

了解快照事务

快照事务在更新时获取写锁定，而使用快照的事务或语句永远不会获取读锁定。因此，读取程序从不阻塞写入程序，写入程序也从不阻塞读取程序，但尝试更新相同的行时，写入程序之间可能会互相阻塞。

请注意，就快照隔离而言，事务不以 BEGIN TRANSACTION 语句开始，而是以事务中的第一个读取、插入、更新或删除操作开始，具体取决于用于事务的快照隔离级别。以下示例说明快照隔离事务何时开始：

```
SET OPTION PUBLIC.allow_snapshot_isolation = 'On';
          SET TEMPORARY OPTION isolation_level = 'snapshot';
   SELECT * FROM Products; --transaction begins and the statement only
                           --sees changes that are already committed
   INSERT INTO Products
          SELECT ID + 30, Name, Description,
          'Extra large', Color, 50, UnitPrice, NULL
          FROM Products
          WHERE Name = 'Tee Shirt';
COMMIT; --transaction ends
```


启用快照隔离

使用 allow_snapshot_isolation 选项可为数据库启用或禁用快照隔离。当将此选项设置为 On 时，会在临时文件中维护行版本，且允许连接使用任何快照隔离级别。当将此选项设置为 Off 时，任何使用快照隔离的尝试都会导致错误。

使数据库启用快照隔离可能会影响性能，因为无论使用快照隔离的事务有多少，都必须维护所有修改行的副本。请参见“游标敏感性和隔离级别”一节 《SQL Anywhere 服务器 - 编程》。

以下语句为数据库启用快照隔离：

```
SET OPTION PUBLIC.allow_snapshot_isolation = 'On';
```

allow_snapshot_isolation 选项的设置可以更改，即使有用户连接到数据库。将此选项的设置从 Off 更改到 On 时，必须完成所有的当前事务后，新事务才能使用快照隔离。将此选项的设置从 On 更改到 Off 时，必须完成使用快照隔离的所有未完成事务后，数据库服务器才停止维护行版本信息。

可以通过查询 SnapshotIsolationState 数据库属性的值查看数据库的当前快照隔离设置：

```
SELECT DB_PROPERTY ( 'SnapshotIsolationState' );
```

SnapshotIsolationState 属性的值可为以下值之一：

- **On** 为数据库启用快照隔离。
- **Off** 为数据库禁用快照隔离。
- **in_transition_to_on** 一旦当前事务完成，就将启用快照隔离。
- **in_transition_to_off** 一旦当前事务完成，就将禁用快照隔离。

为数据库启用快照隔离后，必须为事务维护行版本（即使未使用快照），直到事务提交或回退。因此，如果从不使用快照隔离，最好将 allow_snapshot_isolation 选项设置为 Off。

快照隔离示例

以下示例使用两个到 SQL Anywhere 示例数据库的连接，说明如何使用快照隔离无阻塞地维护一致性。

♦ 使用快照隔离

1. 执行以下命令，创建一个到 SQL Anywhere 示例数据库的 Interactive SQL 连接 (Connection1)：

   ```
   dbisql -c "DSN=SQL Anywhere 11
   Demo;UID=DBA;PWD=sql;ConnectionName=Connection1"
   ```

2. 执行以下命令，创建一个到 SQL Anywhere 示例数据库的 Interactive SQL 连接 (Connection2)：

   ```
   dbisql -c "DSN=SQL Anywhere 11
   Demo;UID=DBA;PWD=sql;ConnectionName=Connection2"
   ```

3. 在 Connection1 中，执行以下命令将隔离级别设置为 1（读取已提交的），该隔离级别在当前行上获取并保持读锁定。

```
SET OPTION isolation_level = 1;
```

4. 在 Connection1 中，执行以下命令：

```
SELECT * FROM Products;
```

ID	Name	Description	Size	Color	Quantity	...
300	Tee Shirt	Tank Top	Small	White	28	...
301	Tee Shirt	V-neck	Medium	Orange	54	...
302	Tee Shirt	Crew Neck	One size fits all	Black	75	...
400	Baseball Cap	Cotton Cap	One size fits all	Black	112	...
...	...	...	...	...	...	...

5. 在 Connection2 中，执行以下命令：

```
UPDATE Products
SET Name = 'New Tee Shirt'
WHERE ID = 302;
```

6. 在 Connection1 中，再次执行 SELECT 语句：

```
SELECT * FROM Products;
```

该 SELECT 语句被阻塞且无法继续执行，因为 Connection2 中的 UPDATE 语句未提交或回退。必须等到 Connection2 中的事务完成后，SELECT 语句才能继续执行。这确保了 SELECT 语句不会将未提交的数据读入其结果中。

7. 在 Connection2 中，执行以下命令：

```
ROLLBACK;
```

Connection2 中的事务完成，Connection1 中的 SELECT 语句继续执行。

8. 使用语句快照隔离级别可以获得与隔离级别 1 相同的并发，但不造成阻塞。

在 Connection1 中，执行以下命令允许快照隔离：

```
SET OPTION PUBLIC.allow_snapshot_isolation = 'On';
```

9. 在 Connection 1 中，执行以下命令将隔离级别更改为语句快照：

```
SET TEMPORARY OPTION isolation_level = 'statement-snapshot';
```

10. 在 Connection1 中，执行以下语句：

```
SELECT * FROM Products;
```

11. 在 Connection2 中，执行以下语句：


```
UPDATE Products
SET Name = 'New Tee Shirt'
WHERE ID = 302;
```

12. 在 Connection1 中，再次发出 SELECT 语句：

```
SELECT * FROM Products;
```

SELECT 语句执行且不被阻塞，但不包括来自 Connection2 执行的 UPDATE 语句的数据。

13. 在 Connection2 中，通过执行以下命令完成事务：

```
COMMIT;
```

14. 在 Connection1 中，完成事务（对 Products 表的查询），然后再次执行 SELECT 语句，以查看更新后的数据：

```
COMMIT;
SELECT * FROM Products;
```

ID	Name	Description	Size	Color	Quantity	...
300	Tee Shirt	Tank Top	Small	White	28	...
301	Tee Shirt	V-neck	Medium	Orange	54	...
302	New Tee Shirt	Crew Neck	One size fits all	Black	75	...
400	Baseball Cap	Cotton Cap	One size fits all	Black	112	...
...	...	...	...	...	...	...

15. 通过执行以下语句，撤销对 SQL Anywhere 示例数据库所做的更改：

```
UPDATE Products
SET Name = 'Tee Shirt'
WHERE id = 302;
COMMIT;
```

有关使用快照隔离的更多示例，请参见：

- “使用快照隔离避免脏读”一节第 139 页
- “使用快照隔离避免非可重复的读取”一节第 144 页
- “使用快照隔离避免幻像行”一节第 148 页

更新冲突和快照隔离

使用快照隔离时，如果事务查看行的旧版本并尝试更新或删除它，则可能发生更新冲突。发生这种情况时，服务器会在检测到冲突时给出错误。对于已提交的更改，服务器在尝试更新或删除时返回错误。对于未提交的更改，在更改提交时，更新或删除会阻塞并且服务器会返回错误。

使用只读语句快照时不会发生更新冲突，因为可更新语句以非快照隔离方式运行，并且总是看到数据库的最新版本。因此，只读语句快照隔离级别具有快照隔离的许多优点，且无需对最初设计为在其它隔离级别上运行的应用程序做很大改动。使用只读语句快照隔离级别时：

- 只读语句从不获取读锁定
- 只读语句总是查看数据库的已提交状态

典型的不一致类型

有三种典型的不一致类型在执行并发事务的过程中可能会出现。这个列表并不完整，还可能会出现其它类型的不一致。ISO SQL/2003 标准中提到了这三种类型，而且，较低隔离级别的行为是依据这些类型定义的，因此它们非常重要。

- **脏读** 事务 A 修改了一行，但未提交或回退所做更改。事务 B 读取了被修改的行。之后，事务 A 在执行 COMMIT 之前又对该行进行了更改，或回退了所做更改。这两种情况下，事务 B 读取到该行时，该行都处于从未被提交的状态。

  有关脏读的详细信息，请参见“教程：脏读”一节第 137 页。

- **非可重复读取** 事务 A 读取了一行。之后，事务 B 修改或删除了该行，并执行了 COMMIT 命令。事务 A 之后再次尝试读取同一行时，该行将已被更改或删除。

  有关非可重复读取的详细信息，请参见“教程：非可重复读取”一节第 141 页。

- **幻像行** 事务 A 读取了一组满足某个条件的行。之后，事务 B 执行 INSERT 命令，或对以前不满足 A 的条件的一行执行 UPDATE 命令。事务 B 提交了这些更改。这些新提交的行现在满足事务 A 的条件。如果事务 A 之后重复读取操作，则它将获得更新的行集。

  有关幻像行的详细信息，请参见“教程：幻像行”一节第 146 页。

隔离级别与脏读、非可重复读取和幻像行

SQL Anywhere 允许脏读、非可重复读取和幻像行（取决于所使用的隔离级别）。下表中的 X 表示该隔离级别允许该行为。

隔离级别	脏读	非可重复的读取	幻像行
0-读取未提交数据	X	X	X
只读语句快照	X[1]	X[2]	X[3]
1-读取已提交数据		X	X
语句快照		X[2]	X[3]
2-可重复读取			X
3-可序列化			

隔离级别	脏读	非可重复的读取	幻像行
快照			

[1] 如果 updatable_statement_isolation 选项指定的隔离级别不防止脏读的发生，则事务中的可更新语句可发生脏读。

[2] 如果 updatable_statement_isolation 选项指定的隔离级别不防止非可重复读取的发生，则事务中的语句可发生非可重复读取。非可重复读取之所以会发生，是因为每个语句启动一个新的快照，因此一个语句可能看到另一个语句看不到的更改。

[3] 如果 updatable_statement_isolation 选项指定的隔离级别不防止幻像行的发生，则事务中语句可发生幻像行。幻像行之所以会发生，是因为每个语句启动一个新的快照，因此一个语句可能看到另一个语句看不到的更改。

此表说明了以下两点：

- 每个隔离级别都消除了这三种典型的不一致类型中的一种。
- 每个级别都消除了所有较低级别所消除的不一致类型。
- 对于语句快照隔离级别，非可重复的读取和幻像行可在事务中发生，但不能在事务的单个语句中发生。

隔离级别在 ODBC 下具有不同的名称。这些名称是基于它们所防止的不一致的名称而命名的。请参见“ValuePtr 参数”一节第 115 页。

游标不稳定性

另一种重要的不一致是**游标不稳定性**。存在这种不一致时，一个事务可以修改正由另一事务的游标所引用的行。游标稳定性可以确保使用游标的应用程序不会导致数据库中的数据出现不一致。

示例

事务 A 使用游标读取了一行。事务 B 修改并提交了该行。在不知道该行已被修改的情况下，事务 A 对其进行了修改。

消除游标不稳定性

SQL Anywhere 在隔离级别 1、2 和 3 上提供了**游标稳定性**。游标稳定性可以确保任何其它事务都不能对游标的当前行中包含的信息进行修改。游标的一行中的信息可能是一个特定表中所包含的信息的副本，也可能是多个表的不同行中的数据的组合。只要在 SELECT 语句中使用了连接或子选择，就可能会涉及多个表。

有关 SQL 编程过程与游标的信息，请参见“使用过程、触发器和批处理”第 777 页。

只有在通过另一个应用程序使用 SQL Anywhere 时，才会使用游标。有关详细信息，请参见“在应用程序中使用 SQL”《SQL Anywhere 服务器 - 编程》。

对于使用游标的应用程序，需要考虑的一个相关但又独特的问题是，应用程序是否可以看到对基础数据的更改。通过指定游标的敏感性，可以控制应用程序能看到哪些更改。

有关游标敏感性的详细信息，请参见“SQL Anywhere 游标”一节 《SQL Anywhere 服务器 - 编程》。

设置隔离级别

数据库的每个连接都有其自己的隔离级别。此外，数据库也可以为每个用户或组存储一个缺省隔离级别。使用 isolation_level 数据库选项的 PUBLIC 设置可以为整个数据库组设置一个缺省隔离级别。

您也可以用表提示来设置隔离级别，但这是一个高级功能，只应在需要时使用它。有关详细信息，请参见“FROM 子句”一节 《SQL Anywhere 服务器 - SQL 参考》中的 WITH *table-hint* 一节。

使用 SET OPTION 命令可以更改您的连接的隔离级别，以及与您的用户 ID 相关联的缺省级别。如果有权限，还可以为其他用户或组更改隔离级别。

如果要使用快照隔离，则必须首先为数据库启用快照隔离。

有关启用和设置快照隔离级别的信息，请参见“启用快照隔离”一节第 109 页。

♦ 为当前用户设置隔离级别

- 执行 SET OPTION 语句。例如，以下语句将把当前用户的隔离级别设置为 3：

```
SET OPTION isolation_level = 3;
```

♦ 为用户或组设置隔离级别

1. 以具有 DBA 权限的用户身份连接到数据库。
2. 在 isolation_level 前加上该组的名称和一个句点，然后执行 SET OPTION 语句。例如，以下命令将把 PUBLIC 组的缺省隔离级别设置为 3。

```
SET OPTION PUBLIC.isolation_level = 3;
```

♦ 为当前连接设置隔离级别

- 使用 TEMPORARY 关键字执行 SET OPTION 语句。例如，以下语句将在您的当前连接期间把隔离级别设置为 3：

```
SET TEMPORARY OPTION isolation_level = 3;
```

缺省隔离级别

连接到数据库时，数据库服务器将按照以下方式确定您的初始隔离级别：

1. 可以为每个用户和组设置缺省隔离级别。如果您的用户 ID 在数据库中有一个存储的级别，数据库服务器将使用该级别。
2. 如果没有，数据库服务器将检查您所属的组，直到找到一个级别。所有用户都是特殊组 PUBLIC 的成员。如果之前未找到任何其它设置，SQL Anywhere 将使用指派给该组的级别。

有关用户和组的详细信息，请参见“管理用户 ID、特权和权限” 《SQL Anywhere 服务器 - 数据库管理》。

有关 SET OPTION 语句语法的详细信息，请参见“SET OPTION 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

有些情况下，可能需要在事务中间更改隔离级别，例如，在只有一个表或多个表需要序列化访问时。有关在事务内更改隔离级别的信息，请参见“在事务内更改隔离级别”一节第 116 页。

从支持 ODBC 的应用程序设置隔离级别

ODBC 应用程序调用 SQLSetConnectAttr，调用时将 Attribute 设置为 SQL_ATTR_TXN_ISOLATION 并根据相应隔离级别设置 ValuePtr：

ValuePtr 参数

ValuePtr	隔离级别
SQL_TXN_READ_UNCOMMITTED	0
SQL_TXN_READ_COMMITTED	1
SQL_TXN_REPEATABLE_READ	2
SQL_TXN_SERIALIZABLE	3
SA_SQL_TXN_SNAPSHOT	快照
SA_SQL_TXN_STATEMENT_SNAPSHOT	语句快照
SA_SQL_TXN_READONLY_STATEMENT_SNAPSHOT	只读语句快照

通过 ODBC 更改隔离级别

您可以使用 *ODBC32.dll* 库中的 SQLSetConnectOption 函数通过 ODBC 更改连接的隔离级别。

SQLSetConnectOption 函数采用三个参数：ODBC 连接句柄的值、您希望设置隔离级别这一事实，以及与该隔离级别相对应的值。下表中显示了这些值。

字符串	值
SQL_TXN_ISOLATION	108
SQL_TXN_READ_UNCOMMITTED	1
SQL_TXN_READ_COMMITTED	2
SQL_TXN_REPEATABLE_READ	4
SQL_TXN_SERIALIZABLE	8

字符串	值
SA_SQL_TXN_SNAPSHOT	32
SA_SQL_TXN_STATEMENT_SNAPSHOT	64
SA_SQL_TXN_READONLY_STATEMENT_SNAPSHOT	128

请不要使用 SET OPTION 语句从 ODBC 应用程序中更改隔离级别。由于 ODBC 驱动程序不分析语句，所以，在 ODBC 中执行任何语句都不会被 ODBC 驱动程序识别。这可能会导致意外的锁定行为。

示例

以下函数调用会将连接 MyConnection 的隔离级别设置为隔离级别 2：

```
SQLSetConnectOption( MyConnection.hDbc,
                     SQL_TXN_ISOLATION,
                     SQL_TXN_REPEATABLE_READ )
```

ODBC 使用隔离功能支持各种数据库锁选项。例如，在 PowerBuilder 中，可以使用事务对象的 Lock 属性在连接到数据库时设置隔离级别。Lock 属性是一个字符串，按以下方式设置：

```
SQLCA.lock = "RU"
```

Lock 选项仅在 CONNECT 发生时起作用。在 CONNECT 之后再更改 Lock 属性不会对连接产生任何影响。

在事务内更改隔离级别

不同的隔离级别可能适合于一个事务的不同部分。SQL Anywhere 支持在事务中间更改数据库的隔离级别。

如果在事务中间更改了 isolation_level 选项，新设置只会影响以下对象：

- 任何在更改后打开的游标
- 任何在更改后执行的语句

您可能希望在事务处理过程中更改隔离级别，以控制事务放置的锁数量。有的事务可能需要读取一个大型表，但却仅对其中少数几行执行具体的操作。如果不一致对该事务不会产生严重影响，则可以在扫描该大型表时设置较低的隔离级别，以避免延迟其它事务的操作。

在其它一些情况下，您也可能希望在事务中间更改隔离级别，例如，在只有一个表或一组表需要序列化访问时。

有关在事务中间更改隔离级别的示例，请参见“教程：幻像行”一节第 146 页。

注意
您也可以使用表提示设置隔离级别（仅限于级别 0-3），但是这是一个高级功能，只应在需要时使用它。有关详细信息，请参见“FROM 子句”一节 《SQL Anywhere 服务器 - SQL 参考》中的 WITH *table-hint* 一节。

使用快照隔离时更改隔离级别

使用快照隔离时，您可以在事务内更改隔离级别。可以通过更改 isolation_level 选项的设置，或通过使用影响查询中隔离级别的表提示，来完成此操作。可以随时使用语句快照、只读语句快照和隔离级别 0-3。但是，如果快照隔离级别在快照之外的隔离级别上开始，则您无法在事务中使用该快照隔离级别。事务由更新启动并继续执行，直到下一个 COMMIT 或 ROLLBACK。如果在快照之外的某个隔离级别上发生第一次更新，那么在事务提交或回退之前，尝试使用快照隔离级别的任何语句都会返回错误 -1065 NON_SNAPSHOT_TRANSACTION。例如：

```
SET OPTION PUBLIC.allow_snapshot_isolation = 'On';

BEGIN TRANSACTION
    SET OPTION isolation_level = 3;
    INSERT INTO Departments
        ( DepartmentID, DepartmentName, DepartmentHeadID )
        VALUES( 700, 'Foreign Sales', 129 );
    SET TEMPORARY OPTION isolation_level = 'snapshot';
    SELECT * FROM Departments;
```

查看隔离级别

使用 CONNECTION_PROPERTY 函数可以检查当前连接的隔离级别。

♦ **查看当前连接的隔离级别：**

● 执行以下语句：

```
SELECT CONNECTION_PROPERTY('isolation_level');
```

事务阻塞和死锁

执行事务时，数据库服务器将在行上放置锁，以防止其它事务干扰受影响的行。**锁**用于控制可允许的干扰程度和类型。

SQL Anywhere 利用**事务阻塞**机制使事务能够并发执行而不会发生干扰，或只有很少的干扰。一个事务能获取一个锁，以防止其它并发事务修改（甚至访问）特定的行。此事务阻塞方案总能避免某些类型的干扰。例如，当一个事务更新表中的某一特定行时，它将锁定该行以确保任何其它事务都不能同时更新或删除同一行。

事务阻塞

当一个事务尝试执行某个操作，但却因另一个事务持有的锁而被禁止时，将出现冲突，并且该事务尝试执行其操作的进程将被阻碍。

有时，一组事务会进入一种特殊状态，在该状态下这些事务都不能继续执行。有关详细信息，请参见“死锁”一节第 118 页。

blocking 选项

如果两个事务在同一行上分别获取了读锁定，当其中一个事务尝试修改该行时，所发生的情况将取决于 blocking 选项的设置。要修改这一行，该事务必须阻塞另一个事务，但由于另一个事务阻塞它，因此无法实现。

- 如果 blocking 选项被设置为 On（缺省设置），则尝试写入的事务将等到另一个事务释放其读锁定。那时，即可进行写入。
- 如果 blocking 选项被设置为 Off，则尝试写入的语句将收到一个错误。

当 blocking 选项被设置为 Off 时，该语句将终止而不会等待，而它所做的任何部分更改都将被回退。在这种情况下，请稍后再尝试执行该事务。

隔离级别越高，越容易出现阻塞，因为执行的锁定和检查更多。通常，隔离级别越高，并发性越低。并发性降低的程度取决于各个并发事务的特性。

有关 blocking 选项的详细信息，请参见“blocking 选项 [数据库]”一节 《SQL Anywhere 服务器 - 数据库管理》。

死锁

事务阻塞可能会导致**死锁**。出现死锁情况时，一组事务将进入一种特殊状态，在该状态下这些事务都不能继续执行。

死锁的原因

有两个原因会导致死锁：

- **循环阻塞冲突** 事务 A 被事务 B 阻塞，而事务 B 又被事务 A 阻塞。显然，等待并不能解决这个问题。必须取消其中一个事务，使另一个事务能够继续执行。如果多个（大于二）事务出现循环阻塞，也会导致这种情况。
- **所有活动数据库线程均被阻塞** 事务被阻塞时，不会释放其数据库线程。如果服务器配置了三个线程，而事务 A、B 和 C 在当前未执行请求的事务 D 上受到阻塞，由于没有可用线程，将出现死锁情况。

为消除事务死锁，SQL Anywhere 会从陷入死锁的连接中选择一个连接，回退对该连接上活动事务的更改并返回错误。SQL Anywhere 使用内部启发式算法选择要回退的连接，该算法优先选择剩余阻塞等待时间最少的连接（由 blocking_timeout 选项确定）。如果所有连接均设置为始终等待，那么促使服务器检测到死锁的连接会被选作牺牲品连接。

为消除线程死锁，SQL Anywhere 会选择最后一个线程进行阻塞，回退对该连接上活动事务的更改并返回错误。

服务器使用的数据库线程数取决于各个数据库的设置。

有关设置数据库线程数量的信息，请参见“控制线程行为”一节 《SQL Anywhere 服务器 - 数据库管理》。

确定被阻塞的连接

您可以使用 sa_conn_info 系统过程来确定哪些连接在其它哪些连接上受到阻塞。该过程将返回一个结果集，每个连接对应于该结果集中的一行。该结果集中有一列显示连接是否受到阻塞，如果是，则指出它在其它哪个连接上受到阻塞。

有关详细信息，请参见“sa_conn_info 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》。

您还可以使用 Deadlock 事件在发生死锁时执行操作。事件处理程序可以使用 sa_report_deadlocks 过程以获得有关导致死锁的条件的信息。要从数据库服务器检索有关死锁的更多详细信息，请使用 log_deadlocks 选项并启用 RememberLastStatement 功能。

以下过程说明如何设置在发生死锁时用于获取死锁相关信息的表和系统事件。如果发现应用程序频繁发生死锁，您可以使用应用程序分析功能帮助诊断死锁的原因。请参见“教程：诊断死锁”一节 第 230 页。

♦ 发生死锁时执行操作

1. 创建一个表以存储从 sa_report_deadlocks 系统过程返回的数据：

```
CREATE TABLE DeadlockDetails(
  deadlockId INT PRIMARY KEY DEFAULT AUTOINCREMENT,
  snapshotId BIGINT,
  snapshotAt TIMESTAMP,
  waiter INTEGER,
  who VARCHAR(128),
  what LONG VARCHAR,
  object_id UNSIGNED BIGINT,
  record_id BIGINT,
  owner INTEGER,
  is_victim BIT,
  rollback_operation_count UNSIGNED INTEGER );
```

2. 创建在发生死锁时触发的事件。

   此事件会将 sa_report_deadlocks 系统过程的结果复制到表中，并且通知管理员有关死锁的情况：

```
CREATE EVENT DeadlockNotification
TYPE Deadlock
HANDLER
BEGIN
 INSERT INTO DeadlockDetails WITH AUTO NAME
 SELECT snapshotId, snapshotAt, waiter, who, what, object_id, record_id,
        owner, is_victim, rollback_operation_count
    FROM sa_report_deadlocks ();
 COMMIT;
 CALL xp_startmail ( mail_user ='George Smith',
                     mail_password ='mypwd' );
 CALL xp_sendmail( recipient='DBAdmin',
                   subject='Deadlock details added to the DeadlockDetails
table.' );
 CALL xp_stopmail ( );
END;
```

3. 将 log_deadlocks 选项设置为 On：

```
SET OPTION PUBLIC.log_deadlocks = 'On';
```

4. 启用最近执行的语句的记录：

```
CALL sa_server_option( 'RememberLastStatement', 'YES' );
```

另请参见

- “log_deadlocks 选项 [数据库]”一节 《SQL Anywhere 服务器 - 数据库管理》
- “sa_report_deadlocks 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》
- “sa_server_option 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》
- “CREATE EVENT 语句”一节 《SQL Anywhere 服务器 - SQL 参考》

从 Sybase Central 查看死锁

在 Sybase Central 中连接到数据库时，由于 log_deadlocks 选项被设置为 On，因此您可以看到数据库中已发生的任何死锁的图示。死锁信息记录在内部缓冲区之中。

♦ 使用 Sybase Central 死锁报告

1. 请在 Sybase Central 的左窗格中选择数据库，然后选择 [**文件**] » [**选项**]。
2. 打开 log_deadlocks 选项。
   a. 在 [**选项**] 列表中选择 **log_deadlocks**。
   b. 在 [**值**] 字段中键入 **On**。
   c. 单击 [**立即设置永久值**]。
   d. 单击 [**关闭**]。

   有关详细信息，请参见“log_deadlocks 选项 [数据库]”一节 《SQL Anywhere 服务器 - 数据库管理》。

3. 在右窗格中，单击 [**死锁**] 选项卡。

   如果数据库中存在死锁，则会出现死锁图示。死锁图示中的每个节点都代表一个连接，并提供有关哪个连接被死锁、用户名和死锁发生时连接尝试执行的 SQL 语句等详细信息。有两类死锁：连接死锁和线程死锁。连接死锁的特征是节点的循环相关性。线程死锁由未在循环相关性中连接的节点表示，节点数等于数据库上的最大线程数加一。

锁定的工作方式

在数据库服务器处理事务时，可以锁定表中的一行或多行。锁可以通过防止其它事务并发访问来保持数据库中存储的信息的可靠性。锁还可以通过标识正在进行更新的信息来提高查询结果的准确性。

数据库服务器会自动放置这些锁，而无需显式指定。它将一直保留事务获取的所有锁，直到该事务完成（例如，通过使用 COMMIT 或 ROLLBACK 语句），但有一个例外。这个例外在“较早释放读锁定”一节第 133 页中进行了说明。

可以访问该行的事务被称为持有锁。其它事务也许可以对锁定行进行有限的访问，也许根本不能访问，这取决于锁的类型。

可以锁定的对象

为了确保数据库一致性并支持事务之间适当的隔离级别，SQL Anywhere 使用以下类型的锁：

- **模式锁** 这些锁控制进行模式更改的能力。例如，事务可以锁定表的模式，以防止其它事务修改该表的结构。
- **行锁** 这些锁用于确保行级别上并发事务之间的一致性。例如，事务可锁定特定的行以防止另一个事务对其进行更改，且如果要修改行，则事务必须在该行上放置一个写锁定。
- **表锁** 这些锁用于确保表级别上并发事务之间的一致性。例如，通过插入新列更改表结构的事务可锁定一个表，这样其它事务就不会受到模式更改的影响。在这种情况下，必须限制其它事务进行访问，以防止出现错误。
- **位置锁** 这些锁用于确保表的顺序扫描或索引扫描内的一致性。事务通常使用由索引确定的顺序对行进行扫描，或按顺序对行进行扫描。在这两种情况下，都可以将锁放置在扫描位置上。例如，在索引中放置锁可以防止其它事务插入具有特定值或特定值范围的行。

模式锁提供了一种机制，以防止模式更改无意间影响正在执行的事务。行锁、表锁和位置锁每一种都有单独的用途，但是它们相交互。每种锁都能防止一类特定的不一致。根据您选择的隔离级别，数据库服务器将使用这些锁类型中的一些或全部来保持您所要求的一致性等级。

锁持续时间

不同类的锁可保持不同的时间：

- **位置** 短期锁定，如特定行上用于在隔离级别 1 实现游标稳定性的读锁定。
- **事务** 一直保持到事务结束的行锁、表锁和位置锁。
- **连接** 一直保持到事务结束之后的模式锁，如使用 WITH HOLD 游标时创建的模式锁。

获取有关锁的信息

为了诊断数据库中的锁定问题，了解被锁定行的内容可能很有用。您可以使用 sa_locks 系统过程，或使用 Sybase Central 中的 [**表锁**] 选项卡，查看数据库中当前持有的锁。两种方法都会提供您所需的信息，包括持有锁的连接、锁持续时间和锁类型。

注意
由于数据库中锁的瞬时性质，Sybase Central 中可见的行，或由 sa_locks 系统过程返回的行，在查询完成时可能已不再存在。

使用 Sybase Central 查看锁

可以在 Sybase Central 中查看锁。在左窗格中选择数据库，然后单击右窗格中的 [**表锁**] 选项卡。这个选项卡会显示每个锁的连接 ID、用户 ID、表名、锁类型和锁名。

使用 sa_locks 系统过程查看锁

sa_locks 系统过程的结果集中包含 row_identifier 列，该列可标识锁引用的表行。要确定在锁定行中存储的实际值，可以在连接谓语中使用表的 rowID，将 sa_locks 系统过程的结果连接到特定的表。例如：

```
SELECT S.conn_id, S.user_id, S.lock_class, S.lock_type, E.*
  FROM sa_locks() S JOIN Employees E WITH( NOLOCK )
     ON RowId(E) = S.row_identifier
  WHERE S.table_name = 'Employees';
```

注意
可能不必指定 NOLOCK 表提示；但是，如果在除 0 以外的其它隔离级别上发出查询，则该查询可能会一直阻塞，直到锁被释放，这将降低此检查方法的有用性。

另请参见

有关 sa_locks 系统过程的详细信息，请参见“sa_locks 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》。

有关 NOLOCK 表提示的信息，请参见“FROM 子句”一节 《SQL Anywhere 服务器 - SQL 参考》。

有关 ROWID 函数的详细信息，请参见“ROWID 函数 [Miscellaneous]”一节 《SQL Anywhere 服务器 - SQL 参考》。

模式锁

模式锁用于序列化对数据库模式的更改，并用于确保使用表的事务不受由其它连接启动的模式更改的影响。例如，模式锁可防止 ALTER TABLE 语句在表正被另一连接上打开的游标读取时从该表删除列。

由两类模式锁：

- **共享锁** 以共享（读取）模式锁定表模式。

- **独占锁** 将表模式锁定为供单个连接独占使用。

当事务直接或间接引用数据库中的表时，会获取共享模式锁。共享模式锁不会互相冲突；任意数量的事务都可以同时在同一表上获取共享锁。共享模式锁会一直保持到通过 COMMIT 或 ROLLBACK 完成事务为止。

允许持有共享模式锁的任何连接更改表数据，只要更改不与其它连接冲突。

通常使用 DDL 语句修改表的模式时，会获取独占模式锁。ALTER TABLE 语句便是 DDL 语句的一个示例，该语句在对表进行修改之前获取该表的独占锁。任何时候只有一个连接可获取表的独占模式锁—锁定该表模式（共享或独占）的任何其它尝试都将阻塞，或失败并产生错误。这就意味着以隔离级别 0（最低限制的隔离级别）执行的连接将不能从以独占模式锁定的表中读取行。

只有持有独占表模式锁的连接才能够更改表数据。

行锁

行锁可确保在某个事务完成（通过发出隐式或显式 COMMIT 语句提交更改，或通过 ROLLBACK 语句中止更改）前，由该事务修改的任何行无法被其它事务修改，因此，行锁可用于防止更新丢失以及其它类型的事务不一致。

有三类行锁：读（共享）锁定、写（独占）锁定以及意图锁。数据库服务器会为每个事务自动获取这些锁。

读锁定

当事务读取某行时，事务的隔离级别会确定是否获取读锁定。在某一行被放置了读锁定后，任何其它事务都不能在该行上获取写锁定。获取读锁定可确保另一个事务不会修改或删除正在被读取的行。任意数量的事务可以同时在任意一行上获取读锁定，因此读锁定有时也称为共享锁，或非独占锁。

读锁定可保持不同的时间。在隔离级别 2 和 3 上，事务获取的任何读锁定会一直保持到事务通过 COMMIT 或 ROLLBACK 完成为止。这些读锁定称为长期读锁定。

对于以隔离级别 1 执行的事务，数据库服务器在游标所在的行上获取短期读锁定。随着应用程序滚动游标，会释放前面行上的短期读锁定，并在后面的行上获取新的短期读锁定。此技术称作**游标稳定性**。因为应用程序在当前行上保持读锁定，所以在应用程序离开该行之前，其它事务无法对该行进行更改。请注意，如果游标是通过涉及多个表的查询打开的，则可获取多个锁定。仅当必须在请求（通常，这些请求是应用程序发出的 FETCH 语句）间保持游标位置时，才获取短期读锁定。例如，处理 SELECT COUNT(*) 查询时不会获取短期读锁定，因为通过此语句打开的游标将永远不会位于特定的基表行上。这种情况下，数据库服务器仅需要保证读取已提交的语义，即该语句处理的行已被其它事务提交。

以隔离级别 0（读取未提交数据）执行的事务不会获取长期读锁定或短期读锁定，因此也不会与其它事务冲突（除了独占模式锁）。但是，隔离级别 0 事务可能处理其它并发事务所做的未提交的更改。可以使用快照隔离避免处理未提交的更改。请参见“快照隔离”一节第 106 页。

写锁定

事务在插入、更新或删除行时将获取写锁定。这适用于处于所有隔离级别（包括隔离级别 0 和快照隔离级别）的事务。获取了写锁定后，任何其它事务都不能在相同行上获取读锁定、意图锁或写锁定。写锁定也称为独占锁，因为任何时候只能有一个事务可以在某一行上持有独占锁。只要一个事务在一行上持有某种类型的锁定，任何其它事务就不能在同一行上获取写锁定。同样的，一个事务获取了写锁定后，其它事务要求锁定该行的请求都将被拒绝。

意图锁

意图锁（也称为意图更新锁）表示修改特定行的意图。事务进行以下操作时，会获取意图锁：

- 发出 FETCH FOR UPDATE 语句
- 发出 SELECT ...FOR UPDATE BY LOCK 语句
- 将 SQL_CONCUR_LOCK 用作 ODBC 应用程序中的并发基础（使用 SQLSetStmtAttr ODBC API 调用的 SQL_ATTR_CONCURRENCY 参数设置）

意图锁不会与读锁定冲突，因此获取意图锁不会阻塞其它事务读取同一行。但是，意图锁会防止其它事务在同一行上获取意图所或写锁定，这样就保证了在更新前该行无法被任何其它事务所更改。

如果以快照隔离执行的事务请求意图锁，则仅当该行在数据库中未修改且为所有并发事务公用时，才会获取意图锁。但是，如果该行是快照副本，则不会获取意图锁，因为原始行已被其它事务所修改。快照事务要更新该行的任何尝试都将失败，并会返回快照更新冲突错误。

表锁

除了支持行锁，SQL Anywhere 还支持表锁。表锁不同于模式锁：表锁在表的所有行上放置锁，而模式锁在表的模式上放置锁。有三类表锁：

- 共享
- 意图写
- 独占

只有在事务结束（发生 COMMIT 或 ROLLBACK）后才释放表锁。

下表显示了哪两个表锁之间会发生冲突。

	共享	意图	独占
共享		冲突	冲突
意图	冲突		冲突
独占	冲突	冲突	冲突

共享表锁

共享表锁允许多个事务读取基表的数据。拥有针对基表的共享表锁的事务可以修改此表，只要没有其它事务持有对所修改行的任意类型的锁。

例如，通过执行 LOCK TABLE ... IN SHARED MODE 语句可获取共享表锁。REFRESH MATERIALIZED VIEW 和 REFRESH TEXT INDEX 语句还提供了 WITH SHARE MODE 子句，使用该子句可以在刷新操作发生的同时在基础表上创建共享表锁。

另请参见

- “LOCK TABLE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “REFRESH MATERIALIZED VIEW 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “REFRESH TEXT INDEX 语句”一节 《SQL Anywhere 服务器 - SQL 参考》

意图写表锁

意图写表锁（也称作意图表锁）会在事务第一次获取行上的写锁定时隐式获取。与共享表锁一样，意图表锁会一直保持到通过 COMMIT 或 ROLLBACK 完成事务为止。意图表锁与共享表锁和独占表锁冲突，但不与其它意图表锁冲突。

独占锁

独占表锁可以防止其它任何事务访问表进行任何操作（读取、写入、模式修改等等）。一次只能有一个事务在任意表上持有独占锁。独占表锁与所有其它表和行锁冲突。但是，与独占模式锁不同，以隔离级别 0 执行的事务仍可读取其表锁被独占持有的表中的行。

可通过使用 LOCK TABLE ... IN EXCLUSIVE MODE 语句显式获取独占表锁。REFRESH MATERIALIZED VIEW 和 REFRESH TEXT INDEX 语句还提供了 WITH EXCLUSIVE MODE 子句，使用该子句可以在刷新操作发生的同时在基础表上创建独占表锁。

另请参见

- “LOCK TABLE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “REFRESH MATERIALIZED VIEW 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “REFRESH TEXT INDEX 语句”一节 《SQL Anywhere 服务器 - SQL 参考》

位置锁

除了行锁，SQL Anywhere 还会实现一种键范围的锁定方式，旨在防止由于幻像（或称为幻像行）的存在而导致的异常。仅当数据库服务器处理以隔离级别 3 运行的事务时，才涉及到位置锁。

我们称以隔离级别 3 运行的事务是可序列化的。这就意味着隔离级别 3 上事务的行为不应受其它事务的并发更新活动影响。特别是，在隔离级别 3 上，事务不能受引入可影响计算结果的行的 INSERT 或 UPDATE（幻像）所影响。SQL Anywhere 使用位置锁防止此类更新发生。正是通过此附加锁定来区分隔离级别 2（可重复读取）和隔离级别 3。

为了防止创建幻像行，SQL Anywhere 在表的物理扫描中获取位置锁。如果是顺序扫描，则扫描位置基于当前行的行标识符。如果是索引扫描，则扫描的位置基于当前行的索引键值（该值可以是唯一的也可以是不唯一的）。通过锁定扫描位置，一个事务可以防止其它事务按照行的顺序执行与特定值范围相关的插入。这包括可更改索引属性值的 INSERT 语句和 UPDATE 语句。扫描位置锁定后，UPDATE 语句可视为请求对索引条目执行 DELETE，随后紧跟一个 INSERT 请求。

SQL Anywhere 支持两类位置锁：幻像锁和防幻像锁。这两类锁都是共享锁，因为任意数量的事务可以在同一行上获取相同类型的锁。但是，幻像锁和防幻像锁互相冲突。

幻像锁

幻像锁（有时也称为防插入锁）放置在扫描位置上以防止其它事务随后创建幻像行。获取了幻像锁后，它可以防止其它事务在表中被放置了防插入锁的行与前一行之间插入新行。幻像锁是长期锁，它会一直保持到事务结束。

只有以隔离级别 3 运行的事务才能获取幻像锁；只有该隔离级别可保证幻像的一致性。

对于索引扫描，通过索引读取的每行上都会获取幻像锁，且会在索引扫描的末尾获取一个额外的幻像锁，以防止在满足条件的索引范围的末尾处向索引插入行。索引扫描上的幻像锁可防止向表中插入新行时创建幻像，也可防止会导致在幻像锁所在位置创建索引条目的索引值的更新。

对于顺序扫描，表中的每行上都会获取幻像锁，以防止任何插入操作变更结果集。因此，隔离级别 3 扫描通常会对数据库并发产生负面影响。虽然一个或多个幻像锁会与插入锁冲突，而一个或多个读锁定会与写锁定冲突，但是幻像锁/插入锁和读锁定/写锁定之间不存在交互。例如，虽然不能在包含读锁定的行上获取写锁定，但可以在只具有幻像锁的行上获取写锁定。这种灵活的安排为数据库服务器提供了更多选择，但这也意味着服务器必须经常注意在获取幻像锁的同时也应获取读锁定。否则，其它事务可以删除该行。

插入锁

插入锁（有时也被称作防幻像锁）是短期锁，放置在扫描位置上以保留插入行的权利。插入锁仅在插入期间保持；该行正确插入数据库页后，会立即在该行上放置写锁定以确保一致性，并会释放插入锁。一个事务在某一行上获取了插入锁后，其它事务都不能在同一行上获取幻像锁。插入锁是必需的，因为服务器必须预期任何活动连接上的隔离级别 3 扫描操作，而任何新的请求都可能引发隔离级别 3 扫描。请注意，当幻像锁和插入锁由同一事务持有时，它们之间不会互相冲突。

锁定冲突

SQL Anywhere 根据需要使用模式锁、行锁、表锁和位置锁来确保您所要求的一致性级别。您不需要显式请求使用某个特定的锁定。而应通过选择最符合您的要求的隔离级别来控制所维护的一致性级别。了解各种锁类型有助于您选择隔离级别和理解各个级别对性能的影响。请记住，任何一个事务都不会由于获取锁定阻塞其自身；锁定冲突仅在两个（或多个）事务之间发生。

哪些锁会发生冲突？

虽然四类锁定各具有其特定用途，但所有类型的锁定交互，并因此可能导致事务之间的锁定冲突。为了确保数据库一致性，一次只能有一个事务更改任意一行。否则，两个同时执行的事务可能会尝试将一个值更改为两个不同的新值。因此，行写锁定应该是独占的，这一点非常重要。相反，如果多个事务想要读取一行，则不会出现任何问题。因为所有事务都不会更改该行，所以也不会发生冲突。因此，行读锁定可以在多个连接间共享。

下表显示了哪两种锁之间会发生冲突。模式锁未包括在内，因为它们不适用于行。

	读 (R)	意图 (R)	写 (R)	共享 (T)	意图 (T)	独占 (T)	幻像 (P)	插入 (P)
读 (R)			冲突			冲突		
意图 (R)		冲突	冲突			冲突		
写 (R)	冲突	冲突	冲突	冲突		冲突		
共享 (T)			冲突		冲突	冲突		
意图 (T)				冲突		冲突		
独占 (T)	冲突	冲突	冲突	冲突	冲突	冲突	冲突	冲突
幻像 (P)						冲突		冲突
插入 (P)						冲突	冲突	

查询过程中的锁定

SQL Anywhere 在用户输入 SELECT 语句时使用的锁定取决于该事务的隔离级别。无论隔离级别为何，所有 SELECT 语句都会在被引用表上获取模式锁。

以隔离级别 0 执行的 SELECT 语句

以隔离级别 0 执行 SELECT 语句时，不需要任何锁定操作。数据库服务器不对各个事务进行保护以防止其它事务更改数据。程序员或数据库用户在解读这些查询的结果时应考虑到这个限制。

以隔离级别 1 执行的 SELECT 语句

SQL Anywhere 在以隔离级别 1 运行事务时使用的锁不比以隔离级别 0 运行时的多很多。数据库服务器只以两种方式修改其操作。

操作中的第一个区别与获取锁定无关，而与是否遵从这些锁定有关。处于隔离级别 0 时，事务可以读取任意行，即使其它事务已在该行获取了写锁定。相反，处于隔离级别 1 的事务在读取每一行之前必须检查该行是否已被放置了写锁定。它不能读取任何被放置了写锁定的行，因为这样做可能会导致读取脏数据。使用 READPAST 提示允许服务器忽略被放置了写锁定的行，但当事务不再阻塞时，其语义不再与隔离级别 1 的语义相符。请参见“FROM 子句”一节 《SQL Anywhere 服务器 - SQL 参考》中的 READPAST 提示。

操作中的第二个区别影响游标稳定性。游标稳定性是通过在游标的当前行上获取一个短期读锁定来实现的。移动游标时，这个读锁定将被释放。如果游标中包含连接的结果，则可能会影响多行。在这种情况下，数据库服务器将在所有为游标的当前行提供信息的行上获取短期读锁定，并在选择游标的另一行作为当前行后释放这些锁定。

以隔离级别 2 执行的 SELECT 语句

在隔离级别 2，数据库服务器会修改其操作以确保可重复读取语义。如果 SELECT 语句从表中的每一行返回值，那么，数据库服务器将在读取表中的每一行时在该行上获取一个读锁定。如果 SELECT 语句包含 WHERE 子句，或其它会对结果中的行加以限制的条件，那么，数据库服务器将读取每一行，测试该行中的值是否符合条件，如果符合，则在该行上获取一个读锁定。所获取的读锁定为长期读锁定，且会一直保持到通过隐式或显式 COMMIT 或 ROLLBACK 语句完成事务为止。与隔离级别 1 相同，隔离级别 2 也确保游标稳定性，且不允许脏读。

以隔离级别 3 执行的 SELECT 语句

以隔离级别 3 工作时，数据库服务器必须确保所有事务调度均可序列化。特别是，除达到隔离级别 2 的要求外，它必须防止幻像行，以保证在所有情况下重新执行同一语句都返回相同的结果。

为达到这个要求，数据库服务器使用了读锁定和幻像锁。以隔离级别 3 执行 SELECT 语句时，数据库服务器会在结果集计算期间处理的每一行上获取读锁定。这样做可确保该事务完成前，其它事务无法修改这些行。

此要求与数据库服务器以隔离级别 2 执行的操作相似，但区别在于必须为每个读取的行获取锁定，无论这些行是否满足 SELECT 语句的 WHERE、ON 或 HAVING 子句中的任何谓语。例如，如果您要选择销售部门中所有雇员的名字，无论以隔离级别 2 还是隔离级别 3 执行该事务，服务器都必须锁定所有包含销售人员信息的行。但是，以隔离级别 3 执行时，服务器还必须在每个不属于销售部门的雇员的相应行上获取读锁定。否则，另一个事务可以在第一个事务仍在执行时将另一个雇员转到销售部门。

当必须为每个读取的行获取读锁定时有两种含义：

- 数据库服务器需要放置的锁数目可能要比隔离级别 2 所需的锁数目要多得多。获取的幻像锁的数目比为扫描而获取的读锁定的数目多一。此加倍的锁定开销增加了请求的执行时间。
- 为每个读取的行获取读锁定会对同一表的数据库更新操作的并发性造成负面影响。

数据库服务器获取的幻像锁的数目可能会有很大不同，这取决于查询优化程序所选择的执行策略。由于隔离级别 3 可能会对整体系统并发性产生负面影响，SQL Anywhere 查询优化程序将尝试避免以隔离级别 3 进行顺序扫描，但优化程序进行此操作的能力取决于语句中谓语和被引用表上可用的相关索引。

例如，假设您要选择雇员 ID 为 123 的雇员的相关信息。由于 EmployeeID 是雇员表的主键，因此查询优化程序会几乎毫无疑问地选择索引策略（使用主键索引）来高效定位行。此外，其它事务也不可能将另一个雇员的 ID 更改为 123，因为主键值必须是唯一的。服务器只需在包含雇员 123 相关信息的行上获取一个读锁定，即可确保不会将该 ID 编号分配给其他雇员。

相反，如果您要选择销售部门中的所有雇员，数据库服务器则需要获取更多锁定。如果缺少相关索引，则数据库服务器必须读取雇员表中的每一行，并测试每个雇员是否属于销售部门。如果属于这种情况，则必须为表中的每一行获取读锁定和幻像锁。

SELECT 语句和快照隔离

以快照、语句快照或只读语句快照隔离级别执行的 SELECT 语句不获取读锁定。这是因为每个快照事务（或语句）会看到以前某一时间点的数据库已提交状态的快照。特定的时间点由语句使用三个快照隔离级别中的哪个级别确定。读取事务从不阻塞更新事务，同样更新事务也从不阻塞读取程序。因此，快照隔离除了具有明显的一致性优势之外，还具有相当可观的并发性优势。但快照隔离也有一个缺点，可能花费很大的系统开销。这是因为快照隔离要保证一致性，必须为其它并发事务保存、跟踪并（最终）删除被更改的行的副本。

插入过程中的锁定

INSERT 操作将创建新行。SQL Anywhere 在插入过程中利用多种类型的锁定以确保数据的完整性。对于以任何隔离级别执行的 INSERT 语句，都将发生以下一系列操作。

1. 在表上获取共享模式锁（如果尚未持有）。
2. 在表上获取意图写表锁（如果尚未持有）。
3. 在页中找到未被锁定的位置以存储新行。为了减少锁定争用，服务器不会立即重新使用通过删除（但尚未被提交）行而变得可用的空间。可能会将新的页分配给表（且数据库文件可能会增大）以容纳新的行。
4. 将所提供的任何值填入新行。
5. 在要添加该行的表中放置一个插入锁。上文中已经讲过插入锁是独占的，因此，获取了该插入锁后，任何其它隔离级别 3 的事务都不能通过获取幻像锁来阻塞这个插入操作。
6. 在新行上放置写锁定。获取写锁定后，会释放插入锁。
7. 将行插入表中。现在，处于隔离级别 0 的其它事务第一次可以看到这一新行存在。然而，由于之前获取了写锁定，这些其它事务无法修改或删除新行。
8. 更新所有受影响的索引并校验唯一性（如果适当）。主键值必须是唯一的。也可以将其它列定义为只包含唯一的值。如果这样的列存在，则需要校验唯一性。
9. 如果该表是外表，则在主表上获取共享模式锁（如果尚未持有），并在主表中相匹配的主行上获取读锁定（如果插入的外键列值非 NULL）。数据库服务器必须确保主行在插入事务 COMMIT 后仍存在。方法是，在主行上获取一个读锁定。放置了这个读锁定后，任何其它事务仍可以读取该行，但都不能删除或更新该行。

   如果相应的主行不存在，则给出参照完整性约束违规。

最后一步之后，在该表上定义的任何 AFTER INSERT 触发器都可能触发。触发器中的处理遵循与应用程序相同的锁定行为。提交（假设满足所有的参照完整性约束）或者回退了事务后，会释放所有长期锁定。

唯一性

可以确保特定的一列或列组合中的所有值都是唯一的。数据库始终通过为唯一列构建索引（即使您没有显式创建）来执行此任务。

特别是，所有主键值都必须是唯一的。数据库服务器自动为每个表的主键构建索引。不要请求数据库服务器在主键上创建索引，因为这样做是多余的。

遗孤和参照完整性

通常，一个外键会参照另一个表中的一个主键或 UNIQUE 约束。如果该主键不存在，不符合要求的外键被称作**遗孤**。SQL Anywhere 会自动确保您的数据库中不包含任何遗孤。这个过程被称作**校验参照完整性**。数据库服务器通过计算遗孤数来校验参照完整性。

wait_for_commit

可以指示数据库服务器将参照完整性校验延迟到事务结束时。利用这种方式，可以插入包含外键的一行，然后再插入包含丢失的主键的主行。这两个操作必须在同一事务中发生。

要请求数据库服务器将参照完整性检查延迟到提交时，请将 wait_for_commit 选项的值设置为 On。缺省情况下，此选项被设置为 Off。要启用该选项，请发出以下命令：

```
SET OPTION wait_for_commit = On;
```

如果插入新外键值时服务器未找到匹配的主行，且 wait_for_commit 为 On，则服务器允许该行以遗孤的形式插入。对于孤立的外行，插入时会发生以下一系列步骤：

- 服务器在主表上获取一个共享模式锁（如果尚未持有）。服务器还会在主表上获取一个意图写锁。
- 服务器将代理行插入到主表中。并未向主表插入实际的行，但出于锁定目的，服务器会为该行生成一个唯一行标识符，并在此代理行上获取写锁定。随后，服务器会向主表的主键索引插入适当的值。

提交事务之前，数据库服务器将通过检查您的事务产生的遗孤数来校验是否保持了参照完整性。在每个事务结束时，遗孤数必须为零。

更新过程中的锁定

数据库服务器使用以下过程修改一条特定记录中包含的信息。与插入相同，所有事务都将执行以下一系列的操作，而无论其隔离级别为何。

1. 在表上获取共享模式锁（如果尚未持有）。
2. 在表上获取意图写表锁（如果尚未持有）。
   a. 标识要更新的候选行。当扫描行时，将它们锁定。缺省锁定行为如“隔离级别和一致性”一节第 105 页所述。

      在隔离级别 2 和 3，会出现与缺省锁定行为不同的以下差异：将获取意图写行级锁定而非读锁定，并且在某些情况下可能会在最终被拒绝作为更新候选行的行上获取意图写锁定。
   b. 对于在步骤 2.a 中标识的每个候选行，遵循其余的顺序。
3. 在受影响的行上放置写锁定。
4. 按照每个 UPDATE 语句，更新每个受影响的列值。

5. 如果索引值被更改，请添加新索引条目。会保留行的原始索引条目，但会将其标记为已删除。持有长期插入锁时插入新值的新索引条目。服务器将校验索引唯一性（如果适当）。
6. 如果对行中的任何外键值进行了变更，则会在主表上获取一个共享模式锁，并按照“插入过程中的锁定”一节第 130 页中概述的插入新外键值的过程进行操作。同样，按照 WAIT_FOR_COMMIT 的过程进行操作（如果适用）。
7. 如果该表是参照完整性关系中的主表，且关系的 UPDATE 操作不是 RESTRICT，则确定外表中受影响的行（通过以下操作：首先在外表上获取共享模式锁，然后在每个表上获取意图写表锁），然后在所有受影响的行上获取写锁定，并根据需要修改各行。请注意，此过程可能分多级进行参照完整性约束的嵌套层次。

最后一步之后，任何 AFTER UPDATE 触发器都可能触发。COMMIT 后，服务器将通过确保此事务生成的遗孤数为 0 来验证参照完整性，并释放所有的锁。

修改列值可能需要执行大量操作。如果您要修改的列不属于主键或外键，数据库服务器需要完成的工作量会少得多。如果要修改的列不包含在索引中（此索引或显式创建，或因为您已将该列声明为唯一的而隐式创建），需要完成的工作量还会更少。

UPDATE 操作过程中的参照完整性校验操作并不比 INSERT 过程中的校验操作简单。实际上，更改主键的值时，就可能会产生遗孤。插入替换值时，数据库服务器必须再次检查是否产生了遗孤。

删除过程中的锁定

DELETE 操作的步骤与 INSERT 操作的步骤几乎相同，只是顺序相反。与插入和更新相同，所有事务都将执行以下一系列的操作，而无论其隔离级别为何。

1. 在表上获取共享模式锁（如果尚未持有）。
2. 在表上获取意图写表锁（如果尚未持有）。
   a. 标识要更新的候选行。当扫描行时，将它们锁定。缺省锁定行为如“隔离级别和一致性”一节第 105 页所述。

      在隔离级别 2 和 3，会出现与缺省锁定行为不同的以下差异：将获取意图写行级锁定而非读锁定，并且在某些情况下可能会在最终被拒绝作为更新候选行的行上获取意图写锁定。

   b. 对于在步骤 2.a 中标识的每个候选行，遵循其余的顺序。
3. 在要删除的行上放置写锁定。
4. 从表中删除该行，这样其它事务就无法再看到该行。在提交该事务之前不能销毁该行，因为这样做之后将无法再回退该事务。会保留已删除行的索引条目，但会将其标记为已删除，直到事务完成。这可防止其它事务重新插入相同的行。
5. 如果该表是参照完整性关系中的主表，且关系的 DELETE 操作不是 RESTRICT，则确定外表中受影响的行（通过以下操作：首先在外表上获取共享模式锁，然后在每个表上获取意图写表锁），然后在所有受影响的行上获取写锁定，并根据需要修改各行。请注意，此过程可能分多级进行参照完整性约束的嵌套层次。

只要不违反参照完整性，就可以提交该事务。为校验参照完整性，数据库服务器还会跟踪该删除操作产生的遗孤。COMMIT 后，服务器会将该操作记录在事务日志文件中并释放所有的锁。

较早释放读锁定

处于隔离级别 3 时，事务在它读取的每一行上获取一个读锁定。通常，事务从不在其结束之前释放锁定。实际上，如果要使调度可序列化，事务就不能过早释放锁定。

SQL Anywhere 总是将写锁定一直保留到事务完成时。这可以防止其它事务修改该行，以免无法回退第一个事务。

只有在一种情况下才会释放读锁定：处于隔离级别 1 时，事务仅在某一行成为游标的当前行时，才在该行上获取读锁定。但是，处于隔离级别 1 时，当该行不再是当前行后，锁定将被释放。这一行为是可以接受的，因为数据库服务器在隔离级别 1 上不需要确保可重复的读取。

有关隔离级别的详细信息，请参见“选择隔离级别”一节第 134 页。

选择隔离级别

选择哪个隔离级别取决于应用程序执行哪类任务。本节提供了一些有关选择隔离级别的指导。

要选择适当的隔离级别，必须在对一致性和准确性的需求与不受阻碍地执行并发事务的需求之间找到平衡点。如果一个事务只涉及一个表中的一两个特定值，而另一个事务搜索许多大型表并因此可能需要锁定许多行或整个表，且可能需要很长时间才能完成，那么前者对其它进程的干扰通常要比后者少。

例如，如果您的事务涉及在银行帐户间转帐，则必须尽可能地确保返回的信息正确。相反，如果只需要粗略地估计不活动帐户所占的比例，则可能不会介意该事务是否等待其它事务，而且可能愿意牺牲一些准确性，以避免干扰该数据库的其他用户。

此外，转帐可能只会影响包含两个帐户余额的那两行，而要计算不活动帐户的比例则必须读取所有帐户。因此，转帐延迟其它事务的可能性更小。

SQL Anywhere 提供了四个隔离级别：级别 0、1、2 和 3。级别 3 提供完全隔离，并确保以可序列化调度的方式交错执行事务。

如果已为数据库启用了快照隔离，则可以使用三个附加隔离级别：快照、语句快照和只读语句快照。

选择快照隔离级别

快照隔离提供并发和一致性优点。使用快照隔离会增加系统开销，因为正在运行的事务可能需要的旧行版本会一直保存。所以，长时间运行的快照可能需要存储大量旧行版本。通常，用于快照的快照要比用于语句快照的快照持续更长时间。因此，与快照相比，语句快照可能占用更少的存储空间，但一致性较差（事务中的每个语句看到的是不同时间点的数据库）。

有关使用快照隔离对性能影响的详细信息，请参见“游标敏感性和隔离级别”一节 《SQL Anywhere 服务器 - 编程》。

大多数情况下，建议使用快照隔离级别，因为它为整个事务提供数据库的单一视图。

语句快照隔离级别提供的一致性较差，但是在由于长时间运行事务而导致临时文件中版本存储所用的空间过大的情况下，它将非常有用。

只读语句快照隔离级别提供的一致性要比语句快照差，但可避免出现更新冲突的可能性。因此，它最适用于移植那些最初打算在不同隔离级别下运行的应用程序。

有关快照隔离的详细信息，请参见“快照隔离”一节第 106 页。

可序列化调度

要并发处理事务，数据库服务器必须执行一个事务中的部分语句，然后执行其它事务的一些语句，然后再继续处理第一个事务中后面的操作。交错执行各个事务中部分操作的顺序称作**调度**。

以这种方式并发地应用多个事务可能会产生多种结果，包括上文中介绍的三种特殊的不一致情况。有时，也可能达到数据库的最终状态，就象依次执行各个事务时那样，意思是说，始终在一个事务全部处理完成之后再开始处理下一个事务。只要在按某个顺序依次执行事务时，数据库达到的状态与实际的调度相同，那么这个调度就叫做**可序列化**调度。

可序列化性是判断正确性的公认标准。我们认为可序列化调度是正确的，因为数据库没有因并发执行事务而受到影响。

隔离级别会影响事务的可序列化性。在隔离级别 3 上，所有调度都是可序列化的。缺省设置为 0。

可序列化意味着并发不会产生附加影响

即使在依次执行事务时，数据库的最终状态也取决于执行这些事务的顺序。例如，如果一个事务将某个特定单元设置为值 5，而另一个事务将其设置为数字 6，那么该单元的最终值将由最后执行那个事务确定。

了解调度可序列化并不能确定按哪个顺序执行事务最好，而只能说明并发不会产生附加影响。按某种顺序依次执行一组事务得到的所有结果都被认为是正确的。

不可序列化的调度将会造成不一致

如果调度不可序列化，将出现多种不一致问题。“典型的不一致类型”一节第 112 页中介绍的不一致就是其中的典型问题。在每种情况下，出现不一致的原因都在于交错执行语句的方式；如果所有事务依次执行，不可能产生这种结果。例如，只有在这种情况下才会发生脏读：一个事务可以选择某些行而同时另一个事务正在同一行中插入数据或更新数据时。

各个隔离级别的典型事务

各个隔离级别分别适用于特定类型的任务。以下信息可以帮助您确定各个特定操作最适合使用哪个级别。

级别 0 的典型事务

涉及浏览或输入数据的事务可能需要几分钟时间才能完成，并且需要读取大量行。如果使用隔离级别 2 或 3，可能会降低并发性。对于这类事务，通常使用隔离级别 0 或 1。

例如，对于需要从数据库中读取大量信息以生成统计摘要的决策支持应用程序，即使读取到的几行数据以后被修改，可能也不会对其产生很大影响。如果要求这样的应用程序使用较高的隔离级别，它可能会在大量数据上获取读锁定，从而导致其它应用程序无法写入数据。

级别 1 的典型事务

隔离级别 1 适合与游标一起使用，因为这种组合既可以确保游标稳定性，又不会大幅提高锁定需求。SQL Anywhere 通过较早释放在游标的当前行上获取的读锁定来实现这一优点。在级别 2 和级别 3 上，这些锁定必须持续到事务结束以确保可重复的读取。

例如，通过游标更新存货量的事务适合使用这个级别，因为随着进货或卖出对存货量进行的每次调整都不会丢失，而这些频繁调整对其它事务的影响又可以降低到最小程度。

级别 2 的典型事务

处于隔离级别 2 时，其它事务不能更改符合您的条件的行。如果必须多次读取行并且需要第一个结果集中包含的行不会更改，则可以使用这个级别。

由于需要相对较大数量的读锁定，因此使用这个隔离级别时应谨慎。与级别 3 的事务一样，仔细设计数据库和索引可以减少获取的锁数目，从而提高数据库的性能。

级别 3 的典型事务

隔离级别 3 适合于将安全性放在首位的事务。由于消除了幻像行，因此可以对行集放心地执行多步操作，而不必担心在操作过程中因出现新行导致结果损坏。

虽然隔离级别 3 可以提供很高的完整性，但在需要支持大量并发事务的大型系统中应谨慎使用。SQL Anywhere 在这一级别上放置的锁数目比任何其它级别都多，因此也增大了一个事务阻碍众多其它事务进程的可能性。

提高隔离级别 2 和 3 的并发性

隔离级别 2 和 3 使用大量锁定，因此，对于经常使用这些隔离级别的数据库而言，优良的设计非常重要。如果必须使用可序列化事务，设计数据库时，特别是在设计索引时，应牢记项目的业务规则，这一点非常重要。还可以将大事务分割成几个较小的事务，从而缩短锁定行的时间，这样便可提高性能。

虽然可序列化事务最有可能阻塞其它事务，但其效率并不一定低。处理这些事务时，SQL Anywhere 可以执行某些优化。虽然增加了锁数目，但这些优化可能仍会使性能得到提高。例如，由于无论是否符合搜索条件，所有读取的行都必须被锁定，因此数据库服务器可以自由地合并读取行的操作和放置锁的操作。

减小锁定的影响

放置大量的锁可能会影响其它并发事务的执行，为防止这种情况，建议避免以隔离级别 3 运行事务。

如果某个操作的性质要求它必须以隔离级别 3 运行，则可以通过将查询设计为读取尽量少的行和索引条目来减少该操作对并发性的影响。这些步骤有助于提高级别 3 事务的运行速度，而且，还有助于减少放置的锁数目，这一点可能更为重要。

当至少一个操作以隔离级别 3 执行时，您可能会发现添加索引可提高事务的速度。索引有两个优点：

- 使用索引可以更有效地对行进行定位
- 利用索引的搜索操作需要的锁数目可能比较少。

有关 SQL Anywhere 使用的锁定方法的详细信息，请参见“锁定的工作方式”一节第 122 页。

有关性能和 SQL Anywhere 如何计划其信息访问以执行命令的详细信息，请参见“提高数据库性能”第 159 页。

隔离级别教程

不同隔离级别的行为有很大不同，应使用哪个级别取决于您的数据库和要执行的操作。以下一组教程可以帮助您确定不同的任务适合使用哪个隔离级别。

教程：脏读

以下教程举例说明并发执行多个事务时可能会出现的一种不一致情况。一家小商品销售公司的两位雇员同时访问公司的数据库。第一个人是公司的销售经理。第二个人是会计。

销售经理想要将公司销售的 T 恤衫的价格提高 $0.95，但在 SQL 语言的语法方面有点儿小问题。同时，会计正要计算当前存货的零售额，以便将它包含到他自愿在下次召开管理会议时提交的报告中。但销售经理却不知道这个情况。

提示
对数据库进行以下变更之前，为谨慎起见，建议使用 SELECT 代替 UPDATE 对更改进行测试。

注意
要按照此教程正常工作，则一定不要选择 Interactive SQL 中的 [**自动释放数据库锁**] 选项（[**工具**] » [**选项**] » [**SQL Anywhere**]）。

在本示例中，假设两个雇员的角色，且他们并发使用 SQL Anywhere 示例数据库。

1. 启动 Interactive SQL。
2. 在 [**连接**] 窗口中，以销售经理的身份连接到 SQL Anywhere 示例数据库：
   - 在 [**ODBC 数据源名称**] 字段中，选择 [**SQL Anywhere 11 Demo**]。
   - 单击 [**高级**] 选项卡，在 [**ConnectionName**] 字段中键入 **Sales Manager**。
   - 单击 [**确定**]。
3. 再启动一个 Interactive SQL 的实例。
4. 在 [**连接**] 窗口中，以会计的身份连接到 SQL Anywhere 示例数据库：
   - 在 [**ODBC 数据源名称**] 字段中，选择 [**SQL Anywhere 11 Demo**]。
   - 单击 [**高级**] 选项卡，在 [**ConnectionName**] 字段中键入 **Accountant**。
   - 单击 [**确定**]。
5. 作为销售经理，将所有 T 恤衫的价格提高 $0.95：
   - 在 [**Sales Manager**] 窗口中，执行以下命令：

```
UPDATE Products
   SET UnitPrice = UnitPrice + 95
   WHERE Name = 'Tee Shirt';
SELECT ID, Name, UnitPrice
   FROM Products;
```

结果为：

ID	name	UnitPrice
300	Tee Shirt	104.00
301	Tee Shirt	109.00
302	Tee Shirt	109.00
400	Baseball Cap	9.00
...	...	...

您很快就发现应输入 0.95 而不是 95，但在还未能改正错误之前，会计在另一个办公室访问了数据库。

6. 公司的会计担心存货会占用太多资金。作为会计，执行以下命令计算所有库存商品的总零售额：

```
SELECT SUM( Quantity * UnitPrice )
 AS Inventory
   FROM Products;
```

结果为：

Inventory
21453.00

可惜的是，计算出的结果并不准确。由于销售经理不小心将 T 恤衫的价格提高了 $95，而结果是用错误的价格计算得到的。这个错误演示了一种被称作**脏读**的典型不一致类型。作为会计，您访问了销售经理已经输入但尚未提交的数据。

您可以清除脏读和“隔离级别和一致性”一节第 105 页中介绍的其它不一致。

7. 作为销售经理，回退第一次所做的更改，然后输入正确的 UPDATE 命令，即可改正错误。检查新值是否正确。

```
ROLLBACK;
UPDATE Products
SET UnitPrice = UnitPrice + 0.95
WHERE NAME = 'Tee Shirt';
COMMIT;
```

ID	name	UnitPrice
300	Tee Shirt	9.95
301	Tee Shirt	14.95
302	Tee Shirt	14.95

ID	name	UnitPrice
400	Baseball Cap	9.00
...	...	...

8. 会计并不知道他计算出的金额不正确。您可以在会计的窗口中再次执行 SELECT 语句查看正确的值。

```
SELECT SUM( Quantity * UnitPrice )
 AS Inventory
   FROM Products;
```

Inventory
6687.15

9. 在销售经理的窗口中完成事务。销售经理应输入 COMMIT 语句使更改成为永久更改，但您应该执行 ROLLBACK，以避免更改 SQL Anywhere 示例数据库的本地副本。

```
ROLLBACK;
```

由于数据库服务器并发处理销售经理的工作和会计的工作，因此会计在不知情的情况下从数据库中得到了错误的信息。

使用快照隔离避免脏读

使用快照隔离时，其它数据库连接只能看到响应其查询的已提交数据。将隔离级别设置为语句快照或快照，会防止发生脏读的可能性。会计可以使用快照隔离以确保执行查询时只能看到已提交的数据。

1. 启动 Interactive SQL。
2. 在 [**连接**] 窗口中，以销售经理的身份连接到 SQL Anywhere 示例数据库：
   - 在 [**ODBC 数据源名称**] 字段中，选择 [**SQL Anywhere 11 Demo**]。
   - 单击 [**高级**] 选项卡，在 [**ConnectionName**] 字段中键入 **Sales Manager**。
   - 单击 [**确定**] 进行连接。
3. 执行以下语句为数据库启用快照隔离：

```
SET OPTION PUBLIC.allow_snapshot_isolation = 'ON';
```

4. 再启动一个 Interactive SQL 的实例。
5. 在 [**连接**] 窗口中，以会计的身份连接到 SQL Anywhere 示例数据库：
   - 在 [**ODBC 数据源名称**] 字段中，选择 [**SQL Anywhere 11 Demo**]。
   - 单击 [**高级**] 选项卡，在 [**ConnectionName**] 字段中键入 **Accountant**。
   - 单击 [**确定**]。
6. 作为销售经理，将所有 T 恤衫的价格提高 $0.95：

- 在标题为 Sales Manager 的窗口中，执行以下命令：

```
UPDATE Products
SET UnitPrice = UnitPrice + 0.95
WHERE Name = 'Tee Shirt';
```

- 使用销售经理制定的新 T 恤衫价格，计算所有库存商品的总零售额：

```
SELECT SUM( Quantity * UnitPrice )
 AS Inventory
   FROM Products;
```

结果为：

Inventory
6687.15

7. 作为会计，执行以下命令计算所有库存商品的总零售额：由于此事务使用快照隔离级别，因此仅计算已提交到数据库的数据的结果。

```
SET OPTION isolation_level = 'Snapshot';
SELECT SUM( Quantity * UnitPrice )
 AS Inventory
   FROM Products;
```

结果为：

Inventory
6538.00

8. 作为销售经理，执行以下语句将您的更改提交到数据库：

```
COMMIT;
```

9. 作为会计，执行以下语句查看当前存货的更新后的零售额：

```
COMMIT;
SELECT SUM( Quantity * UnitPrice )
 AS Inventory
   FROM Products;
```

结果为：

Inventory
6687.15

由于用于会计的事务的快照以第一次读取操作开始，因此您必须执行 COMMIT 以结束事务，并允许会计在快照事务开始后查看对数据所做的更改。请参见“了解快照事务”一节第 108 页。

10. 作为销售经理，执行以下语句撤销对 T 恤衫价格的更改并将 SQL Anywhere 示例数据库恢复到其原始状态：

```
UPDATE Products
SET UnitPrice = UnitPrice - 0.95
```



```
WHERE Name = 'Tee Shirt';
COMMIT;
```

教程：非可重复读取

“教程：脏读”一节第 137 页中的示例演示了第一类不一致（即脏读）。在该示例中，会计对零售额进行了计算，而销售经理正在更新一个价格。会计的计算使用了销售经理已经输入但正在进行改正的错误信息。

以下示例演示了另一类不一致：非可重复读取。在本示例中，假设两个相同的雇员角色，且他们都并发使用 SQL Anywhere 示例数据库。销售经理想要给塑料太阳帽定一个新的销售价格。会计想要核实某些出现在最近一份订单上的商品的价格。

本例开始时两个连接都处于隔离级别 1，而不是隔离级别 0（随 SQL Anywhere 提供的 SQL Anywhere 示例数据库的缺省隔离级别）。将隔离级别设置为 1 可以消除上一教程中演示的不一致类型（即脏读）。

注意
要按照此教程正常工作，则一定不要选择 Interactive SQL 中的 [**自动释放数据库锁**] 选项（[**工具**] » [**选项**] » [**SQL Anywhere**]）。

1. 启动 Interactive SQL。
2. 在 [**连接**] 窗口中，以销售经理的身份连接到 SQL Anywhere 示例数据库：
   - 在 [**ODBC 数据源名称**] 字段中，选择 [**SQL Anywhere 11 Demo**]。
   - 单击 [**高级**] 选项卡，在 [**ConnectionName**] 字段中键入 **Sales Manager**。
   - 单击 [**确定**]。
3. 再启动一个 Interactive SQL 的实例。
4. 在 [**连接**] 窗口中，以会计的身份连接到 SQL Anywhere 示例数据库：
   - 在 [**ODBC 数据源名称**] 字段中，选择 [**SQL Anywhere 11 Demo**]。
   - 单击 [**高级**] 选项卡，在 [**ConnectionName**] 字段中键入 **Accountant**。
   - 单击 [**确定**]。
5. 执行以下命令，将会计的连接的隔离级别设置为 1。

   ```
   SET TEMPORARY OPTION isolation_level = 1;
   ```
6. 在销售经理的窗口中执行以下命令，将隔离级别设置为 1：

   ```
   SET TEMPORARY OPTION isolation_level = 1;
   ```
7. 会计决定列出太阳帽的价格。作为会计，执行以下命令：

   ```
   SELECT ID, Name, UnitPrice FROM Products;
   ```

ID	Name	UnitPrice
300	Tee Shirt	9.00
301	Tee Shirt	14.00
302	Tee Shirt	14.00
400	Baseball Cap	9.00
401	Baseball Cap	10.00
500	Visor	7.00
501	Visor	7.00
...	...	...

8. 销售经理决定为塑料太阳帽定一个新的销售价格。作为销售经理，执行以下命令：

```
SELECT ID, Name, UnitPrice FROM Products
WHERE Name = 'Visor';
UPDATE Products
SET UnitPrice = 5.95 WHERE ID = 501;
COMMIT;
SELECT ID, Name, UnitPrice FROM Products
WHERE Name = 'Visor';
```

ID	Name	UnitPrice
500	Visor	7.00
501	Visor	5.95

9. 比较 [Sales Manager] 窗口中太阳帽的价格和 [Accountant] 窗口中相同太阳帽的价格。会计再次执行 SELECT 语句，并将看到销售经理的新销售价格。

```
SELECT ID, Name, UnitPrice
FROM Products;
```

ID	Name	UnitPrice
300	Tee Shirt	9.00
301	Tee Shirt	14.00
302	Tee Shirt	14.00
400	Baseball Cap	9.00
401	Baseball Cap	10.00

ID	Name	UnitPrice
500	Visor	7.00
501	Visor	5.95
...	...	...

这种不一致被称作**非可重复读取**，因为如果会计在*同一事务*中再次执行相同的 SELECT 命令，则不会得到相同的结果。

当然，如果会计已经完成了他的事务，例如，在再次使用 SELECT 之前发出了 COMMIT 或 ROLLBACK 命令，情况则有所不同。该数据库可供多个用户同时使用，而且完全允许某人在会计执行事务之前或之后更改值。结果中的变化仅仅因为发生在执行其事务的过程中才不一致。这种情况将导致调度不可序列化。

10. 会计注意到了这种情况，并决定从现在开始不希望看到价格发生更改。隔离级别 2 消除了非可重复读取。作为会计，执行以下语句：

```
SET TEMPORARY OPTION isolation_level = 2;
SELECT ID, Name, UnitPrice
FROM Products;
```

11. 销售经理决定最好将塑料太阳帽的销售推迟到下个星期，这样，她就不必为预计明天会收到的一个大订单报那个较低的价格。在她的窗口中，尝试执行以下语句。该命令将开始执行，然后她的窗口将呈现冻结状态。

```
UPDATE Products
SET UnitPrice = 7.00
WHERE ID = 501;
```

数据库服务器在隔离级别 2 上必须确保可重复读取。由于会计正在使用隔离级别 2，因此数据库服务器在会计读取的 Products 表的每一行上都放置一个读锁定。当销售经理尝试将价格更改回原来的值时，她的事务必须在 Products 表中包含塑料太阳帽的那一行上获取一个写锁定。由于写锁定是独占的，因此她的事务必须等到会计的事务释放其读锁定后才能继续执行。

12. 会计查看完价格后，由于他不希望无意中更改了数据库，因此使用 ROLLBACK 语句完成他的事务。

```
ROLLBACK;
```

数据库服务器执行该语句后，销售经理的事务完成。

ID	Name	UnitPrice
500	Visor	7.00
501	Visor	7.00

13. 销售经理现在就可以完成她的工作了。她希望提交她所做的更改，恢复到原来的价格。

```
COMMIT;
```

锁的类型和不同的隔离级别

将会计的隔离级别从级别 1 升级到级别 2 后，数据库服务器将在以前未获取任何锁定的位置上使用读锁定。一般来说，各个隔离级别所需的锁类型和处理其它事务持有的锁的方式各不相同。

处于隔离级别 0 时，数据库服务器只需要写锁定。它使用这些锁来确保任何两个事务都无法完成相互冲突的修改。例如，级别 0 的事务在更新或删除一行之前在该行上获取一个写锁定，而插入新行时，这些新行上已经具有写锁定。

级别 0 的事务对它们读取的行不进行任何检查。例如，当一个级别 0 的事务读取一行时，它不会检查其它事务是否已在该行上获取了何种锁。由于不需要进行任何检查，因此级别 0 的事务很快。速度快是以牺牲一致性为代价的。如果读取的行已被其它事务执行了写锁定，就有返回脏数据的危险。

在级别 1 上，事务在读取行之前将检查该行是否已被写锁定。虽然需要多执行一项操作，但这些事务可以确保读取的所有数据都是已提交的数据。请尝试重复第一个教程，将隔离级别设置为 1（而不是 0），然后您会发现，在销售经理更新 T 恤衫价格的事务保持未完成状态时，无法进行会计计算。

当会计将其隔离级别提高到级别 2 后，数据库服务器开始使用读锁定。从这时开始，数据库服务器将在符合会计的选择条件的每一行上为他的事务获取一个读锁定。

事务阻塞

在上面的教程中，销售经理的窗口在执行她的 UPDATE 命令时被冻结。数据库服务器开始执行她的命令，之后发现会计的事务已在销售经理需要更改的行上获取了一个读锁定。这时，数据库服务器只是暂停该 UPDATE 命令的执行。在会计使用 ROLLBACK 完成了他的事务之后，数据库服务器自动释放了他的锁定。发现不再有任何阻碍后，数据库服务器将继续执行销售经理的 UPDATE 命令。

一般来说，当一个事务试图在另一个事务持有锁的行上获取独占锁时，或当一个事务试图在另一个事务持有独占锁的行上获取共享锁时，将出现锁定冲突。这个事务必须等待另一个事务完成才能继续。我们说，那个必须等待的事务被另一个事务**阻塞**。

当数据库服务器发现导致某个事务无法立即继续执行的锁定冲突时，它可以或者暂停执行该事务，或者终止该事务，回退所有更改，并返回一个错误。可以通过设置 blocking 选项来控制其行为。当 blocking 被设置为 On 时，第二个事务将等待，如上面的教程所述。

有关 blocking 选项的详细信息，请参见“blocking 选项”一节第 118 页。

使用快照隔离避免非可重复的读取

您还可以使用快照隔离帮助您避免阻塞。因为使用快照隔离的事务仅查看已提交的数据，所以会计的事务不会阻塞销售经理的事务。

1. 启动 Interactive SQL。
2. 在 [**连接**] 窗口中，以销售经理的身份连接到 SQL Anywhere 示例数据库：
   - 在 [**ODBC 数据源名称**] 字段中，选择 [**SQL Anywhere 11 Demo**]。
   - 单击 [**高级**] 选项卡，在 [**ConnectionName**] 字段中键入 **Sales Manager**。
   - 单击 [**确定**]。

3. 再启动一个 Interactive SQL 的实例。
4. 在 [**连接**] 窗口中，以会计的身份连接到 SQL Anywhere 示例数据库：
   - 在 [**ODBC 数据源名称**] 字段中，选择 [**SQL Anywhere 11 Demo**]。
   - 单击 [**高级**] 选项卡，在 [**ConnectionName**] 字段中键入 **Accountant**。
   - 单击 [**确定**]。
5. 执行以下语句为数据库启用快照隔离，并指定使用快照隔离级别：

```
SET OPTION PUBLIC.allow_snapshot_isolation = 'On';
SET TEMPORARY OPTION isolation_level = snapshot;
```

6. 会计决定列出太阳帽的价格。作为会计，执行以下命令：

```
SELECT ID, Name, UnitPrice
FROM Products
ORDER BY ID;
```

ID	Name	UnitPrice
300	Tee Shirt	9.00
301	Tee Shirt	14.00
302	Tee Shirt	14.00
400	Baseball Cap	9.00
401	Baseball Cap	10.00
500	Visor	7.00
501	Visor	7.00
...	...	...

7. 销售经理决定为塑料太阳帽定一个新的销售价格。作为销售经理，执行以下命令：

```
UPDATE Products
SET UnitPrice = 5.95 WHERE ID = 501;
COMMIT;
SELECT ID, Name, UnitPrice FROM Products
WHERE Name = 'Visor';
```

8. 会计再次执行他的查询且并未看到价格的变化，这是因为事务使用第一次读取时提交的数据。

```
SELECT ID, Name, UnitPrice
FROM Products;
```

9. 作为销售经理，将塑料太阳帽的价格更改回其原来的价格。

```
UPDATE Products
SET UnitPrice = 7.00
WHERE ID = 501;
COMMIT;
```


数据库服务器不会在 Products 表中会计正在读取的行上放置读锁定，因为会计正在查看销售经理对 Products 表进行任何修改之前获取的已提交数据的快照。

10. 会计查看完价格后，由于他不希望无意中更改了数据库，因此使用 ROLLBACK 语句完成他的事务。

```
ROLLBACK;
```

教程：幻像行

在本教程中，您会发现将出现一个幻像行。

注意
要按照此教程正常工作，则一定不要选择 Interactive SQL 中的 [**自动释放数据库锁**] 选项（[**工具**] » [**选项**] » [**SQL Anywhere**]）。

1. 启动两个 Interactive SQL 的实例。请参见“教程：非可重复读取”一节第 141 页的第 1 步到第 4 步。
2. 在 [Sales Manager] 窗口中执行以下命令，将隔离级别设置为 2。

```
SET TEMPORARY OPTION isolation_level = 2;
```

3. 在 [Accountant] 窗口中执行以下命令，将隔离级别设置为 2。

```
SET TEMPORARY OPTION isolation_level = 2;
```

4. 在 [Accountant] 窗口中输入以下命令，列出所有部门。

```
SELECT * FROM Departments
ORDER BY DepartmentID;
```

DepartmentID	DepartmentName	DepartmentHeadID
100	R & D	501
200	Sales	902
300	Finance	1293
400	Marketing	1576
500	Shipping	703

5. 销售经理决定设立一个新部门，专门管理国外市场。Philip Chin（EmployeeID 为 129）将作为这个新部门的经理。

```
INSERT INTO Departments
   (DepartmentID, DepartmentName, DepartmentHeadID)
   VALUES(600, 'Foreign Sales', 129);

COMMIT;
```


最后一个命令为这个新部门创建了一个新条目。这个条目作为新的一行出现在销售经理窗口中的表的底部。

在 [Sales Manager] 窗口中输入以下命令，列出所有部门。

```
SELECT *
FROM Departments
ORDER BY DepartmentID;
```

DepartmentID	DepartmentName	DepartmentHeadID
100	R & D	501
200	Sales	902
300	Finance	1293
400	Marketing	1576
500	Shipping	703
600	Foreign Sales	129

6. 但是，会计并不知道这个新部门。处于隔离级别 2 时，数据库服务器通过放置锁来确保所有行都不会发生更改，但却不会通过放置锁来防止其它事务插入新行。

   会计只有再次执行 SELECT 命令才能发现新行。在会计的窗口中，再次执行 SELECT 语句。您将看到新行已附加到表中。

   ```
   SELECT *
   FROM Departments
   ORDER BY DepartmentID;
   ```

DepartmentID	DepartmentName	DepartmentHeadID
100	R & D	501
200	Sales	902
300	Finance	1293
400	Marketing	1576
500	Shipping	703
600	Foreign Sales	129

   出现的新行被称作**幻像行**，这是因为，从会计的角度来看，该行就象幻影那样，不知从何而来。会计的连接处于隔离级别 2。在该级别上，数据库服务器只在会计使用的行上获取锁。而对其它行则不进行任何控制，因此没有任何限制阻止销售经理插入新行。

7. 会计希望在将来避免这种奇异的事情发生，因此他将其当前事务的隔离级别提高到级别 3。为会计输入以下命令。

```
SET TEMPORARY OPTION isolation_level = 3;
SELECT *
FROM Departments
ORDER BY DepartmentID;
```

8. 而销售经理想要再增加一个部门来处理针对大型企业合作伙伴的销售业务。在销售经理的窗口中执行以下命令。

```
INSERT INTO Departments
 (DepartmentID, DepartmentName, DepartmentHeadID)
   VALUES(700, 'Major Account Sales', 902);
```

   销售经理的窗口在执行该命令的过程中将会暂停，因为会计放置的锁阻止了该命令。在工具栏上，单击 [**中断 SQL 语句**]（或选择 [**SQL**] » [**停止**]），中断这次输入。

9. 为了避免更改 SQL Anywhere 示例数据，您应该回退未完成的插入 Major Account Sales 部门行的事务，并使用另一个事务删除 Foreign Sales 部门。

   a. 在销售经理的窗口中执行以下命令，回退最后的未完成事务：

```
ROLLBACK;
```

   b. 同样在销售经理的窗口中，执行以下两条语句，删除以前插入的行并提交此操作。

```
DELETE FROM Departments
WHERE DepartmentID = 600;

COMMIT;
```

解释

在会计将他的隔离级别提高到级别 3 并再次选择了 Departments 表中的所有行之后，数据库服务器在该表中的每一行上都放置了防插入锁，并且添加了一个额外的幻像锁以防止在该表的末尾插入新行。当销售经理尝试在该表的末尾插入新行时，就是这最后一个锁阻塞了她的命令。

注意，即使销售经理的连接仍处于隔离级别 2，她的命令也会被阻塞。数据库服务器按照每个事务的隔离级别和语句的要求放置防插入锁（与读锁定类似）。一旦放置这些锁定，所有其它并发事务必须都遵从。

有关锁定的详细信息，请参见“锁定的工作方式”一节第 122 页。

使用快照隔离避免幻像行

您可以使用快照隔离级别维护与隔离级别 3 相同的一致性，而且不会造成任何种类的阻塞。销售经理的命令未被阻塞，会计也不会看到幻像行。

如果尚未执行“教程：幻像行”一节第 146 页中的第 1 步到第 4 步，请执行这些步骤。这些步骤介绍了如何启动两个 Interactive SQL 实例。

1. 通过执行以下命令为会计启用快照隔离。

```
SET OPTION PUBLIC. allow_snapshot_isolation = 'On';
SET TEMPORARY OPTION isolation_level = snapshot;
```

2. 在 [Accountant] 窗口中输入以下命令，列出所有部门。

```
SELECT * FROM Departments
ORDER BY DepartmentID;
```


DepartmentID	DepartmentName	DepartmentHeadID
100	R & D	501
200	Sales	902
300	Finance	1293
400	Marketing	1576
500	Shipping	703

3. 销售经理决定设立一个新部门，专门管理国外市场。Philip Chin（EmployeeID 为 129）将作为这个新部门的经理。

```
INSERT INTO Departments
   (DepartmentID, DepartmentName, DepartmentHeadID)
   VALUES(600, 'Foreign Sales', 129);
COMMIT;
```

最后一个命令为这个新部门创建了一个新条目。这个条目作为新的一行出现在销售经理窗口中的表的底部。

```
SELECT * FROM Departments
ORDER BY DepartmentID;
```

DepartmentID	DepartmentName	DepartmentHeadID
100	R & D	501
200	Sales	902
300	Finance	1293
400	Marketing	1576
500	Shipping	703
600	Foreign Sales	129

4. 会计可以再次执行他的查询，并且不会看到新行，这是因为事务尚未结束。

```
SELECT *
FROM Departments
ORDER BY DepartmentID;
```

DepartmentID	DepartmentName	DepartmentHeadID
100	R & D	501
200	Sales	902

DepartmentID	DepartmentName	DepartmentHeadID
300	Finance	1293
400	Marketing	1576
500	Shipping	703

5. 而销售经理想要再增加一个部门来处理针对大型企业合作伙伴的销售业务。在销售经理的窗口中执行以下命令。

```
INSERT INTO Departments
 (DepartmentID, DepartmentName, DepartmentHeadID)
   VALUES(700, 'Major Account Sales', 902);
```

销售经理的更改未被阻塞，因为会计在使用快照隔离。

6. 会计必须结束他的快照事务，以查看销售经理提交到数据库的更改。

```
COMMIT;
   SELECT * FROM Departments
   ORDER BY DepartmentID;
```

现在会计看到了 Foreign Sales 部门，但没有看到 Major Account Sales 部门。

DepartmentID	DepartmentName	DepartmentHeadID
100	R & D	501
200	Sales	902
300	Finance	1293
400	Marketing	1576
500	Shipping	703
600	Foreign Sales	129

7. 为了避免更改 SQL Anywhere 示例数据，您应该回退未完成的插入 Major Account Sales 部门行的事务，并使用另一个事务删除 Foreign Sales 部门。

   a. 在销售经理的窗口中执行以下命令，回退最后的未完成事务：

```
ROLLBACK;
```

   b. 同样在销售经理的窗口中，执行以下两条语句，删除之前插入的行并提交此操作。

```
DELETE FROM Departments
WHERE DepartmentID = 600;

COMMIT;
```


教程：锁定的实际含义

在本教程中，会计和销售经理都有涉及 SalesOrder 和 SalesOrderItems 表的任务。会计需要核实根据销售雇员在 2001 年 4 月的销售情况付给他们的佣金支票的金额。销售经理发现有几个订单没有添加到数据库中，想要添加这些订单。

他们的工作演示了幻像锁定。**幻像锁**是一个共享锁，放置在带有索引的扫描位置上以防止幻像行。当使用隔离级别 3 的事务选择了符合给定条件的行时，数据库服务器将放置防插入锁，以防止其它事务插入也符合给定条件的行。替您放置的锁的数量取决于搜索条件和数据库的设计。

注意
要按照此教程正常工作，则一定不要选择 Interactive SQL 中的 [**自动释放数据库锁**] 选项（[**工具**] » [**选项**] » [**SQL Anywhere**]）。

1. 启动两个 Interactive SQL 的实例。请参见“教程：非可重复读取”一节第 141 页的第 1 步到第 4 步。
2. 在 [Sales Manager] 窗口和 [Accountant] 窗口中执行以下命令，将其隔离级别都设置为 2。

   ```
   SET TEMPORARY OPTION isolation_level = 2;
   ```

3. 每个月都会向销售代表支付佣金，所付佣金是他们在该月的销售额乘以一定的百分比。会计正在准备 2001 年 4 月的佣金支票。他的第一项任务是计算各个销售代表在该月的总销售额。

   在会计的窗口中输入以下命令。价格、销售订单信息和雇员数据存储在单独的表中。使用外键关系连接这些表，以将所需信息合并到一起。

   ```
   SELECT EmployeeID, GivenName, Surname,
      SUM(SalesOrderItems.Quantity * UnitPrice)
         AS "April sales"
   FROM Employees
      KEY JOIN SalesOrders
      KEY JOIN SalesOrderItems
      KEY JOIN Products
   WHERE '2001-04-01' <= OrderDate
      AND OrderDate < '2001-05-01'
   GROUP BY  EmployeeID, GivenName, Surname
   ORDER BY EmployeeID;
   ```

EmployeeID	GivenName	Surname	April sales
129	Philip	Chin	2160.00
195	Marc	Dill	2568.00
299	Rollin	Overbey	5760.00
467	James	Klobucher	3228.00
...	...	...	...

4. 销售经理发现由 Philip Chin 拿到的一个大订单没有输入数据库。Philip 希望能立即拿到佣金，因此销售经理输入了这个遗漏的订单。该订单是在 4 月 25 日拿到的。

在销售经理的窗口中，输入以下命令。将销售订单和各项商品输入到单独的表中，因为一个订单可能包含多项商品。应先为销售订单创建条目，然后再将各项商品添加到该订单中。为保持参照完整性，只有在订单已经存在的情况下，数据库服务器才允许事务将各项商品添加到该订单中。

```
INSERT into SalesOrders
VALUES ( 2653, 174, '2001-04-22', 'r1',
      'Central', 129);
INSERT into SalesOrderItems
VALUES ( 2653, 1, 601, 100, '2001-04-25' );
COMMIT;
```

5. 会计不可能知道销售经理刚添加了一个新订单。如果早些将这个新订单添加到数据库中，计算 Philip Chin 四月份的销售额时将包括这个订单。

   在会计的窗口中再次计算四月份的总销售额。使用相同的命令进行计算，您将发现 Philip Chin 在四月份的销售额变为 $4560.00。

EmployeeID	GivenName	Surname	April sales
129	Philip	Chin	4560.00
195	Marc	Dill	2568.00
299	Rollin	Overbey	5760.00
467	James	Klobucher	3228.00
...	...	...	...

   假设会计现在对四月份拿到的所有订单作出标记，以表示已支付了佣金。虽然销售经理刚输入的订单没有计入 Philip 四月份的总销售额中，但在第二次搜索时可能会找到该订单并将其标记为佣金已付！

6. 处于隔离级别 3 时，数据库服务器放置防插入锁以确保任何其它事务都不能添加符合搜索或选择条件的行。

   在销售经理的窗口中，执行以下语句，删除那个新订单。

```
DELETE
FROM SalesOrderItems
WHERE ID = 2653;
DELETE
FROM SalesOrders
WHERE ID = 2653;
COMMIT;
```

7. 在会计的窗口中，执行以下两个语句。

```
ROLLBACK;
SET TEMPORARY OPTION isolation_level = 3;
```

8. 在会计的窗口中，执行与以前相同的查询。

```
SELECT EmployeeID, GivenName, Surname,
   SUM(SalesOrderItems.Quantity * UnitPrice)
      AS "April sales"
```



```
FROM Employees
   KEY JOIN SalesOrders
   KEY JOIN SalesOrderItems
   KEY JOIN Products
WHERE '2001-04-01' <= OrderDate
   AND OrderDate < '2001-05-01'
GROUP BY  EmployeeID, GivenName, Surname;
```

由于已将隔离级别设置为 3，因此数据库服务器将自动放置防插入锁，以确保销售经理在会计完成其事务之前不能插入四月份的订单商品项。

9. 返回到销售经理的窗口。再次尝试输入 Philip Chin 的那份遗漏的订单。

   ```
   INSERT INTO SalesOrders
   VALUES ( 2653, 174, '2001-04-22',
            'r1','Central', 129);
   ```

   销售经理的窗口将会停止响应，该操作无法完成。在工具栏上，单击 [**中断 SQL 语句**]（或选择 [**SQL**] » [**停止**]），中断这次输入。

10. 虽然销售经理不能将订单输入到四月份，但您可能会认为她还可以将订单输入到五月份。

    将该命令的日期更改为 5 月 5 日，然后重试。

    ```
    INSERT INTO SalesOrders
    VALUES ( 2653, 174, '2001-05-05', 'r1',
          'Central', 129);
    ```

    销售经理的窗口再次停止响应。在工具栏上，单击 [**中断 SQL 语句**]（或选择 [**SQL**] » [**停止**]），中断这次输入。虽然除防止插入所必需的锁以外数据库服务器没有放置多余的锁，但这些锁也可能会干扰大量其它事务。

    数据库服务器在表索引中放置锁。例如，如果它在一个索引中放置了幻像锁，那么就不能在该索引的前一位置插入新行。但是，如果没有合适的索引，它必须锁定表中的每一行。

    有些情况下，防插入锁可能会阻塞某些在表中插入行的操作，但允许其它操作。

11. 销售经理要在编号为 2651 的订单中再添加一项商品。可使用以下命令。

    ```
    INSERT INTO SalesOrderItems
    VALUES ( 2651, 2, 302, 4, '2001-05-22' );
    ```

    销售经理的窗口停止响应。在工具栏上，单击 [**中断 SQL 语句**]（或选择 [**SQL**] » [**停止**]），中断这次输入。

12. 在本教程的最后，请撤消所有更改，以避免更改 SQL Anywhere 示例数据库。在销售经理的窗口中输入以下命令。

    ```
    ROLLBACK;
    ```

    在会计的窗口中输入相同命令。

    ```
    ROLLBACK;
    ```

关闭两个窗口。

主键生成和并发

您可能会遇到数据库应自动生成唯一编号的情况。例如，如果您正在构建一个用于存储销售发票的表，您可能希望数据库自动指派唯一的发票编号，而不希望让销售人员随意指定。

示例

例如，可以通过在前一个发票编号上加 1 来获取发票编号。如果有多人同时在数据库中添加发票，则不能使用这种方法。两个雇员可能会决定使用相同的发票编号。

这个问题有多种解决方法：

- 为每个添加新发票的人指派一个发票编号范围。

  您可以通过创建一个具有用户名列和发票编号列的表来实施这种方案。每个添加发票的用户在该表中都将有一行。用户每次添加发票时，该表中的编号将递增，并用于新添加的发票。为处理数据库中的所有表，该表应具有三列：表名、用户名和最后的键值。应定期验证每个人是否有足够的编号。

- 创建一个具有以下两列的表：表名和最后的键值。

  此表中有一行包含使用的最后一个发票号。每次有人添加发票时，将建立一个新连接，递增该表中的编号，然后立即提交更改。递增后的编号可以用于新添加的发票。其他用户仍可以取得发票编号，因为您使用了一个单独的事务来更新该行，而且这个事务很快就已完成。

- 使用带有 NEWID 的缺省值及 UNIQUEIDENTIFIER 二进制数据类型的列可生成全局唯一标识符。

  UUID 和 GUID 值可用于标识表中的唯一行。在一台计算机上所生成的值与在其它计算机上生成的值不相同。因此，可将它们用作复制和同步环境中的键。

  有关生成唯一标识符的详细信息，请参见“NEWID 缺省值”一节第 83 页。

- 使用缺省值为 AUTOINCREMENT 的列。例如：

```
CREATE TABLE Orders (
   OrderID INTEGER NOT NULL DEFAULT AUTOINCREMENT,
   OrderDate DATE,
   primary key( OrderID )
);
```

  每次在表中插入行时，如果没有为自动增量列指定值，将生成一个唯一的值。如果指定了一个值，将使用指定的值。如果指定的值大于该列的当前最大值，将使用指定的值作为以后插入行时自动递增的起点。自动增量列中最近插入的行的值可以用全局变量 @@identity 表示。

数据定义语句和并发

有些数据定义语句（例如，CREATE INDEX、ALTER TABLE 和 TRUNCATE TABLE）可以更改整个表，只要该表当前正被另一连接使用，就无法完成这些语句。这些数据定义语句可能需要很长时间，而且数据库服务器在处理这些命令的过程中，将不处理引用相同表的请求。

CREATE TABLE 语句不会造成任何并发冲突。

GRANT 语句、REVOKE 语句和 SET OPTION 语句也不会造成并发冲突。这些命令会影响任何发送到数据库服务器的新 SQL 语句，但不会影响现有的未完成的语句。

不能对已连接到数据库的用户执行 GRANT 和 REVOKE 命令。

数据定义语句和同步的数据库
在使用同步的数据库中使用数据定义语句时应特别谨慎。请参见 MobiLink - 服务器管理和“数据定义语句”一节 《SQL Remote》 。

小结

在数据库中，事务和锁定是重要性仅次于表之间关系的两个方面。使用锁定时一定要谨慎，并且在构建事务时也一定要仔细，这一点对任何数据库的完整性和性能都有好处。如果要创建的数据库必须并发执行大量命令，这两者都非常重要。

事务可以将 SQL 语句组合成逻辑工作单元。要完成事务，可以回退所有更改，或者提交更改使其成为永久更改。

出现系统故障时，事务对数据恢复非常重要。它们在交错执行并发事务的语句中也扮演着关键角色。

要提高性能，必须并发执行多个事务。每个事务都由 SQL 语句组成。如果并发执行两个或多个事务，数据库服务器必须调度各个语句的执行。与依次执行的事务不同，并发事务会造成不一致。

隔离级别采用四种不一致类型来定义：

- **脏读** 一个事务读取了由另一个事务修改但尚未提交的数据。
- **非可重复的读取** 一个事务读取相同的行两次，但得到的值却不同。
- **幻像行** 一个事务使用特定的条件执行了两次选择行的操作，但在第二次的结果集中发现了新行。
- **更新丢失** 由于允许一个事务基于以前的数据保存更新，导致另一个事务对一行的更改完全丢失。

如果按调度执行语句时数据库达到的状态与依次执行各个事务时相同，那么这个调度就叫做可序列化调度。我们认为，可序列化调度是**正确的**。可序列化调度不会造成以上任何一种不一致。

通过锁定，可以控制允许的干扰程度和类型。SQL Anywhere 提供了四个锁定级别：隔离级别 0、1、2 和 3。使用最高级别（级别 3）时，SQL Anywhere 可确保调度是可序列化的，也就是说，执行所有事务的结果与依次运行这些事务相同。

可惜的是，一个事务获取的锁定可能会阻碍其它事务的进度。由于这个问题，只要较低隔离级别可能会引起的不一致是可以忍受的，就应使用较低的级别。通过较高的隔离级别来提高数据一致性常常意味着并发性会降低，也就是说，数据库处理并发事务的效率会降低。常常需要在一致性与性能之间找到平衡点，从而确定适合各个操作的隔离级别。

如果不同事务的锁定要求之间发生冲突，可能会导致阻塞或死锁。SQL Anywhere 具有用于处理这两种情况的机制，并为您提供了用于控制它们的选项。

但是，使用较高隔离级别的事务也不*总是*会影响并发性。只有在其它事务需要访问锁定行时才会受到阻碍。可以通过仔细设计数据库和事务来提高并发性。例如，可以通过把一个事务分割成两个较短的事务来缩短持有锁定的时间，或者，您可能会发现添加索引可以让事务以较高的隔离级别运行，而放置的锁定反而会减少。

监控和提高数据库性能

本节介绍了如何执行数据库和应用程序分析活动、监控和提高性能以及解决特定的性能问题。

提高数据库性能

目录

要提高数据库性能，必须确定现有数据库的运行是否处于最佳状态。本节介绍有关使用 SQL Anywhere 分析工具来分析和解决数据库性能问题的信息。

SQL Anywhere 提供了多种诊断工具用来检测生产数据库性能问题。其中的大多数工具都依赖于**诊断跟踪**基础结构 - 一个由用于捕获和存储诊断数据的表、文件及其它组件组成的系统。可以利用诊断跟踪数据来执行各种诊断和监控任务，如**应用程序分析**。

可以使用多种方法来分析 SQL Anywhere 的性能数据，其中包括：

- **[应用程序分析向导]** 此向导可在 Sybase Central 的 [应用程序分析] 模式下使用，它提供了一种完全自动化的性能检查方法。此向导结束时，提供了一些改进建议。请参见“应用程序分析”一节第 161 页。
- **[数据库跟踪向导]** 此向导可在 Sybase Central 的 [应用程序分析] 模式下使用，它能够自定义所收集的性能数据的类型。这就允许您监控特定用户或活动的表现。请参见“使用诊断跟踪进行高级应用程序分析”一节第 171 页。
- **请求跟踪分析** 利用此功能，您可以将诊断数据收集的范围缩小到由特定用户或连接发出的请求（语句）。请参见“执行请求跟踪分析”一节第 185 页。
- **索引顾问** 此功能可分析数据库中的索引并提供一些改进建议。可通过 [应用程序分析] 模式来访问此工具，或是将其作为独立工具进行访问。请参见“索引顾问”一节第 167 页。
- **过程分析** 利用此功能，您可以确定执行过程、用户定义的函数、事件、系统触发器和触发器需要多长时间。过程分析作为 Sybase Central 中的一种功能提供。请参见“[应用程序分析] 模式中的过程分析”一节第 162 页。

  也可以使用系统过程来实施过程分析。请参见“使用系统过程进行过程分析”一节第 189 页。

- **执行计划** 此功能允许使用执行计划访问数据库中与语句相关的信息。您可以在 Interactive SQL 中查看执行计划，也可以使用 SQL 函数查看执行计划。可按几种不同格式检索执行计划，并且可保存该计划。请参见“读取执行计划”一节第 569 页。

注意
在本文档中，会将**应用程序分析**和**诊断跟踪**两个词交替使用。诊断跟踪是高级应用程序分析。

应用程序分析

利用应用程序分析生成的数据，您可以了解应用程序与数据库的交互情况，还可以发现并消除性能问题。有两种方法可供用于生成分析信息；一种是自动化方法，即使用 [**应用程序分析向导**]；另一种是使用 Sybase Central 的 [应用程序分析] 模式中提供的各种工具和功能。

Windows Mobile 不支持 [**应用程序分析向导**]；但支持 [**数据库跟踪向导**]。无法自动从 Windows Mobile 设备创建跟踪数据库，且无法跟踪 Windows Mobile 设备上的本地数据库。必须将 Windows Mobile 设备的数据转到在台式计算机上的数据库服务器上运行的 Windows Mobile 数据库副本上进行跟踪。

- **自动化应用程序分析** 使用 Sybase Central 中的 [**应用程序分析向导**] 可发现常见的性能问题。该向导使您可以定义要分析的活动类型，并会在完成分析时向您提供提高数据库性能的建议。[索引顾问] 也已经被集成到 [**应用程序分析向导**] 中，它可以利用数据来建议索引改进。

  自动化方法非常适用于几乎没有数据库连接的环境或不需要进行复杂分析的环境。

- **使用诊断跟踪进行高级应用程序分析** 使用 [**数据库跟踪向导**] 可自定义在跟踪会话期间返回的数据以及数据的存储位置。也可以使用命令行来返回和存储自定义的跟踪数据。可以控制所分析的活动并把目标锁定在特定问题。例如，您可以锁定数据库服务器执行的特定语句、所使用的查询计划、死锁、相互阻塞的连接和性能统计信息等。

  对于数据库具有较高负载的环境或需要进行复杂分析来诊断问题的环境，建议使用高级方法。通过自定义跟踪会话，您可以将跟踪范围缩小至特定的活动，并可将跟踪数据定向到远程数据库。这两种操作都会减少所分析数据库的负载。

  请参见“使用诊断跟踪进行高级应用程序分析”一节第 171 页。

应用程序分析向导

Sybase Central 中的 [**应用程序分析向导**] 为进行应用程序分析提供了一种执行诊断跟踪会话的自动化方法。该向导会收集有关应用程序与数据库交互情况的数据，允许您对所收集的数据进行访问，并会提供索引建立方面的建议（如果有）。请参见“应用程序分析向导”一节第 161 页。

当使用 Sybase Central 中的 [**应用程序分析向导**] 时，该向导会自动创建一个跟踪数据库，该数据库的名称与您在向导中为分析文件指定的名称相同。有关为应用程序分析和诊断跟踪所创建的数据库文件的详细信息，请参见“跟踪会话数据”一节第 171 页。

[**应用程序分析向导**] 不能用于为运行于 Windows Mobile 上的数据库创建跟踪会话。而必须使用 [**数据库跟踪向导**]。请参见“创建诊断跟踪会话”一节第 181 页。

要使 [**应用程序分析向导**] 在切换到 [**应用程序分析**] 模式时无法自动启动，请在该向导的第一个页面上选择 [**以后在切换到应用程序分析模式后不再显示此向导**]。也可通过选择 [**以后不再显示此页**] 取消显示向导的第一页。要随时更改这些选项，请选择 [**工具**] » [**SQL Anywhere 11**] » [**首选项**]，再选择 [**实用程序**] 选项卡，然后选择适当的选项。

♦ 使用 [应用程序分析向导] (Sybase Central)

1. 打开 Sybase Central。

2. 以具有 DBA 特权或 PROFILE 特权的用户身份连接到数据库。
3. 选择 [**模式**] » [**应用程序分析**]。

   如果未出现 [**应用程序分析向导**]，则选择 [**应用程序分析**] » [**打开应用程序分析向导**]。
4. 请按照 [**应用程序分析向导**] 中的说明进行操作。请不要单击 [**完成**]；这将结束分析并关闭向导。

   该向导会执行以下任务：
   - 创建本地数据库来保存诊断跟踪信息
   - 启动网络服务器
   - 启动跟踪会话
   - 提示您运行要分析的应用程序
5. 返回到 [**应用程序分析向导**]，然后单击 [**完成**]。向导结束后，会返回其结果，并允许您查看它在跟踪会话期间收集的数据。

有关从 [**应用程序分析向导**] 返回的索引建议的详细信息，请参见“了解索引顾问建议”一节 第 168 页。

有关跟踪会话期间所收集的过程分析信息的详细信息，请参见“如何读取过程分析结果”一节 第 165 页。

另请参见

- “PROFILE 特权”一节 《SQL Anywhere 服务器 - 数据库管理》

[应用程序分析] 模式中的过程分析

本节介绍如何使用 Sybase Central 中的应用程序分析模式来执行过程分析。这是访问过程分析结果的推荐方法。但也可以使用 SQL 命令来执行过程分析。请参见“使用系统过程进行过程分析”一节第 189 页。

过程分析显示执行过程、用户定义函数、事件、系统触发器和触发器需要多长时间。您还可以在分析期间当这些对象运行后，查看它们的逐行执行时间。然后，利用过程分析结果中提供的信息，您可以确定应调整哪些对象以提高数据库的性能。

过程分析还可以帮助您通过请求记录来分析您发现开销较高的特定数据库过程（包括存储过程、函数、事件和触发器）。它还可以帮助您发现高开销的隐藏过程，例如，触发器、事件和嵌套的存储过程调用。此外，它还可以帮助在过程主体中定位有潜在问题的区域。

过程分析结果由数据库服务器存储在内存中。分析信息是累积性的，并可精确到 1 ms。

启用过程分析

过程分析一旦启用，数据库服务器就会收集分析信息，直到您禁用分析功能或数据库服务器关机为止。

注意
数据库服务器关闭时会将所有分析信息都删除。要导出分析信息，请使用 sa_procedure_profile 系统过程。请参见“sa_procedure_profile 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》。

不能使用 SQL 语句来查询数据库服务器所保留的分析信息。分析信息以内存储数据库服务器数据结构进行存储。

♦ 启用过程分析 (Sybase Central)

1. 打开 Sybase Central。
2. 以具有 DBA 特权或 PROFILE 特权的用户身份连接到数据库。
3. 在左窗格中，选择数据库。
4. 选择 [**模式**] » [**应用程序分析**]。

   如果未出现 [**应用程序分析向导**]，则选择 [**应用程序分析**] » [**打开应用程序分析向导**]。
5. 请按照 [**应用程序分析向导**] 中的说明进行操作。
6. 在 [**分析选项**] 页面上，选择 [**存储过程、函数、触发器或事件执行时间**]。
7. 单击 [**完成**]。

   如果切换到另一模式，将出现一个提示，询问您是否要停止收集过程分析信息。选择 [**否**] 以在继续进行分析的同时在其它模式下继续工作。

另请参见

- “重置过程分析”一节第 163 页
- “禁用过程分析”一节第 164 页
- “分析过程分析的结果”一节第 165 页
- “PROFILE 特权”一节 《SQL Anywhere 服务器 - 数据库管理》

重置过程分析

要清除有关过程、函数、事件和触发器的现有分析信息，可重置过程分析。重置不会在过程分析已启用的情况下将其停止，也不会在过程分析已禁用的情况下将其启动。

♦ 重置分析 (Sybase Central)

1. 打开 Sybase Central。
2. 以具有 DBA 特权或 PROFILE 特权的用户身份连接到数据库。
3. 在左窗格中，选择数据库。
4. 选择 [**模式**] » [**应用程序分析**]。如果出现 [**应用程序分析向导**]，则单击 [**取消**]。
5. 如果过程分析已启用：在 [**应用程序分析详细信息**] 窗格中，单击数据库然后再单击 [**查看所选数据库上的分析设置**]。

如果过程分析没有启用，在左窗格中右击数据库然后选择 [**属性**]。

6. 单击 [**分析设置**] 选项卡。
7. 单击 [**立即重置**]。
8. 单击 [**确定**]。

另请参见

- “启用过程分析”一节第 162 页
- “禁用过程分析”一节第 164 页
- “分析过程分析的结果”一节第 165 页

禁用过程分析

收集完过程、触发器和函数的分析信息后，您就可以禁用过程分析。禁用过程分析时，您还可以选择删除到目前为止所收集的分析信息。如果已经完成了分析工作，您或许希望这样做。

如果没有选择删除分析数据，则可以在 Sybase Central 的 [应用程序分析] 模式下查看这些数据（甚至在禁用了过程分析之后）。

♦ 禁用分析而不删除分析信息 (Sybase Central)

1. 打开 Sybase Central。
2. 以具有 DBA 特权或 PROFILE 特权的用户身份连接到数据库。
3. 在左窗格中，选择数据库。
4. 选择 [**模式**] » [**应用程序分析**]。如果出现 [**应用程序分析向导**]，则单击 [**取消**]。
5. 在 [**应用程序分析详细信息**] 窗格中，单击 [**停止收集所选数据库中的分析信息**]。

♦ 删除分析信息并禁用过程分析 (Sybase Central)

1. 打开 Sybase Central。
2. 以具有 DBA 特权或 PROFILE 特权的用户身份连接到数据库。
3. 在左窗格中，选择数据库。
4. 选择 [**模式**] » [**应用程序分析**]。如果出现 [**应用程序分析向导**]，则单击 [**取消**]。
5. 在 [**应用程序分析详细信息**] 窗格中，选择数据库然后单击 [**查看所选数据库上的分析设置**]。
6. 单击 [**分析设置**] 选项卡。
7. 单击 [**立即清除**]。
8. 单击 [**确定**]。

另请参见

- “启用过程分析”一节第 162 页
- “重置过程分析”一节第 163 页
- “分析过程分析的结果”一节第 165 页

分析过程分析的结果

虽然被称作过程分析，但实际上您能够查看数据库中的存储过程、用户定义函数、触发器、系统触发器和事件的分析结果。

♦ **查看过程分析信息 (Sybase Central)**

1. 以具有 DBA 特权的用户身份连接到数据库并启用过程分析。请参见“启用过程分析”一节第 162 页。
2. 在左窗格中，双击以下内容之一：[**触发器**]、[**系统触发器**]、[**过程和函数**] 或 [**事件**]。
3. 在右窗格中，单击 [**分析结果**] 选项卡。

   启用过程分析后执行的所有所选类型的对象的列表随即出现。

   可能看不到某个预期对象，这是因为尚未执行该对象。或者，它也许已经执行，但结果还未刷新。按 F5 键刷新该列表。

   如果您发现列出的对象比预期的要多，这是因为一个对象可以调用其它对象，所以列出的项目可能比用户显式调用的项目要多。
4. 要查看对某特定对象的详细分析结果，请在 [**分析结果**] 选项卡上双击该对象。

   右窗格的详细信息便替换为该对象的详细分析信息。

如何读取过程分析结果

[**分析结果**] 选项卡汇总了自启动过程分析以来在数据库中执行的所有对象的分析信息（按类型分组）。所显示的信息包括：

列	说明
Name	对象的名称。
Owner	对象的所有者。
Table 或 Table Name	触发器所属的表（该列仅出现在数据库 [分析] 选项卡上）。
Event	对象的类型，例如，过程。
Type	系统触发器的触发器类型。可以是 Update 或 Delete。
# Execs.	每个对象被调用的次数。

列	说明
# msec.	每个对象的总执行时间。

这些列及其内容可能会随着对象类型的不同而变化。

双击某一特定对象（如过程）时，[**分析结果**] 选项卡中将出现特定于该对象的详细信息。所显示的信息包括：

列	说明
Execs	对象中代码行的执行次数。
Milliseconds	执行某行所花费的总时间。
%	执行某行所花费的总时间的百分比表示。
Line	对象中的行号。
Source	执行的代码。

对于代码中与其它行相比执行时间较长的行，应对其进行分析，并考虑是否能通过更有效的方法来获得相同的功能。要访问过程分析信息，您必须连接到数据库，启用分析并具有 DBA 特权。

索引顾问

要运行 [索引顾问]，必须具有 DBA 或 PROFILE 特权。

选择一组适当的索引可以提高数据库性能。SQL Anywhere [索引顾问] 通过为数据库提供有关最佳索引集的建议来帮助您选择索引。

您可以使用 Interactive SQL 对单个查询运行 [索引顾问]，也可使用 Sybase Central 中的 [应用程序分析] 模式对数据库运行 [索引顾问]。在分析数据库时，[索引顾问] 将使用跟踪会话来收集数据并提出建议。它使用这些索引来预计查询执行开销，以便了解哪些索引会导致执行计划得到改进。[索引顾问] 会评估多列索引及单列索引，还会调查聚簇索引或非聚簇索引的影响。

[索引顾问] 会生成候选索引并确定它们对性能的影响，从而对数据库或单个查询进行分析。为了研究不同候选索引的影响，[索引顾问] 会在各组不同的索引下反复重新优化查询。但它不执行查询。

注意
可使用 Sybase Central 连接到版本 9 数据库服务器。但是，Sybase Central 中窗口的布局会恢复为版本 9 的布局（不包含 [应用程序分析] 模式）。有关如何在 Sybase Central 中查找和使用 [索引顾问] 的信息，请参见版本 9 文档。

另请参见

- “使用索引”一节第 66 页
- “索引”一节第 596 页
- “应用程序分析”一节第 161 页
- “PROFILE 特权”一节 《SQL Anywhere 服务器 - 数据库管理》
- “应用程序分析”一节第 161 页
- “了解索引顾问建议”一节第 168 页

获取对查询的 [索引顾问] 建议

♦ **获得对查询的 [索引顾问] 建议 (Sybase Central)**

1. 打开 Sybase Central。
2. 以具有 DBA 特权或 PROFILE 特权的用户身份连接到数据库。
3. 右击数据库，然后选择 [**打开 Interactive SQL**]。
4. 在 [**SQL 语句**] 窗格中，键入查询。
5. 选择 [**工具**] » [**索引顾问**]。

获取对数据库的 [索引顾问] 建议

要获得对整个数据库的 [索引顾问] 建议，请使用 Sybase Central 中的 [应用程序分析] 模式。[索引顾问] 在提供建议之前需要先分析数据。以下过程是使用 [**应用程序分析向导**] 所收集的数据来收集

数据并获取建议的一种快速方法。但是，如果已经拥有应用程序分析数据（例如，已使用 [**数据库跟踪向导**] 分析了数据库），则仍可以在您所创建的跟踪数据库上运行 [索引顾问]。

♦ **使用 [应用程序分析向导] 获取对数据库的 [索引顾问] 建议 (Sybase Central)**

1. 以 DBA 身份或具有 PROFILE 特权的用户身份连接到数据库。
2. 选择 [**模式**] » [**应用程序分析**]。
3. 请按照 [**应用程序分析向导**] 中的说明进行操作。

   如果 [**应用程序分析向导**] 未出现，则选择 [**应用程序分析**] » [**打开应用程序分析向导**]，然后按照向导说明完成操作。
4. 在 Sybase Central 中，选择 [**应用程序分析**] » [**在跟踪数据库上运行索引顾问**]。
5. 请按照 [**索引顾问向导**] 中的说明进行操作。

了解索引顾问建议

分析跟踪会话之前，[索引顾问] 会询问您需要的建议类型：

- **推荐的聚簇索引** 如果选择此选项，[索引顾问] 将分析聚簇索引和非聚簇索引的影响。

  对于某些负载而言，与非聚簇索引相比，选择得当的聚簇索引可以在更大程度上提高性能，但您必须重组表（使用 REORGANIZE TABLE 语句），这样才能使索引生效。此外，如果考虑聚簇索引的影响，分析将占用更长的时间。请参见“使用聚簇索引”一节第 67 页。
- **保持现有的次级索引** [索引顾问] 可以通过在数据库中维护现有的一组次级索引或忽略现有的次级索引来执行分析。次级索引是这样一个索引：它不是唯一约束，也不是主键或外键。在选择访问计划时，总是会考虑为实施参照完整性约束而存在的索引。

分析包括以下步骤：

- **生成候选索引** 对于每个跟踪会话，[索引顾问] 都会生成一组候选索引。在大表上创建实际索引是一个耗时的操作，因此 [索引顾问] 将它的候选索引创建为虚拟索引。虚拟索引不能用来实际执行查询，但优化程序可以使用虚拟索引来预计执行计划的开销，就像实际存在此类索引一样。虚拟索引允许 [索引顾问] 执行假定推测分析，这样就不会产生创建和管理实际索引的开销。虚拟索引的列数限制为四列。
- **测试候选索引的收益和开销** [索引顾问] 会要求优化程序预计在使用和不使用候选索引的不同组合的情况下在跟踪数据库中执行查询的开销。
- **生成建议** [索引顾问] 汇编查询开销的结果，并按索引提供的总收益对索引排序。它提供一个 SQL 脚本，您可以通过运行该脚本来实现建议，也可以保存该脚本供自己检查和分析。

了解 [索引顾问] 结果

[索引顾问] 提供一组选项卡来存放给定分析的结果。可以保存分析结果供以后审核。

[概览] 选项卡

[**概览**] 选项卡提供了分析的概要信息，包括查询数量、推荐索引的数量、推荐索引所需的页数，以及推荐索引预期产生的收益。收益数量用内部开销单位计量。

[推荐索引] 选项卡

[**推荐索引**] 选项卡包含有关每个推荐索引的数据。所提供的信息包括：

- **聚簇** 每个表最多可以有一个聚簇索引。某些情况下，聚簇索引实现的收益要明显多于非聚簇索引。请参见“使用聚簇索引”一节第 67 页。
- **页数** 选择创建索引时，预计保存索引所需的数据库页数量。请参见“表大小和页面大小”一节第 595 页。
- **相对收益** 一个 1 至 10 之间的数字，表明创建指定索引的预计总体收益。数字越大，说明收益越大。

  相对收益是使用独立于 [总成本收益] 列的内部算法计算的。预计相对收益时，有几个因素是总开销收益中没有的。例如，一个索引的存在可以显著影响与另一索引相关的收益。在此情况下，相对收益将尝试估计每个索引各自的影响。

  有关详细信息，请参见“执行 [索引顾问] 结果”一节第 170 页。
- **总收益** 与索引相关的成本缩减，针对跟踪会话中的所有操作汇总得出，用内部成本单位（成本模型）计量。请参见“优化程序的工作原理”一节第 525 页。
- **更新开销** 添加索引将增加开销，一方面需要额外的存储空间，另一方面在修改数据时也需要额外的工作。[更新开销] 列是与索引相关的额外维护开销的预计值。它用内部开销单位计量。
- **总成本收益** 总收益减去与索引相关的更新开销。

[请求] 选项卡

[**请求**] 选项卡用于为跟踪会话内的各个请求细分建议的影响。该信息包含在应用推荐索引之前和之后的预计开销，以及由查询使用的虚拟索引。点击一个按钮即可查看为请求找到的最佳执行计划。

[更新] 选项卡

[**更新**] 选项卡用于细分建议的影响。

[未使用的索引] 选项卡

[**未使用的索引**] 选项卡用于列出已存在于数据库中，但未在跟踪会话内的任何请求执行中使用的索引。只列出次级索引：也就是说，既不会列出主键和外键上的索引，也不会列出唯一约束。

[日志] 选项卡

[**日志**] 选项卡用于列出针对此分析已完成的活动。

另请参见

- “使用索引”一节第 66 页
- “索引”一节第 596 页
- “应用程序分析”一节第 161 页

执行 [索引顾问] 结果

尽管 [索引顾问] 提供一个可用于执行其结果的 SQL 脚本，但您可能想要在执行结果前先对结果进行访问。例如，您可能想要将分析期间所生成的建议索引名称重命名。

访问结果时，请考虑以下事项：

- **建议的索引是否与您自己的期望相符？** 如果您非常了解数据库中的数据，并且了解目前对数据库运行的查询，那么您可能需要凭借自己的认识来检查建议的索引是否有用。或许建议的索引只影响很少运行的单个查询，或许建议的索引位于小表上，总体影响较小。或许 [索引顾问] 建议应删除的索引用于其它一些没有包含在您的跟踪会话中的任务。
- **建议的各个索引所产生的效果之间是否存在很强的相关性？** 索引建议会尝试单独评估各个索引的相对收益。不过，情况可能会是这样：两个索引只有都存在时才会被使用（如果它们存在，查询可能会使用两个，如果缺少其中一个，则哪个也不用）。您可以进一步研究 [**请求**] 选项卡并检查查询计划，以查看建议索引的使用方式。
- **您是否能在创建聚簇索引时重组表？** 为了充分利用聚簇索引，您应该使用 REORGANIZE TABLE 语句重组创建聚簇索引时所在的表。如果 [索引顾问] 推荐许多聚簇索引，您可能需要卸载和重装数据库才能获得全部益处。卸载和重装表是一个非常耗时的操作，并且需要大量的磁盘空间资源。您可能需要确认自己具有实现建议所需的时间和资源。
- **分析期间的服务器和连接的状态是否反映了生产运营中的实际状态？** 分析结果取决于数据库服务器的状态，包括哪些数据位于高速缓存中。它们也取决于连接的状态，包括某些数据库选项设置。由于分析只创建虚拟索引而不执行请求，因此，数据库服务器的状态在分析期间实质是静态的（其它连接引入的更改除外）。如果该状态不能代表数据库的典型操作，您可能想在其它不同条件下重新运行分析。

另请参见

- “了解索引顾问建议”一节第 168 页
- “使用 SQL 命令文件”一节第 721 页
- “使用索引”一节第 66 页
- “索引”一节第 596 页
- “REORGANIZE TABLE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “应用程序分析”一节第 161 页

使用诊断跟踪进行高级应用程序分析

诊断跟踪是一种高级的应用程序分析方法。数据库服务器所产生的诊断跟踪数据可以包括数据库服务器所处理的语句的时间戳和连接 ID。对于查询，诊断跟踪数据还包括隔离级别、读取的行数、游标类型和查询执行计划。对于 INSERT、UPDATE 和 DELETE 语句，该数据还包括受到影响的行数。诊断跟踪还可用于记录有关锁定和死锁的信息，并可捕获大量性能统计信息。

您使用诊断跟踪期间收集的数据执行深入的应用程序分析活动，如确定并解决：

- 特定的性能问题
- 执行速度非常慢的语句
- 不正确的选项设置
- 导致优化程序选取次优计划的情况
- 资源争用（CPU、内存、磁盘 I/O）
- 应用程序逻辑问题

某些工具（如 [索引顾问]）也使用跟踪数据来就如何更改数据库或应用程序才能提高性能给出特定的建议。

跟踪体系结构是稳健且可伸缩的。它可以记录要求将记录存入日志的所有信息，以及用以支持定制分析的详细信息。有关请求日志记录的信息，请参见“执行请求跟踪分析”一节第 185 页。

另请参见

- “应用程序分析”一节第 161 页

跟踪会话数据

诊断跟踪数据在**跟踪会话**期间收集。有三种方法可用于捕获跟踪会话数据：

- Sybase Central 中的 [**数据库跟踪向导**]
- 显然，执行 [**应用程序分析向导**] 期间会自动进行此操作
- ATTACH TRACING 和 DETACH TRACING 语句

当跟踪会话正在进行中时，SQL Anywhere 会为指定的数据库生成诊断信息。所生成的诊断数据量取决于跟踪设置。有关如何配置所生成的跟踪数据的数量和类型的详细信息，请参见“配置诊断跟踪”一节第 172 页。

可将被分析的数据库称为**生产数据库**、源数据库或被分析数据库。存储跟踪数据的数据库称作**跟踪数据库**。生产数据库和跟踪数据库可以是同一个数据库。然而，为避免增加生产数据库的大小，建议您将跟踪数据存储在一个单独的数据库中。数据库文件的大小增加后就不能再缩小。另外，如果存储和维护跟踪数据的开销在另一个数据库中执行，则生产数据库的性能会更好，尤其当生产数据库很大且使用频繁时更是如此。

跟踪数据库中保存跟踪数据的表称作**诊断跟踪表**。这些表属于 dbo。有关这些表的详细信息，请参见“诊断跟踪表”一节 《SQL Anywhere 服务器 - SQL 参考》。

注意
Windows Mobile 不支持 [**应用程序分析向导**]；但支持 [**数据库跟踪向导**]。此外，必须将 Windows Mobile 设备的数据转到在台式计算机上的数据库服务器上运行的 Windows Mobile 数据库副本上进行跟踪。无法自动从 Windows Mobile 设备创建跟踪数据库，且无法跟踪 Windows Mobile 设备上的本地数据库。

跟踪会话期间创建的文件

根据所使用的是 [**应用程序分析向导**] 还是 [**数据库跟踪向导**]，跟踪会话创建和使用的文件会有所不同。

运行 [**应用程序分析向导**] 时，该向导会以静默方式在后台捕获跟踪会话，同时创建跟踪数据库来存储诊断表。此外部数据库会使用您在向导中指定的名称和位置进行创建，并具有扩展名 *.adb*。该向导还会创建一个分析日志文件，该文件与跟踪数据库在同一个目录下，并使用相同的名称，但是扩展名为 *.alg*。此分析日志文件包含由向导完成的分析工作的结果，并可随时在文本编辑器中打开。

处理完 [**应用程序分析向导**] 所生成的数据后，就可以删除与会话相关联的跟踪数据库和分析日志文件。

当使用 [**数据库跟踪向导**] 创建跟踪会话时，该向导会询问您是选择在内部将跟踪数据保存在生产数据库中，还是在外部将它们保存在一个单独的数据库中（例如，*tracingData.db*）。建议创建外部跟踪数据库。请参见“创建外部跟踪数据库”一节第 185 页。

注意
跟踪信息*不会*作为数据库卸载或重装操作的一部分被卸载。如果需要从一个数据库向另一个数据库传输跟踪信息，必须通过复制 sa_diagnostic_* 表的内容来手动执行；但是，不建议采用此方法。

配置诊断跟踪

不能更改 Sybase Central 中 [**应用程序分析向导**] 的预配置跟踪设置。但可以使用 [**数据库跟踪向导**] 来配置跟踪活动的几乎所有方面。可使用以下方法之一来配置诊断跟踪设置：

- 使用 Sybase Central 中的 [**数据库跟踪向导**]。建议使用此方法，因为此方法允许您查看所有生效的跟踪设置。请参见“更改诊断跟踪配置设置”一节第 180 页。
- 使用系统过程来更改存储在诊断跟踪表中的设置。有关用于管理应用程序分析的系统过程的详细信息，请参见“sa_set_tracing_level 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》和“sa_save_trace_data 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》。

跟踪设置存储在 sa_diagnostic_tracing_level 系统表中。请参见“sa_diagnostic_tracing_level 表”一节 《SQL Anywhere 服务器 - SQL 参考》。

SendingTracingTo 和 ReceivingTracingFrom 数据库属性分别标识跟踪数据库和生产数据库。有关这些属性的详细信息，请参见“数据库属性”一节 《SQL Anywhere 服务器 - 数据库管理》。

选择诊断跟踪级别

诊断跟踪设置被分成几个级别，但您也可以在这些级别内进一步对设置进行自定义。在各个级别所收集的信息的类型称作**诊断跟踪类型**。下面是对您可以指定的级别及其所包含的诊断跟踪类型的说明。有关下述诊断跟踪类型的说明，请参见“诊断跟踪类型”一节第 175 页。

自定义诊断跟踪设置使您可以减少诊断跟踪会话中不需要的跟踪数据的数量。例如，假设用户 AliceB 一直抱怨她的应用程序运行太慢，但其他用户并未遇到同样的问题。现在您想确切地知道 AliceB 的查询出了什么问题。这意味着，您应当收集作为 AliceB 应用程序的一部分运行的所有查询和其它语句的清单，以及长时间运行的查询的所有查询计划。为此，您只需将诊断跟踪级别设置为 3 并生成一天或两天的跟踪数据。但是，因为此级别会大大影响其他用户的性能，所以您应当将跟踪限制为仅对 AliceB 的活动来进行。为此，请将诊断跟踪级别设置为 3，然后将诊断跟踪范围自定义为 USER，并指定 AliceB 作为用户名。让诊断跟踪会话运行几个小时，然后检查结果。

建议使用 [**数据库跟踪向导**] 来自定义诊断跟踪设置。请参见“更改诊断跟踪配置设置”一节第 180 页。

也可以使用 sa_set_tracing_level 系统过程；但是，使用此方法无法进行与上述方法一样多的自定义。另请参见“sa_set_tracing_level 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》。

您最好不要在跟踪会话进行时更改诊断跟踪设置，因为这样会让解释数据变得更加困难。但是，也可以这样做。请参见“在跟踪会话运行过程中更改诊断跟踪设置”一节第 181 页。

诊断跟踪级别

以下是 [**数据库跟踪向导**] 中指定的诊断跟踪级别的列表。有关各种诊断跟踪类型的说明，请参见“诊断跟踪类型”一节第 175 页。

各级别对性能的影响是基于以下假定估计的：将跟踪数据发送到另一个数据库服务器上的跟踪数据库中（推荐）。

- **级别 0** 此级别保持跟踪会话处于运行状态，但不会将任何跟踪数据发送到跟踪表。
- **级别 1** 收集性能计数器以及已执行语句的取样（每 5 秒一次）。对于此级别，诊断跟踪类型包括：
  - volatile_statistics，每 1 秒取样一次
  - non_volatile_statistics，每 60 秒取样一次

  此级别对性能影响甚微。
- **级别 2** 此级别收集性能计数器、已执行计划的取样（每 5 秒一次），并记录所有已执行的语句。对于此级别，诊断跟踪类型包括：
  - volatile_statistics，每 1 秒取样一次
  - non_volatile_statistics，每 60 秒取样一次
  - 语句
  - 计划，每 5 秒取样一次

此级别对性能影响程度中等—最多不超过 20% 的开销。

- **级别 3** 此级别所记录的详细信息与级别 2 相同，但计划采样更加频繁（每 2 秒一次），并记录详细的阻塞和死锁信息。对于此级别，诊断跟踪类型包括：
  - volatile_statistics，每 1 秒取样一次
  - non_volatile_statistics，每 60 秒取样一次
  - 语句
  - 阻塞
  - 死锁
  - statements_with_variables
  - 计划，每 2 秒取样一次

  此级别对性能的影响最大—开销大于 20%。

诊断跟踪范围

下面是诊断跟踪**范围**的列表。范围值可用于将跟踪限制为引发数据库中活动的谁（或什么）。例如，您可以将范围设置为跟踪来自指定连接的请求。范围值存储在 dbo.sa_diagnostic_tracing_level 诊断表的 scope 列中，并可能具有相应的参数（通常是一个存储在 identifier 列的标识符，如对象名或用户名）。scope 列中的值反映了 [**数据库跟踪向导**] 中指定的设置。

scope 列中的值	说明
DATABASE	记录数据库中发生的所有事件的跟踪数据（假定事件符合指定的级别和条件）。当需要确定大开销查询的来源时，用于对数据库进行长期后台监控，或用于短期诊断。 当指定 DATABASE 时，不存在要指定的标识符。
ORIGIN	记录源自数据库外部或内部的查询的跟踪数据。 当指定范围 ORIGIN 时，您可以指定下列两个可能的标识符：External 或 Internal。External 指定记录来自数据库服务器外部的查询（并符合指定的级别和条件）的语句文本和相关详细信息。Internal 指定对来自数据库服务器内部的查询（并符合指定的级别和条件）记录同样的信息。
USER	仅对指定用户和指定用户创建的连接所发出的查询记录跟踪数据。此范围用于诊断源自特定用户的有问题的查询。 此范围的标识符为对其执行跟踪的那个用户的名称。

scope 列中的值	说明
CONNECTION_NAME 或 CONNECTION_NUMBER	仅对当前连接所执行的语句记录跟踪数据。当用户具有多个连接且其中一个连接正在执行大开销的语句时，使用这些范围。 此范围的标识符分别为连接的名称或连接的编号。
FUNCTION、PROCEDURE、EVENT、TRIGGER 或 TABLE	仅对使用指定对象的语句记录跟踪数据。如果该对象引用了其它对象，则也记录所引用对象的所有数据。例如，如果对某个过程进行跟踪，该过程使用一个函数，而该函数又触发某个事件，则将记录所有这三个对象的语句（假定这些对象符合提供给记录的指定级别和条件）。当使用某个特定对象的开销很大时，或者当引用对象的语句要花费相当长的时间才能完成时，使用此范围。 TABLE 范围用于表、实例化视图和非实例化视图。 此范围的标识符为对象的全限定名。

另请参见

- “诊断跟踪类型”一节第 175 页
- “诊断跟踪条件”一节第 178 页

诊断跟踪类型

下表列出了可为诊断跟踪选择的跟踪**类型**。所有诊断跟踪类型都存储在 dbo.sa_diagnostic_tracing_level 诊断表的 trace_type 列中，分别需要如下所述的对应条件，并且可能具有相应的诊断跟踪条件（存储在 trace_condition 列中）。有关所有可能条件的列表，请参见“诊断跟踪条件”一节第 178 页。

trace_type 列中的值反映了在 [**数据库跟踪向导**] 中指定的设置。

trace_type 列中的值	说明
VOLATILE_STATISTICS	收集频繁更改数据库和服务器统计信息的样本。 范围和条件：此诊断跟踪类型需要 DATABASE 范围，并使用 SAMPLE_EVERY 条件作为收集数据的间隔。请参见“诊断跟踪范围”一节第 174 页和“诊断跟踪条件”一节第 178 页。

trace_type 列中的值	说明
NONVOLATILE_STATISTICS	收集不经常更改的数据库和服务器统计信息的样本。非易变统计信息的收集频率不能高于易变统计信息的收集频率。只有收集易变统计信息才能收集非易变统计信息，并且非易变统计信息的采样时间间隔应当是为易变统计信息所指定的时间间隔的倍数。 范围和条件：此诊断跟踪类型需要 DATABASE 范围，并使用 SAMPLE_EVERY 条件作为收集数据的间隔。请参见“诊断跟踪范围”一节第 174 页和“诊断跟踪条件”一节第 178 页。
CONNECTION_STATISTICS	收集连接统计信息的样本。如果范围是 DATABASE，则收集到数据库的所有连接的统计信息。如果范围是 USER，则收集指定用户的所有连接的统计信息。如果范围是 CONNECTION_NAME 或 CONNECTION_NUMBER，则仅收集指定连接的统计信息。只有收集易变统计信息才能收集 CONNECTION_STATISTICS，并且采样时间间隔必须是为 VOLATILE_STATISTICS 指定的时间间隔的倍数。 范围和条件：此诊断跟踪类型可与 DATABASE、USER、CONNECTION_NUMBER 和 CONNECTION_NAME 范围一起使用，并使用 SAMPLE_EVERY 条件作为收集数据的间隔。请参见“诊断跟踪范围”一节第 174 页和“诊断跟踪条件”一节第 178 页。
BLOCKING	根据指定的范围和条件收集关于阻塞的信息。如果范围为 CONNECTION_NAME 或 CONNECTION_NUMBER，则当某个连接阻塞另一个连接或这个连接被另一个连接阻塞时，将记录这一阻塞。 范围和条件：此诊断跟踪类型可与所有范围一起使用，并可使用以下任一条件进行收集：NONE、NULL、SAMPLE_EVERY。请参见“诊断跟踪范围”一节第 174 页和“诊断跟踪条件”一节第 178 页。

trace_type 列中的值	说明
PLANS	根据条件和范围，收集查询的执行计划。 范围和条件：此诊断跟踪类型可与所有范围一起使用，并可使用以下任一条件进行收集：NONE、NULL、SAMPLE_EVERY 和 ABSOLUTE_COST。请参见“诊断跟踪范围”一节第 174 页和“诊断跟踪条件”一节第 178 页。
PLANS_WITH_STATISTICS	收集带有执行统计信息的计划。计划将在游标关闭时被记录。如果指定了 RELATIVE_COST_DIFFERENCE 条件，则输出的部分统计信息可能是最佳推测统计信息。 范围和条件：此诊断跟踪类型可与所有范围一起使用，并可使用任一条件进行收集。
STATEMENTS	收集符合指定范围和条件的 SQL 语句。当每个过程第一次执行时，都要收集内部变量。如果指定了 STATEMENTS_WITH_VARIABLES、PLANS、PLANS_WITH_STATISTICS、OPTIMIZATION_LOGGING 或 OPTIMIZATION_LOGGING_WITH_PLANS 诊断跟踪类型，则会自动包含此诊断跟踪类型。 范围和条件：此诊断跟踪类型可与所有范围一起使用，并可使用任一条件进行收集：请参见“诊断跟踪范围”一节第 174 页和“诊断跟踪条件”一节第 178 页。
STATEMENTS_WITH_VARIABLES	收集 SQL 语句和附加在语句上的变量。对于每个变量（内部变量或主机变量），也将收集所有为其指派的值。 范围和条件：此诊断跟踪类型可与所有范围一起使用，并可使用任一条件进行收集：请参见“诊断跟踪范围”一节第 174 页和“诊断跟踪条件”一节第 178 页。

trace_type 列中的值	说明
OPTIMIZATION_LOGGING	收集关于优化程序为执行每个查询而考虑的连接策略的数据。收集关于每个策略的执行开销的信息，以及重新构造结构树所必需的基本信息。还会收集应用于查询的重写的信息。如果所使用的范围不是 DATABASE、CONNECTION_NAME、CONNECTION_NUMBER、ORIGIN 或 USER，则第一个记录的语句文本可能会和查询的初始文本不同，因为在能够确定是否应将优化记录应用于当前语句之前，可能会应用某些重写。只要指定了 OPTIMIZATION_LOGGING_WITH_PLANS 跟踪类型，便会自动添加此诊断跟踪类型。 此诊断跟踪类型对应所有范围，并且不使用任何条件。请参见“诊断跟踪范围”一节第 174 页。
OPTIMIZATION_LOGGING_WITH_PLANS	收集关于优化程序所考虑的连接策略的数据。收集关于每个策略的执行开销的信息，以及描述此连接策略树形结构的完整的 XML 计划。还会收集应用于查询的重写的信息。如果所使用的范围不是 DATABASE、CONNECTION_NAME、CONNECTION_NUMBER、ORIGIN 或 USER，则第一个记录的语句文本可能会和查询的初始文本不同，因为在能够确定是否应将优化记录应用于当前语句之前，可能会应用某些重写。只要指定了 OPTIMIZATION_LOGGING_WITH_PLANS 跟踪类型，便会自动添加 OPTIMIZATION_LOGGING 跟踪类型。 此诊断跟踪类型对应所有范围，并且不使用任何条件。请参见“诊断跟踪范围”一节第 174 页。

另请参见

- “诊断跟踪范围”一节第 174 页
- “诊断跟踪条件”一节第 178 页

诊断跟踪条件

下表列出了可以设置的诊断跟踪**条件**。这些条件控制着为形成特定诊断跟踪类型的跟踪条目而必须满足的条件。大多数条件需要一个值，如下所述。条件存储在 dbo.sa_diagnostic_tracing_level 诊断表的 trace_condition 列中，并可能具有相应的值（如以毫秒为单位的时间量），这些值存储在 value 列中。condition 列中的值反映了 [**数据库跟踪向导**] 中指定的设置。

trace_condition 列中的值	说明
NONE 或 NULL	记录满足级别和范围要求的所有跟踪数据。建议不要将高开销的诊断跟踪级别（例如，计划）与此条件一起使用（因为需要较长时间段）。
SAMPLE_EVERY	如果从记录了上一次事件后，经过的时间超过指定的时间间隔，则记录满足级别和范围要求的跟踪数据。 值：此条件采用一个正整数（以毫秒为单位来反映时间）。
ABSOLUTE_COST	记录那些执行开销大于或等于指定值的语句。 值：此条件采用一个开销值（以毫秒为单位指定）。
RELATIVE_COST_DIFFERENCE	记录其预期执行时间和实际执行时间之间的差值大于或等于指定值的那些语句。 值：此条件采用一个开销值（以百分比形式指定）。例如，要记录比预计执行时间至少慢两倍的语句，可指定值 200。

另请参见

- “诊断跟踪范围”一节第 174 页
- “诊断跟踪类型”一节第 175 页

确定当前诊断跟踪设置

使用 Sybase Central 中的 [**数据库跟踪向导**] 来查看当前诊断跟踪设置。完成设置检查后，取消该向导。还可以通过查询 sa_diagnostic_tracing_level 表来检索有效的诊断跟踪设置。

无论跟踪会话是否正在运行，您都可以检索诊断跟踪设置。

♦ 确定当前诊断跟踪设置 (Sybase Central)

1. 以具有 DBA 特权或 PROFILE 特权的用户身份连接到数据库。
2. 选择 [**模式**] » [**应用程序分析**]。如果出现 [**应用程序分析向导**]，则单击 [**取消**]。
3. 在左窗格中，右击数据库然后选择 [**跟踪**]。

   如果 [**数据库跟踪向导**] 未出现，则选择 [**跟踪**] » [**配置**]。
4. 在 [**编辑跟踪级别**] 列表中查看当前为诊断跟踪指定的设置。
5. 单击 [**取消**]。

♦ 确定当前诊断跟踪设置 (Interactive SQL)

1. 以具有 DBA 特权或 PROFILE 特权的用户身份连接到数据库。
2. 查询 sa_diagnostic_tracing_level 表，找到 enabled 列为 1 的行。

数据库服务器将返回当前使用的诊断跟踪设置。enabled 列中的 1 指明设置有效。

示例

以下语句显示如何查询 sa_diagnostic_tracing_level 诊断表以检索当前诊断跟踪设置：

```
SELECT * FROM sa_diagnostic_tracing_level WHERE enabled = 1;
```

下表是查询的一个示例结果集：

id	scope	identifier	trace_type	trace_condition	value	enabled
1	database	(NULL)	volatile_statistics	sample_every	1,000	1
2	database	(NULL)	nonvolatile_statistics	sample_every	60.000	1
3	database	(NULL)	connection_statistics	(NULL)	60,000	1
4	database	(NULL)	阻塞	(NULL)	(NULL)	1
5	database	(NULL)	死锁	(NULL)	(NULL)	1
6	database	(NULL)	plans_with_statistics	sample_every	2,000	1

另请参见

- “sa_diagnostic_tracing_level 表”一节 《SQL Anywhere 服务器 - SQL 参考》
- “PROFILE 特权”一节 《SQL Anywhere 服务器 - 数据库管理》

更改诊断跟踪配置设置

诊断跟踪设置特定于某一生产数据库。创建跟踪会话时，您可以使用 Sybase Central 中的 [**数据库跟踪向导**] 来更改诊断跟踪设置。要了解如何启动 [**数据库跟踪向导**]，请参见 “创建诊断跟踪会话”一节第 181 页。

[**数据库跟踪向导**] 中所配置的诊断跟踪设置不会影响 [**应用程序分析向导**] 的设置或行为。[**应用程序分析向导**] 的设置已经过预配置，且无法更改。

您还可以使用 sa_set_tracing_level 系统过程来更改诊断跟踪级别。此方法不会启动跟踪会话，并且如果跟踪会话已经运行，该操作会失败。并且，利用此方法您无法像前一种方法那样控制其它设置（如范围、条件、值等等）。有关此过程的详细信息，请参见“sa_set_tracing_level 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》。

♦ 更改诊断跟踪级别 (Interactive SQL)

1. 以具有 DBA 特权或 PROFILE 特权的用户身份连接到数据库。
2. 使用 sa_set_tracing_level 系统过程来设置诊断跟踪级别。

示例

以下语句使用 sa_set_tracing_level 系统过程将诊断跟踪级别设置为 1：

```
CALL sa_set_tracing_level( 1 );
```

与诊断跟踪级别 1 关联的缺省设置会覆盖现有的设置。要查看与各个诊断跟踪级别相关联的缺省设置，请参见“诊断跟踪级别”一节第 173 页。

在跟踪会话运行过程中更改诊断跟踪设置

在跟踪会话运行过程中，您可以使用 Sybase Central 中的 [**数据库跟踪向导**] 来更改诊断跟踪设置。

♦ **在跟踪会话运行期间更改诊断跟踪设置 (Sybase Central)**

1. 以具有 DBA 特权或 PROFILE 特权的用户身份连接到数据库。
2. 在左窗格中，右击数据库然后选择 [**跟踪**] » [**更改跟踪级别**]。
3. 添加新的跟踪级别或删除现有跟踪级别。
4. 单击 [**确定**]。

创建诊断跟踪会话

启动诊断跟踪会话时，您还可配置想要执行的跟踪类型，并指定存储跟踪数据的位置。您的跟踪会话将一直运行，直到您显式要求该会话停止。

要启动跟踪会话，TCP/IP 必须在运行跟踪数据库和生产数据库的数据库服务器上运行。请参见“使用 TCP/IP 协议”一节 《SQL Anywhere 服务器 - 数据库管理》。

注意
启动跟踪会话也称作附加跟踪。同样，停止跟踪会话也称作分离跟踪。启动和停止跟踪的 SQL 语句分别为 ATTACH TRACING 和 DETACH TRACING。

♦ **创建诊断跟踪会话 (Sybase Central)**

1. 以具有 DBA 特权或 PROFILE 特权的用户身份连接到数据库。
2. 右击数据库并选择 [**跟踪**]。
3. 单击 [**下一步**]。
4. 在 [**跟踪详细信息级别**] 页中，选择跟踪级别。
5. 在 [**编辑跟踪级别**] 页中，自定义诊断跟踪设置。
6. 在 [**创建外部数据库**] 页中：
   - 选择 [**创建新的跟踪数据库**]。

- 选择一个位置来保存该数据库。
- 填写 [**用户名**] 和 [**口令**] 字段。
- 选择 [**在当前服务器上启动数据库**]。
- 单击 [**创建数据库**]。

7. 在 [**启动跟踪**] 页中：
   - 选择 [**在外部数据库中保存跟踪数据**]。
   - 填写 [**用户名**] 和 [**口令**] 字段。指定用来连接到生产数据库的用户名和口令。
   - 在 [**其它连接参数**] 字段中键入数据库服务器和数据库名（以部分连接字符串的形式）。例如，`ENG=Server47;DBN=TracingDB`

     注意
     外部数据库的连接字符串只支持 DBN、DBF、ENG、DBKEY 和 LINKS (CommLinks)。

   - 在 [**是否限制存储跟踪数据的量**] 列表中，选择一个选项。
8. 单击 [**完成**]。
9. 完成对诊断跟踪数据的收集后，右击该数据库并选择 [**跟踪**] » [**停止跟踪并保存**]。

♦ 创建诊断跟踪会话 (Interactive SQL)

1. 以 DBA 身份或具有 PROFILE 特权的用户身份连接到数据库。
2. 使用 sa_set_tracing_level 系统过程来设置跟踪级别。
3. 通过执行 ATTACH TRACING 语句来启动跟踪。
4. 通过执行 DETACH TRACING 语句来停止跟踪。

   您可在 Sybase Central 的 [应用程序分析] 模式下查看诊断跟踪数据。请参见“应用程序分析”一节第 161 页。

示例

以下示例说明如何对当前数据库启动诊断跟踪，如何将跟踪数据存储在单独的数据库中，以及如何对要存储的数据量设置两个小时的限制。此示例全部在一行上输入：

```
ATTACH TRACING TO
'UID=DBA;PWD=sql;ENG=dbsrv11;DBN=tracing;LINKS=tcpip' LIMIT HISTORY 2 HOURS;
```

以下示例说明如何对当前数据库启动诊断跟踪，如何将跟踪数据存储在本地数据库中，以及如何对要存储的数据量设置两兆字节的限制：

```
ATTACH TRACING TO LOCAL DATABASE LIMIT SIZE 2 MB;
```

以下示例说明如何停止诊断跟踪并保存跟踪会话期间捕获的诊断数据：

```
DETACH TRACING WITH SAVE;
```

以下示例说明如何停止诊断跟踪但不保存诊断数据。

```
DETACH TRACING WITHOUT SAVE;
```


另请参见

- “ATTACH TRACING 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “DETACH TRACING 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “sa_set_tracing_level 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》
- “PROFILE 特权”一节 《SQL Anywhere 服务器 - 数据库管理》

分析诊断跟踪信息

诊断跟踪数据记录了发生在数据库服务器上的、与各诊断跟踪级别和跟踪会话设置相对应的所有活动。在查看数据时，必须考虑所使用的设置。例如，在诊断会话中找不到期望看到的语句可能说明该语句从未运行过，也可能说明该语句开销不够大，无法满足仅跟踪大开销语句这样一个条件。

出于许多原因，您可能希望详细检查数据库服务器正在执行的活动。这些原因包括解决性能问题、预计未来负载计划的资源使用，以及调试应用程序逻辑。

另请参见

- “应用程序分析教程”第 229 页

解决性能问题

可使用应用程序分析功能来确定性能问题是否由以下原因引起：

- 应用程序处理时间过长
- 糟糕的查询计划
- 共享硬件资源（如 CPU 或磁盘 I/O）争用
- 数据库对象争用
- 次优的数据库设计

在解决数据库性能较差的问题时，首要任务是确定主要原因是应用程序还是数据库服务器。要确定客户端应用程序所消耗的处理时间，可使用应用程序分析工具中的 [**详细信息**] 选项卡并按单个连接过滤结果。如果来自该连接的不同请求之间存在时间差异，则主要的延迟发生在应用程序客户端内。

如果是数据库服务器影响了性能，则需要确定具体原因。

另请参见

- “应用程序分析教程”第 229 页

检测何时硬件资源成为一个限制因素

随着数据库承受越来越多的负载，性能通常受到 CPU 周期、内存空间或磁盘 I/O 带宽的限制。应用程序或数据库服务器效率低下可能是性能降低的原因。如果无法检测到效率低下问题，可能需要

增加额外的硬件资源。要查看常见效率低下问题的列表以及解决这些问题的建议，请参见“解决性能问题”一节第 183 页。

增加资源可能无法解决可伸缩性问题或提高计算机性能。例如，如果数据库服务器已经完全占用所有分配的 CPU，这可能表明需要分配更多的 CPU 资源。但是，将可用于数据库服务器的 CPU 的数量加倍，不一定表示数据库服务器可执行的工作量会加倍。

可以使用 [**应用程序分析详细信息**] 区域的 [**统计**] 选项卡检测硬件资源是否为性能的一个限制因素。

- **检测 CPU 是否为限制因素** 要检测 CPU 是否为限制因素，请检查 ProcessCPU 统计信息。如果图表中未出现此统计信息，则单击 [**添加统计信息**] 按钮并选择 [**ProcessCPU**]。如果图表中出现以接近每 CPU（指派给数据库服务器）每秒 1 点的速率增加的 ProcessCPU，则 CPU 是一个限制因素。例如，对于在两个 CPU 上运行的数据库服务器，如果 Process CPU 计数器在十秒之内从 2220 增长到 2237，这表明这十秒期间 CPU 的使用率为 (2237-2220) / 10s * 100 % = 170%，也就是说每个 CPU 以其容量的 170% / 2 = 85% 运行。
- **检测内存是否为限制因素** 要检测内存（缓冲池的大小）是否为限制因素，请检查 CacheHits 和 CacheReads 数据库统计信息。如果图表中未出现这些统计信息，则单击 [**添加统计信息**] 按钮并选择 **[CacheHits] 和 [CacheReads]**。如果 CacheHits 小于 CacheReads 的 10%，这说明缓冲池太小。如果比率在 10-70% 之间，这可能说明缓冲池太小—您应当尝试增加数据库服务器的高速缓存大小。如果比率高于 70%，则高速缓存大小可能适当。注意，此策略仅适用于数据库服务器在稳定状态下运行—即，它正在处理标准负载且不是刚刚启动。
- **检测 I/O 带宽是否为限制因素** 要检测 I/O 带宽是否为限制因素，请检查 CurrIO 数据库统计信息。如果图表中未出现此统计信息，则单击 [**添加统计信息**] 按钮并选择 [**CurrIO**]。寻找此统计信息的最大持续值。例如，在图表中寻找高平稳段；它越宽影响就越显著。如果该图表包含一些持续值，它们等于或大于 3 加上数据库服务器使用的物理磁盘数目的和，则可能说明磁盘系统无法跟上数据库服务器活动的级别。

另请参见

- “性能监控器统计”一节第 197 页
- “应用程序分析教程”第 229 页
- “解决性能问题”一节第 183 页

调试应用程序逻辑

如果在应用程序代码中或在存储的过程、触发器、函数或事件中存在错误，则检查由数据库服务器执行且与不正确的代码相关的所有语句可能非常有用。对于动态生成 SQL 的应用程序，您可以检查数据库服务器所看到的实际文本，以检测应用程序构建 SQL 文本方法中的错误。此类错误可能导致查询执行失败，或返回与查询要返回的结果不同的结果。例如，在开发期间，您的应用程序可能偶而报告遇到 SQL 语法错误，但是您的应用程序并未编制为报告失败查询的 SQL 文本。如果在应用程序运行时进行跟踪，则可以查找返回语法（或其它）错误的语句，并查看应用程序所生成的确切文本。

对于内部数据库对象（如过程和触发器），可以使用 Sybase Central 中的调试程序。但有时这样做可能会更有效：让数据库服务器跟踪给定过程执行的所有语句，然后使用应用程序分析工具来检查这些语句。例如，给定的存储过程调用 1000 次可能返回一个错误结果，但您并不清楚在什么条件下出现了故障。此时您应对该过程启用诊断跟踪并运行应用程序，而不是在调试程序中单步执行

1000 次过程代码。然后，您可以检查数据库服务器执行的语句集，查找与不正确的过程执行相对应的语句集，并确定过程失败的原因或在何种条件下过程出现异常行为。如果您知道在何种条件下过程出现异常行为，您就可以在过程中设置断点并使用调试程序进一步查找原因。请参见“调试过程、函数、触发器和事件”第 827 页。

执行请求跟踪分析

当您的特定应用程序或请求出现问题时，您可以执行请求跟踪分析来确定出现了何种问题。请求跟踪分析涉及到配置 [**数据库跟踪向导**]，以将诊断数据的收集范围缩小到仅为遇到问题的用户、连接或请求。然后使用 [应用程序分析] 模式中的各种数据查看工具，确定任何可能的冲突或瓶颈。

- **执行请求跟踪分析**

1. 以具有 DBA 特权或 PROFILE 特权的用户身份连接到数据库。
2. 选择 [**模式**] » [**应用程序分析**]。如果出现 [**应用程序分析向导**]，则单击 [**取消**]。
3. 右击数据库并选择 [**跟踪**] 或选择 [**跟踪**] » [**配置和启动跟踪**]。
4. 请按照 [**数据库跟踪向导**] 中的说明进行操作。
5. 完成对跟踪数据的收集后，右击该数据库并选择 [**跟踪**] » [**停止跟踪并保存**]。
6. 在 [**应用程序分析详细信息**] 窗格中，单击 [**打开分析文件或连接到跟踪数据库**]。
7. 选择 [**在跟踪数据库中**]，然后单击 [**打开**]。
8. 填写 [**用户名**] 和 [**口令**] 字段，然后单击 [**确定**]。
9. 在 [**应用程序分析详细信息**] 窗格中，选择 [**记录会话 ID**] 列表中的最后一个条目。
10. 单击 [**应用程序分析详细信息**] 窗格底部的 [**数据库跟踪数据**] 选项卡。

    您可以从几个提供了所收集数据不同视图的选项卡中进行选择，以进行分析。例如，[**概览**] 选项卡允许您查看跟踪会话期间对数据库执行的所有请求，其中包括每个请求执行的次数、执行持续时间、执行该请求的用户等。如果列表较长而您要查找某个特定请求，请单击 [**概览**] 选项卡上的 [**过滤**] 标题栏，并在 [**SQL 语句包含**] 字段中输入某一字符串。

    要查看某个特定请求的详细信息，可右击该请求并选择 [**显示所选概览语句的详细 SQL 语句**]。随即打开 [**详细信息**] 选项卡。右击包含该请求的行，可对信息进行其它选择，包括查看其它 SQL 语句、连接和阻塞的详细信息。

创建外部跟踪数据库

创建跟踪会话时，您可以选择将跟踪数据存储在被分析的数据库中。这适用于在其中对应用程序进行测试的开发环境，或到数据库的连接很少的情况。但是，如果您的数据库通常在任何给定的时间都要处理 10 个或更多的连接，则建议将跟踪数据存储在外部跟踪数据库中，以降低对性能的影响。

启动跟踪会话时，可使用 [**数据库跟踪向导**] 创建外部跟踪数据库。[**数据库跟踪向导**] 会从生产数据库卸载模式和权限信息。可以使用跟踪数据库存储后续跟踪会话所用的数据。有关创建跟踪会话的信息，请参见“创建诊断跟踪会话”一节第 181 页。

可使用卸载实用程序 (dbunload) 在没有跟踪会话的情况下手工创建跟踪数据库。

♦ 使用卸载实用程序 (dbunload) 创建外部跟踪数据库

1. 以具有 DBA 特权或 PROFILE 特权的用户身份连接到数据库。
2. 执行 dbunload 命令（类似下面的内容），将模式从生产数据库卸载至新的跟踪数据库中：

   ```
   dbunload -c "UID=DBA;PWD=sql;ENG=demo;DBN=demo" -an tracing.db -n -k
   ```

   本示例创建一个新的数据库，它具有由 -an 选项提供的名称 (*tracing.db*)。-n 选项将模式从所分析的数据库（在本例中为 SQL Anywhere 示例数据库 *demo.db*）卸载至新的跟踪数据库中。-k 选项使用应用程序分析工具用来分析跟踪数据的信息来填充跟踪数据库。
3. 如果您想将跟踪数据库存储在不同的计算机上，可将其复制到新位置。

另请参见

- “卸载实用程序 (dbunload)”一节 《SQL Anywhere 服务器 - 数据库管理》
- “PROFILE 特权”一节 《SQL Anywhere 服务器 - 数据库管理》

其它诊断工具和技术

除了应用程序分析和诊断跟踪，还有多种其它的诊断工具和技术可以帮助您分析和监控 SQL Anywhere 数据库的当前性能。

请求记录

请求记录会记录从应用程序接收的各个请求以及发送到应用程序的响应。请求记录在确定应用程序要求数据库服务器所执行的操作时用处最大。

在不清楚是数据库服务器还是客户端有故障时，要对特定应用程序进行性能分析，也最好先从请求记录开始。可以使用请求记录来确定问题的根源是否是向数据库服务器发出特定请求。

注意
请求记录功能提供的所有功能和数据也可使用诊断跟踪来获得。诊断跟踪还可提供其它的功能和数据。请参见“使用诊断跟踪进行高级应用程序分析”一节第 171 页。

所记录的信息包括时间戳、连接 ID 和请求类型等。对于查询，记录的信息还包括隔离级别、读取的行数和游标类型。对于 INSERT、UPDATE 和 DELETE 语句，记录的信息还包括受到影响的行数以及触发的触发器数。

小心
请求日志可能包含敏感信息，这是因为请求日志中包含了含有口令的 SQL 语句（如 GRANT CONNECT、CREATE DATABASE 和 CREATE EXTERNAL LOGIN 语句）的完整文本。如果考虑到安全性，您就应当限制对请求日志文件的访问。

当启动数据库服务器时，您可以使用 -zr 服务器选项来启动请求记录。使用 -zo 服务器选项，您可以将输出重定向到一个请求日志文件，以便进一步分析。-zn 和 -zs 选项允许您指定要保存的请求日志文件数以及请求日志文件的最大大小。

有关这些选项的详细信息，请参见：

- “-zr 服务器选项”一节 《SQL Anywhere 服务器 - 数据库管理》
- “-zo 服务器选项”一节 《SQL Anywhere 服务器 - 数据库管理》
- “-zn 服务器选项”一节 《SQL Anywhere 服务器 - 数据库管理》
- “-zs 服务器选项”一节 《SQL Anywhere 服务器 - 数据库管理》

注意
这些服务器选项不会影响 Sybase Central 中的诊断跟踪。基于文件的请求记录与 Sybase Central 中的诊断跟踪功能毫无关系，后者使用数据库中属于 dbo 的诊断表来存储请求日志信息。

sa_get_request_times 系统过程读取请求日志，并用日志中的语句和它们的执行时间填充全局临时表 (satmp_request_time)。对于 INSERT/UPDATE/DELETE 语句，记录的时间是执行语句时的时间。对于查询，记录的时间是从 PREPARE 到 DROP（描述/打开/读取/关闭）所经过的总时间。这意味着您需要知道任何打开的游标。

分析 satmp_request_time 以找到可能需要改进的语句。开销低但经常执行的语句可能会出现性能问题。

您可以使用 sa_get_request_profile 调用 sa_get_request_times，并将 satmp_request_time 汇总到另一个名为 satmp_request_profile 的全局临时表中。此过程还可将语句组合在一起，并提供调用次数、执行时间，等等。请参见“sa_get_request_times 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》和“sa_get_request_profile 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》。

过滤请求日志

使用 sa_server_option 系统过程，可以对输出到请求日志的内容进行过滤，以便只包括来自特定连接或特定数据库的请求。在监控具有多个活动连接或多个数据库的数据库服务器时，这有助于减小日志的大小。请参见“sa_server_option 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》。

♦ 根据连接过滤

- 使用以下语法：

```
CALL sa_server_option( 'RequestFilterConn' , connection-id );
```

可以通过执行 `CALL sa_conn_info( );` 获得 *connection-id*。

♦ 根据数据库过滤

- 使用以下语法：

```
CALL sa_server_option( 'RequestFilterDB' , database-id );
```

当连接到数据库时，通过执行 `SELECT CONNECTION_PROPERTY( 'DBNumber' )` 可以获得 *database-id*。在显式重置过滤或关闭数据库服务器之前，过滤将一直有效。

♦ 重置过滤

- 使用下面两个语句中的任何一个，按连接或按数据库重置过滤：

```
CALL sa_server_option( 'RequestFilterConn' , -1 );
CALL sa_server_option( 'RequestFilterDB' , -1 );
```

将主机变量输出到请求日志

主机变量值可以输出到请求日志。

♦ 包括主机变量值

- 要在请求日志中包括主机变量值：
  - 使用具有值 **hostvars** 的服务器选项 -zr
  - 执行以下语句：

```
CALL sa_server_option( 'RequestLogging' , 'hostvars' );
```

请求日志分析过程 sa_get_request_times 可识别日志中的主机变量，并将它们添加到全局临时表 satmp_request_hostvar 中。

使用系统过程进行过程分析

过程分析可提供关于所有连接使用的存储过程、用户定义的函数、事件、系统触发器和触发器的用法的重要信息。可在 Sybase Central 或 Interactive SQL 中使用系统过程调用来执行过程分析。Sybase Central 提供了更强大的功能和更大的灵活性来帮助您执行过程分析。出于这个原因，建议您使用 Sybase Central 的 [应用程序分析] 模式中提供的过程分析功能来执行过程分析。请参见“[应用程序分析] 模式中的过程分析”一节第 162 页。

使用 sa_server_option 启用分析

♦ 在 Interactive SQL 中启用过程分析

1. 以具有 DBA 特权或 PROFILE 特权的用户身份连接到数据库。
2. 调用 sa_server_option 系统过程，将 ProcedureProfiling 选项设置为 ON。

   例如，输入以下内容：

```
CALL sa_server_option( 'ProcedureProfiling' , 'ON' );
```

如有必要，您可以查看特定用户正在使用的过程，并且不会阻止其它连接使用数据库。当连接已存在或有多个用户使用同一用户 ID 进行连接时，这是非常有用的。

♦ 在 Interactive SQL 中按用户过滤过程分析

1. 以 DBA 身份或具有 PROFILE 特权的用户身份连接到数据库。
2. 如下所示调用 sa_server_option 系统过程：

```
CALL sa_server_option( 'ProfileFilterUser' , 'userid' );
```

   userid 的值是当前被监控用户的名称。

另请参见

- “sa_server_option 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》

使用 sa_server_option 重置分析

当您重置分析后，数据库会清除旧信息，并立即开始收集有关过程、函数、事件和触发器的新信息。此节将说明如何使用 sa_server_option 系统过程来从 Interactive SQL 重置过程分析。

下面几节假定您已经作为 DBA 或拥有 PROFILE 特权的用户连接到数据库，并且过程分析已启用。

♦ 在 Interactive SQL 中重置分析

- 调用 sa_server_option 系统过程，将 ProcedureProfiling 选项设置为 RESET。

  例如，输入以下内容：

```
CALL sa_server_option( 'ProcedureProfiling' , 'RESET' );
```

另请参见

- “sa_server_option 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》

使用 sa_server_option 禁用分析

当收集完分析信息后，可以禁用分析或清除分析。如果禁用分析，则数据库将停止收集分析信息，而在此刻之前收集的信息会留在 Sybase Central 中的 [分析] 选项卡上。如果清除分析，数据库将关闭分析功能，并且清除 Sybase Central 中 [分析] 选项卡上的所有分析数据。此节将说明如何使用 sa_server_option 系统过程来从 Interactive SQL 禁用过程分析。

♦ **禁用分析 (Interactive SQL)**

- 调用 sa_server_option 系统过程，将 ProcedureProfiling 选项设置为 OFF。

  例如，输入以下内容：

  ```
  CALL sa_server_option( 'ProcedureProfiling' , 'OFF' );
  ```

♦ **禁用分析并清除现有数据 (Interactive SQL)**

- 调用 sa_server_option 系统过程，将 ProcedureProfiling 选项设置为 CLEAR。

  例如，输入以下内容：

  ```
  CALL sa_server_option( 'ProcedureProfiling' , 'CLEAR' );
  ```

另请参见

- “sa_server_option 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》

使用系统过程检索分析信息

您可以使用系统过程来查看以下对象的过程分析信息：储存过程、函数、事件、系统触发器和触发器。过程分析也必须已经启用。请参见“使用 sa_server_option 启用分析”一节第 189 页。

sa_procedure_profile 系统过程显示详细的分析信息，包括每个对象中行的执行时间；结果集中的每一行表示对象中的一个可执行代码行。

sa_procedure_profile_summary 系统过程显示每个对象的总执行时间，同时给出运行的所有对象的摘要；结果集中的每一行表示一个对象的执行详细信息。

当查看这些系统过程返回的结果时，所列出的对象可能比明确调用的对象要多。这是因为一个对象可以调用另一个对象。例如，触发器可能会调用一个存储过程，而该存储过程又调用另一个存储过程。

♦ 查看摘要分析信息 (Interactive SQL)

1. 以具有 DBA 特权或 PROFILE 特权的用户身份连接到数据库。
2. 执行 sa_procedure_profile_summary 系统过程。

   例如，输入以下内容：

   ```
   CALL sa_procedure_profile_summary;
   ```
3. 选择 [**SQL**] » [**执行**]。

   [**结果**] 窗格中会出现一个结果集，此结果集包含有关数据库中所有过程的信息。

♦ 查看详细分析信息 (Interactive SQL)

1. 以具有 DBA 特权或 PROFILE 特权的用户身份连接到数据库。
2. 执行 sa_procedure_profile 系统过程。

   例如，输入以下内容：

   ```
   CALL sa_procedure_profile;
   ```
3. 选择 [**SQL**] » [**执行**]。

   [**结果**] 窗格中会显示一个包含分析信息的结果集。

另请参见

- “sa_procedure_profile_summary 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》
- “sa_procedure_profile 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》
- “sa_server_option 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》

计时实用程序

samples-dir\SQLAnywhere 中提供了一些性能测试实用程序，包括 fetchtst、instest 和 trantest。有关 *samples-dir* 位置的详细信息，请参见“示例目录”一节 《SQL Anywhere 服务器 - 数据库管理》。

Fetchtst 实用程序用于测量任意查询的读取速率。Instest 实用程序确定将行插入表中所需的时间。Trantest 实用程序用于测量在数据库设计和事务组既定的情况下，某一给定服务器配置能够处理的负载。

与含统计信息的图形式计划相比，这些工具可以提供更准确的计时，并可以指出给定的服务器和数据库配置可达到的最佳性能（例如吞吐量）。

在实用程序所在文件夹中的 *readme.txt* 文件中，可以找到这些工具的完整文档。

监控数据库性能

SQL Anywhere 提供了一组统计信息，可用来监控数据库性能。访问这些统计信息的方法有三种：

- **SQL 函数** 利用这些函数，应用程序可直接访问 SQL Anywhere 数据库统计信息。请参见“使用 SQL 函数监控统计信息”一节第 192 页。
- **Sybase Central 性能监控器** 此图形工具用于查询数据库，并仅将那些您已配置性能监控器对其进行绘制的统计信息绘制成图表。请参见“使用 Sybase Central 性能监控器监控统计信息”一节第 193 页。
- **Windows 性能监控器** 这是 Windows 操作系统提供的监控工具。请参见“使用 Windows 性能监控器监控统计信息”一节第 195 页。
- **SQL Anywhere 控制台实用程序 (dbconsole)** 该实用程序用于为数据库服务器连接提供管理和监控工具。请参见“SQL Anywhere 控制台实用程序 (dbconsole)”一节 《SQL Anywhere 服务器 - 数据库管理》。

这些方法可用于实时监控。但是，也可以作为诊断跟踪的一部分捕获统计信息，并保存这些信息，以便以后进行分析。有关诊断跟踪的详细信息，请参见“使用诊断跟踪进行高级应用程序分析”一节第 171 页。

有关可监控的 SQL Anywhere 统计信息的完整列表，请参见“性能监控器统计”一节第 197 页。

使用 SQL 函数监控统计信息

SQL Anywhere 提供了一组系统函数，它们可以访问每个连接、每个数据库或者整个服务器范围的信息。可用信息的类型有多种，其中包括静态信息（如数据库服务器名称）和详细的性能相关统计信息（如磁盘和内存使用情况）。

检索系统信息的函数

以下函数检索系统信息：

- **PROPERTY 函数** 此函数提供给定属性在整个服务器范围的值。请参见“PROPERTY 函数 [System]”一节 《SQL Anywhere 服务器 - SQL 参考》。
- **DB_PROPERTY 和 DB_EXTENDED_PROPERTY 函数** 这两个函数提供给定数据库（缺省情况下为当前数据库）的给定属性的值。请参见“DB_PROPERTY 函数 [System]”一节 《SQL Anywhere 服务器 - SQL 参考》和“DB_EXTENDED_PROPERTY 函数 [System]”一节 《SQL Anywhere 服务器 - SQL 参考》。
- **CONNECTION_PROPERTY 和 CONNECTION_EXTENDED_PROPERTY 函数** 这两个函数提供给定连接（缺省情况下为当前连接）的给定属性的值。请参见“CONNECTION_PROPERTY 函数 [System]”一节 《SQL Anywhere 服务器 - SQL 参考》和“CONNECTION_EXTENDED_PROPERTY 函数 [String]”一节 《SQL Anywhere 服务器 - SQL 参考》。

仅提供您要检索的属性的名称作为参数。这些函数将返回当前服务器、连接或数据库的值。

有关使用系统函数可检索的属性的完整列表，请参见“系统函数”一节 《SQL Anywhere 服务器 - SQL 参考》。

示例

以下语句将名为 server_name 的变量设置为当前服务器的名称：

```
SET server_name = PROPERTY( 'name' );
```

以下查询用于返回当前连接的用户 ID：

```
SELECT CONNECTION_PROPERTY( 'UserID' );
```

以下查询用于返回当前数据库根文件的文件名：

```
SELECT DB_PROPERTY( 'file' );
```

提高查询效率

为了提高性能，监控数据库活动的客户端应用程序应该使用 PROPERTY_NUMBER 函数来标识指定的属性，然后使用该编号重复检索统计信息。

通过这种方法获得的属性名称可用于多种不同的数据库统计信息，包括从已执行的事务日志页写操作的次数和检查点操作的次数，到从内存高速缓存中读取索引叶页的次数。

下面的一组语句说明通过 Interactive SQL 执行的过程：

```
CREATE VARIABLE propnum INT;
CREATE VARIABLE propval INT;
SET propnum = PROPERTY_NUMBER( 'CacheRead' );
SET propval = DB_PROPERTY( propnum );
```

有关 PROPERTY_NUMBER 函数的详细信息，请参见“PROPERTY_NUMBER 函数 [System]”一节 《SQL Anywhere 服务器 - SQL 参考》。

您可以通过 Sybase Central 性能监控器以图形方式查看其中的多种统计信息。请参见“使用 Sybase Central 性能监控器监控统计信息”一节第 193 页。

使用 Sybase Central 性能监控器监控统计信息

Sybase Central 性能监控器可用于跟踪与数据库服务器操作（包括磁盘和内存访问）相关的详细信息。Sybase Central 性能监控器可将任何可连接到的 SQL Anywhere 数据库服务器的统计信息绘制成图表。

Sybase Central 性能监控器的功能包括：

- 实时更新（以可调整的间隔）
- 带颜色代码且大小可调整的图例
- 可配置的外观属性

Sybase Central 性能监控器对数据库进行查询，以收集其统计信息。这可能影响某些统计数据（如高速缓存读取/秒）。如果不希望统计数据受到监控影响，则可以改用 Windows 性能监控器。请参见“使用 Windows 性能监控器监控统计信息”一节第 195 页。

如果您同时运行多个版本的 SQL Anywhere，您也可以同时运行多个版本的性能监控器。

有关可监控的 SQL Anywhere 统计信息的完整列表，请参见“性能监控器统计”一节第 197 页。

打开 Sybase Central 性能监控器

当选择了 [**性能监控器**] 选项卡时，Sybase Central 性能监控器便会出现在 Sybase Central 的右窗格中。图表仅显示您配置其进行显示的那些统计信息。有关在性能监控器图表中添加和删除统计信息的详细信息，请参见“添加和删除统计信息”一节第 194 页。

♦ **打开性能监控器：**

1. 在左窗格中，选择服务器。
2. 在右窗格中，单击 [**性能监控器**] 选项卡。

另请参见

- “使用 Windows 性能监控器监控统计信息”一节第 195 页
- “添加和删除统计信息”一节第 194 页

添加和删除统计信息

♦ **将统计信息添加到 Sybase Central 性能监控器：**

1. 在左窗格中，选择服务器。
2. 在右窗格中，单击 [**统计**] 选项卡。
3. 右击当前未被监控的统计信息，然后选择 [**添加到性能监控器**]。

♦ **从 Sybase Central 性能监控器中删除统计信息：**

1. 在左窗格中，选择服务器。
2. 在右窗格中，单击 [**统计**] 选项卡。
3. 右击当前正被监控的统计信息，然后选择 [**从性能监控器中删除**]。

提示
也可以在统计信息的属性窗口中将统计信息添加到性能监控器或从性能监控器中删除统计信息。

有关可监控的 SQL Anywhere 统计信息的完整列表，请参见“性能监控器统计”一节第 197 页。

另请参见

- “打开 Sybase Central 性能监控器”一节第 194 页
- “使用 Windows 性能监控器监控统计信息”一节第 195 页

使用 Windows 性能监控器监控统计信息

除了使用 Sybase Central 性能监控器之外，您也可以使用 Windows 性能监控器。

与 Sybase Central 性能监控器相比，Windows 性能监控器可提供更多的性能统计信息（尤其是网络通信统计信息）。它还使用共享内存方案，而不是对数据库服务器执行查询，因此不会对统计信息本身造成影响。

Windows 提供了 Windows 性能监控器。如果您同时运行多个版本的 SQL Anywhere，也可以同时运行多个版本的性能监控器。

有关可监控的 SQL Anywhere 性能统计信息的完整列表，请参见“性能监控器统计”一节 第 197 页。

在启动用于控制 Windows 性能监控器所使用的内存的数据库服务器时，可以指定若干数据库服务器选项，以及性能监控器可以监控的最大连接数和最大数据库数。请参见：

- “-ks 服务器选项”一节 《SQL Anywhere 服务器 - 数据库管理》
- “-ksc 服务器选项”一节 《SQL Anywhere 服务器 - 数据库管理》
- “-ksd 服务器选项”一节 《SQL Anywhere 服务器 - 数据库管理》

下面介绍如何在 Windows XP 上启动 Windows 性能监控器。对于其它版本的 Windows，请查阅 Windows 操作系统文档，以获得有关如何启动 Windows 性能监控器的信息。

♦ 在 Windows XP 上使用 Windows 性能监控器

1. 当 SQL Anywhere 服务器正在运行时，启动性能监控器：
   - 从 Windows [**控制面板**] 中，选择 [**管理工具**]。
   - 选择 [**性能**]。
2. 在工具栏上，单击加号工具 (+)。
3. 从 [**性能对象**] 列表中选择以下对象之一：
   - **SQL Anywhere 11 连接**　它用于监控单个连接的性能。当前必须存在一个连接才能看到此选项。
   - **SQL Anywhere 11 数据库**　这将监控单个数据库的性能。
   - **SQL Anywhere 11 服务器**　这将监控整个服务器范围的性能。

   [**计数器**] 框显示您可以查看的统计信息的列表。

   如果您选择了 [SQL Anywhere 连接] 或 [SQL Anywhere 数据库]，则 [**范例**] 框会显示那些可查看其统计信息的连接或数据库的列表。
4. 在 [**计数器**] 列表中，单击要查看的统计信息。若要选择多项统计信息，请在单击的同时按住 Ctrl 或 Shift 键。
5. 如果您已经选择 [SQL Anywhere 11 连接] 或 [SQL Anywhere 11 数据库]，请从 [**范例**] 框中选择要监控的数据库连接或数据库。
6. 要获得所选计数器的说明，请单击 [**说明**]。

7. 要显示该计数器，请单击 [**添加**]。
8. 当选择完要显示的所有计数器后，单击 [**关闭**]。

性能监控器统计

SQL Anywhere 提供以下统计信息：

速率为每秒报告一次。

高速缓存统计

以下统计信息描述了高速缓存的使用情况。

统计信息	作用域	说明
高速缓存命中/秒	连接和数据库	显示在高速缓存中找到页的数据库页查找的速率。
高速缓存读取:索引内部/秒	连接和数据库	显示从高速缓存中读取索引内部节点页的速率。
高速缓存读取:索引叶/秒	连接和数据库	显示从高速缓存中读取索引叶页的速率。
高速缓存读取:表/秒	连接和数据库	显示从高速缓存中读取表页面的速率。
高速缓存读取:总页数/秒	连接和数据库	显示从高速缓存中查找数据库页面的速率。
高速缓存读取:工作表	连接和数据库	显示当前从高速缓存中读取工作表页面的速率。

统计信息	作用域	说明
高速缓存替换:总页数/秒	服务器	显示为了给所需要的另一页释放空间而从高速缓存中清除数据库页的速率。
高速缓存大小:当前值	服务器	显示数据库服务器高速缓存的当前大小，以千字节为单位。
高速缓存大小:最大值	服务器	显示数据库服务器高速缓存所允许的最大值，以千字节为单位。
高速缓存大小:最小值	服务器	显示数据库服务器高速缓存所允许的最小值，以千字节为单位。
高速缓存大小:峰值	服务器	显示数据库服务器高速缓存的峰值大小，以千字节为单位。

检查点和恢复统计

以下统计信息将数据库处于空闲状态时所执行的检查点和恢复操作隔离出来。

统计信息	作用域	说明
检查点刷新/秒	数据库	显示检查点过程中写出相邻页范围的速率。
检查点紧急情况	数据库	显示检查点紧急情况，以百分比表示。
检查点/秒	数据库	显示执行检查点的速率。
检查点日志:位图大小	数据库	显示检查点日志位图的大小。
检查点日志:提交到磁盘/秒	数据库	显示执行检查点日志 commit_to_disk 操作的速率。

统计信息	作用域	说明
检查点日志:日志大小	数据库	显示检查点日志的大小（以页为单位）。
检查点日志:保存的页图像/秒	数据库	显示进行修改前在检查点日志中保存页的速率。
检查点日志:正在使用的页	数据库	显示检查点日志中当前正在使用的页数。
检查点日志:重新定位页/秒	数据库	显示检查点日志中对页进行重新定位的速率。
检查点日志:保存前像/秒	数据库	显示向检查点日志添加新的数据库页前像的速率。
检查点日志:写入页/秒	数据库	显示向检查点日志中写入页的速率。
检查点日志:写入/秒	数据库	显示检查点日志中执行磁盘写入的速率。一次写入可以包含多页。
检查点日志:写入位图/秒	数据库	显示在位图页的检查点日志中执行磁盘写入的速率。
空闲活动/秒	数据库	显示数据库服务器的空闲线程变为活动状态以执行空闲写入和空闲检查点等的速率。
空闲检查点时间	数据库	显示执行空闲检查点所花费的总时间，以秒为单位。
空闲检查点/秒	数据库	显示数据库服务器的空闲线程完成检查点的速率。只要空闲线程在高速缓存中写出最后的脏页，会出现空闲检查点。
空闲写入/秒	数据库	显示数据库服务器的空闲线程发出磁盘写入的速率。
恢复 I/O 估计	数据库	显示恢复数据库所需的 I/O 操作的估计数。
恢复紧急情况	数据库	显示恢复紧急情况，以百分比表示。

通信统计

以下统计信息描述客户端/服务器的通信活动。

统计信息	作用域	说明
通信:收到的字节/秒	连接和服务器	显示接收网络数据的速率（以字节为单位）。
通信:未压缩时收到的字节/秒	连接和服务器	显示在禁用压缩的情况下，接收字节的速率。
通信:发送的字节/秒	连接和服务器	显示通过网络传输字节的速率。
通信:未压缩时发送的字节/秒	连接和服务器	显示在禁用压缩的情况下，发送字节的速率。
通信:可用缓冲区	服务器	显示可用网络缓冲区的数量。
通信:收到的多个数据包/秒	服务器	显示接收多个数据包的速率。
通信:发送的多个数据包/秒	服务器	显示发送多个数据包的速率。
通信:收到的数据包/秒	连接和服务器	显示接收网络数据包的速率。
通信:未压缩时收到的数据包/秒	连接和服务器	显示在禁用压缩的情况下，接收网络数据包的速率。
通信:发送的数据包/秒	连接和服务器	显示发送网络数据包的速率。
通信:未压缩时发送的数据包/秒	连接和服务器	显示在禁用压缩的情况下，发送网络数据包的速率。

统计信息	作用域	说明
通信:远程发送等待/秒	服务器	显示通信链接因没有可用于发送信息的缓冲区而必须等待的速率。仅对 TCP/IP 收集此统计信息。
通信:收到的请求	连接和服务器	显示客户端/服务器通信请求数或往返次数。它与 [通信:收到的数据包] 的不同之处在于：多包请求将计为一个请求，并且不包括活动包。
通信:发送失败/秒	服务器	显示基础协议未能发送数据包的速率。
通信:总缓冲区	服务器	显示网络缓冲区的总数。
通信:唯一客户端地址	服务器	显示连接到数据库服务器的唯一客户端网络地址的数量。这通常是连接的客户机数，并可能小于连接总数。

磁盘 I/O 统计

以下统计将磁盘读取和磁盘写入进行组合，提供了有关专用于磁盘 I/O 活动数量的全面信息。

统计信息	作用域	说明
磁盘：活动 I/O	数据库	显示数据库服务器所发出的尚未完成的文件 I/O 的当前数量。
磁盘：最大活动 I/O	数据库	显示 "磁盘:活动 I/O" 已达到的最大值。

磁盘读取统计

以下统计描述了专门从磁盘读取信息的活动的数量和类型。

统计信息	作用域	说明
磁盘读取:总页数/秒	连接和数据库	显示从文件中读取页面的速率。
磁盘读取:活动	数据库	显示数据库服务器所发出的尚未完成的文件读取的当前数量。
磁盘读取:索引内部/秒	连接和数据库	显示从磁盘读取索引内部节点页面的速率。

统计信息	作用域	说明
磁盘读取:索引叶/秒	连接和数据库	显示从磁盘读取索引叶页的速率。
磁盘读取:表/秒	连接和数据库	显示从磁盘读取表页面的速率。
磁盘读取:最大活动	数据库	显示 "磁盘读取:活动" 已达到的最大值。
磁盘读取:工作表	连接和数据库	显示从磁盘读取工作表页面的速率。

磁盘写入统计

以下统计描述了专门向磁盘写入信息的活动的数量和类型。

统计信息	作用域	说明
磁盘写入:活动	数据库	显示数据库服务器所发出的尚未完成的文件写入的当前数量。
磁盘写入:最大活动	数据库	显示 "磁盘写入:活动" 已达到的最大值。
磁盘写入:提交文件/秒	数据库	显示数据库服务器强制磁盘高速缓存刷新的速率。Windows 平台使用非缓冲（直接）I/O，因此磁盘高速缓存不需要刷新。
磁盘写入:数据库扩展/秒	数据库	显示数据库文件扩展的速率，以页数/秒表示。
磁盘写入:临时扩展/秒	数据库	显示临时文件扩展的速率，以页数/秒表示。
磁盘写入:页/秒	连接和数据库	显示已修改页面写入到磁盘的速率。
磁盘写入:事务日志/秒	连接和数据库	显示页面写入到事务日志的速率。
事务日志组提交/秒	连接和数据库	显示当请求提交事务日志但该日志已被写入（因而该提交以“释放”状态完成）时的速率。

索引统计

以下统计描述了索引的使用。

统计信息	作用域	说明
索引:添加/秒	连接和数据库	显示将项目添加到索引的速率。
索引:查找/秒	连接和数据库	显示在索引中查找条目的速率。
索引:完全比较/秒	连接和数据库	显示必须执行索引中散列值以外的比较的速率。

内存诊断统计

以下统计描述了数据库服务器如何使用内存。

统计	范围	说明
高速缓存:多页分配	服务器	显示多页分配数。
高速缓存:混乱	服务器	显示高速缓存管理器未能找到要分配页的次数。
高速缓存:已访问搜寻	服务器	显示清除回收以分配页时访问的页数。
高速缓存:搜寻	服务器	显示高速缓存管理器清除回收以分配页的次数。
高速缓存页:已分配结构	服务器	显示已为数据库服务器数据结构分配的高速缓存页数。
高速缓存页:文件	服务器	显示用于保存数据库文件中数据的高速缓存页数。
高速缓存页:脏文件	服务器	显示处于脏状态的高速缓存页数（需要写操作）。
高速缓存页:空闲	服务器	显示未使用的高速缓存页数。
高速缓存页:固定	服务器	显示当前无法重新使用的页数。

统计	范围	说明
高速缓存替换:总页数/秒	服务器	显示为了给所需要的另一页释放空间而从高速缓存中清除数据库页的速率。
堆:雕刻	连接和服务器	显示供短期使用（如查询优化）的堆数。
堆:查询处理	连接和服务器	显示用于查询处理（散列和排序操作）的堆数。
堆:可重定位	连接和服务器	显示可重定位堆的数量。
堆:被锁可重定位	连接和服务器	显示高速缓存中当前锁定的可重定位堆的数量。
映射物理内存/秒	服务器	显示使用 Address Windowing Extensions 将数据库页地址空间窗口映射到高速缓存中物理内存的速率。
记忆页:雕刻	连接和服务器	显示供短期使用（如查询优化）的堆页的数量。
记忆页:已固定游标	服务器	显示用于保持游标堆驻留在内存中的页数。
记忆页:查询处理	连接和服务器	显示用于查询处理（散列和排序操作）的高速缓存页数。
查询内存:当前活动	连接和服务器	显示主动使用查询内存的当前请求数。
查询内存:预计活动	服务器	显示在数据库服务器处于稳定状态时对主动使用查询内存的请求数平均值的估计。
查询内存:额外可用	服务器	显示可用于在基础内存密集型授权之外进行授权的内存量。
查询内存:授权失败数	连接和服务器	显示任何请求等待但未能获得查询内存的总次数。

统计	范围	说明
查询内存:授权请求数	连接和服务器	显示任何请求尝试获取查询内存的总次数。
查询内存:授权等待数	连接和服务器	显示任何请求等待内存的总次数。
查询内存:授权页	连接和服务器	显示当前授予请求的页数。
查询内存:正在等待的请求	连接和服务器	显示等待查询内存的当前请求数。

记忆页统计

以下统计描述了数据库服务器所使用的内存的数量和用途。

统计信息	作用域	说明
记忆页:锁定表	数据库	显示用于存储锁定信息的页数。
记忆页:锁定堆	服务器	显示高速缓存中锁定的堆页数。
记忆页:主堆	服务器	显示用于全局数据库服务器数据结构的页数。
记忆页:映射页	数据库	显示用于访问锁表、频率表和表布局的映射页数。
记忆页:过程定义	数据库	显示用于过程的可重定位堆页的数量。
记忆页:可重定位	数据库	显示用于可重定位堆（游标、语句、过程、触发器、视图等）的页数。
记忆页:重新定位/秒	数据库	显示从临时文件读取可重定位堆页的速率。

统计信息	作用域	说明
记忆页:回退日志	连接和数据库	显示回退日志中的页数。
记忆页:触发器定义	数据库	显示用于触发器的可重定位堆页的数量。
记忆页:视图定义	数据库	显示用于视图的可重定位堆页的数量。

请求统计

以下统计描述了专用于响应来自客户端应用程序请求的数据库服务器活动。

统计信息	作用域	说明
游标	连接	显示数据库服务器当前所维护的已声明游标的数量。
打开的游标	连接	显示数据库服务器当前所维护的打开游标的数量。
锁定计数	连接和数据库	显示锁定的个数。
请求/秒	服务器	显示进入数据库服务器以允许其处理新请求或继续处理现有请求的速率。
请求:活动	服务器	显示当前正在处理请求的数据库服务器线程的数量。
任务:交换	服务器	显示当前正用于并行执行查询的数据库服务器线程的数量。
请求:未调度	服务器	显示当前排队等待可用数据库服务器线程的请求的数量。
快照计数	连接和数据库	显示活动快照数。
语句高速缓存命中	连接和服务器	显示数据库服务器重新使用由客户端高速缓存的语句准备的速率。
语句高速缓存未命中	连接和服务器	显示需要数据库服务器再次准备由客户端高速缓存的语句准备的速率。
语句准备	连接和数据库	显示数据库服务器处理语句准备的速率。

统计信息	作用域	说明
语句	连接	显示数据库服务器当前所维护的经过准备的语句的数量。
事务提交	连接	显示处理提交请求的速率。
事务回退	连接	显示处理回退请求的速率。

杂项统计

统计信息	作用域	说明
可用 IO	服务器	显示当前可用的 I/O 控制块数。
连接计数	数据库	显示此数据库的连接数量。
主堆字节	服务器	显示用于全局数据库服务器数据结构的字节数。
查询:计划高速缓存页	连接和数据库	显示用于高速缓存执行计划的页数。
查询:内存不足策略	连接和数据库	显示由于内存条件不够而使服务器在执行期间改变其执行计划的次数。
查询:实现的行/秒	连接和数据库	显示在查询处理过程中，行写入工作表的速率。
请求:GET DATA/秒	连接和数据库	显示连接发出 GET DATA 请求的速率。
临时表页数	连接和数据库	显示用于临时表的临时文件中的页数。
版本存储页数	数据库	显示启用快照隔离时正用于行版本存储的临时文件的页数。

性能提高提示

获取适当的硬件

在 PC 上运行时，应满足下面的最低 CPU、内存和磁盘要求：

- 最低 4 MB 内存。如果要使用管理工具（如 Sybase Central 和 Interactive SQL），则至少需要 48 MB 的 RAM。
- 有足够的空间用于存放数据库和日志文件。

如果您的服务器只满足最低硬件要求，则性能可能会受到影响；此时最好升级您的硬件。一般而言，您需要评估硬件配置以查看它是否适用于数据库服务器所承担的工作负载种类。

您可以在启动数据库服务器时指定 -fc 选项，以在数据库服务器出现文件系统已满时执行回调函数。

有关详细信息，请参见“-fc 服务器选项”一节 《SQL Anywhere 服务器 - 数据库管理》。

始终使用事务日志

使用事务日志可提供数据保护，同时能显著提高 SQL Anywhere 的性能。

在不使用事务日志的情况下运行时，SQL Anywhere 会在每个事务结束时执行一次检查点操作，这会消耗相当多的资源。

但如果使用事务日志运行，则 SQL Anywhere 只是在更改发生时才写入详细介绍这些更改的注释。它可以选择在最有效的时间一次写入所有新数据库页。**检查点**将确保信息进入数据库文件并确保信息一致且最新。

如果将事务日志存储在包含主数据库文件的设备之外的其它物理设备上，则可以进一步提高性能。额外的驱动器头通常不必进行查找便可到达事务日志的末尾。

检查并发问题

在数据库服务器处理事务时，可以锁定表中的一行或多行。锁可以通过防止其它事务并发访问来保持数据库中存储的信息的可靠性。锁还可以通过标识正在进行更新的信息来提高查询结果的准确性。

数据库服务器会自动放置这些锁，而无需显式指定。它会保存事务获取的所有锁，直到完成该事务。可以访问该行的事务被称为持有锁。其它事务也许可以对锁定行进行有限的访问，也许根本不能访问，这取决于锁的类型。

如果多个用户频繁地同时访问一行或多行，则可能会降低性能。如果您怀疑有锁定问题，可以考虑使用 sa_locks 过程来获取有关数据库中锁的信息。请参见“sa_locks 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》。

如果发现了锁定问题，可使用 AppInfo 连接属性来找到有关所涉及的连接过程的信息。请参见“连接属性”一节 《SQL Anywhere 服务器 - 数据库管理》。

选择优化程序的目标

optimization_goal 选项用于控制 SQL Anywhere 是为减少响应时间 (First-row) 还是为降低总资源消耗 (All-rows) 而优化 SQL 语句。简而言之，您可以选择优化查询处理是为了快速返回第一行还是尽可能降低返回完整结果集的开销。

如果将选项设置为 First-row，SQL Anywhere 会选择用于减少查询结果的首行提取时间访问计划，而且很可能以总检索时间为代价。特别是，优化程序通常会尽量避免需要实现结果以便减少返回首行的时间的访问计划。例如，使用此设置，优化程序会倾向于利用索引来满足查询的 ORDER BY 子句的访问计划，而不是需要显式排序操作的计划。

优化程序针对某一特定语句所使用的优化目标可通过以下规则确定：

- 如果主查询块的 FROM 子句中含有表，且表提示已设置为 FASTFIRSTROW，则使用 First-row 优化目标对语句进行优化。
- 如果语句具有 OPTION 子句，且该子句包含对 optimization_goal 选项的设置，则使用此设置来优化语句。
- 否则，优化程序将使用 optimization_goal 选项的当前设置。

请注意，即使优化目标是 First-row，但优化程序可能仍无法找到能够快速返回第一行的计划。例如，对于因具有 DISTINCT、GROUP BY 或 ORDER BY 子句而需要实例化但因没有相关索引从而无法提供必要顺序的语句，将使用 All-rows 目标进行优化。

如果此选项设置为 All-rows（缺省值），则对 SQL Anywhere 查询进行优化后将选择预计总检索时间最少的访问计划。将 optimization_goal 设置为 All-rows 可能适用于要处理整个结果集的应用程序，例如 PowerBuilder DataWindow 应用程序。

另请参见

- “optimization_goal 选项 [数据库]”一节 《SQL Anywhere 服务器 - 数据库管理》
- “FROM 子句”一节 《SQL Anywhere 服务器 - SQL 参考》

另请参见以下 SQL 语句的 OPTION 子句：

- “DELETE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “INSERT 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “SELECT 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “MERGE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “UPDATE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “UNION 子句”一节 《SQL Anywhere 服务器 - SQL 参考》

收集小表上的统计信息

SQL Anywhere 利用统计信息来确定每个语句最有效的执行策略。SQL Anywhere 自动收集和更新这些统计信息，并将它们永久地存储在数据库中。处理一个语句时所收集的统计信息对于搜索后续语句的有效执行方式很有用。

缺省情况下，SQL Anywhere 为含有五行或更多行的所有表创建统计信息。如果需要为含有少于五行的表创建统计信息，则可以使用 CREATE STATISTICS 语句做到这一点。此语句为所有表创建统计信息，与表的行数无关。一旦创建了统计信息，SQL Anywhere 就会自动对其进行维护。请参见“CREATE STATISTICS 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

声明约束

当不同表中的列值之间存在隐含关系时，表之间存在未声明的主键-外键关系。无疑，不声明该关系可以节省维护索引的时间，但是，声明该关系可以在进行连接时提高查询的性能，因为该成本模型可以更好地进行预计。请参见“使用表和列约束”一节第 85 页。

增加高速缓存大小

SQL Anywhere 会将最近使用的页存储在高速缓存中。如果某一请求需要多次访问该页或者另一个连接需要使用同一页，它们会发现该页在内存中已经存在，这样就可以避免从磁盘中读取信息。这对于加密数据库尤其重要，因为加密数据库比未加密的数据库需要更大的高速缓存。

如果您的高速缓存太小，SQL Anywhere 将无法在内存中将页保存足够长的时间，从而无法实现上述优点。

在 Unix 和 Windows 中，数据库服务器会根据需要动态地更改高速缓存的大小。但是，高速缓存仍然受到实际可用的内存量以及其它应用程序占用的内存量的限制。

提示
由于从内存中检索信息比从磁盘中读取信息快许多倍，因此增加高速缓存大小通常会显著地提高性能。您可能会发现增加更多的内存是值得的，因为这样可以获取更大的高速缓存。

请参见“使用高速缓存提高性能”一节第 221 页。

最大程度地减少级联参照动作

就性能而言，级联参照操作的开销非常高，因为它们会导致更新每个事务的多个表。例如，如果从 Employees 到 Departments 都使用 ON UPDATE CASCADE 定义了外键，那么，在更新部门 ID 时，Employees 表就会自动更新。虽然级联参照动作很方便，但有时在应用程序逻辑中实现它们可能会更有效。请参见“确保数据完整性”第 73 页。

监控查询性能

SQL Anywhere 包括多种用于测试查询性能的工具。这些工具存储在 *samples-dir\SQLAnywhere* 下的子目录中，如下所述。有关每种工具的完整文档，可以在 *readme.txt* 文件中找到，该文件与工具位于相同的文件夹中。有关 *samples-dir* 位置的详细信息，请参见“示例目录”一节 《SQL Anywhere 服务器 - 数据库管理》。

有关用于测量查询执行时间的系统过程的信息，请参见“sa_get_request_profile 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》和“sa_get_request_times 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》。

fetchtst

功能 确定检索某一结果集所需的时间。

位置 *samples-dir\SQLAnywhere\PerformanceFetch*

odbcfet

功能 确定检索某一结果集所需的时间。此工具与 fetchtst 相似，但是功能较少。

位置 *samples-dir\SQLAnywhere\PerformanceFetch*

instest

功能 确定将行插入表中所需的时间。

位置 *samples-dir\SQLAnywhere\PerformanceInsert*

trantest

功能 测量给定数据库服务器配置（已给定数据库设计和一组事务）可以处理的负载。

位置 *samples-dir\SQLAnywhere\PerformanceTransaction*

规范化表结构

一个或多个数据库表可能包含相同信息（例如，重复出现在多个表中的某一列）的多个副本，此时该表可能需要规范化。

规范化将减少关系数据库中的重复项。例如，假设贵公司的员工在若干个不同的办事处工作。要将数据库规范化，应该将有关办事处的信息（如办事处的地址和主要电话号码）放入单独的表中，而不是为每个员工复制所有这些信息。

如果重复信息量不大，则最好复制该信息，并使用触发器或其它约束维护信息的完整性。

有关规范化数据的详细信息，请参见“在 SQL Anywhere 中创建数据库”第 3 页。

复查表中列的顺序

对某一行中各列的访问按照其创建顺序依次进行。例如，要访问行末尾的列，SQL Anywhere 就必须跳过行中先出现的列。主键列始终存储在行的开头。因此，创建表时让较小的和/或经常被访问的列放在表中很少被访问的列的前面，是很重要的。

另请参见

- “在 SQL Anywhere 中创建数据库”第 3 页
- “管理主键”一节第 22 页

将不同的文件放置在不同的设备上

磁盘驱动器的运行速度远远低于现代的处理器或内存。通常，等待磁盘读写页就是导致数据库服务器速度慢的原因。

将不同的物理数据库文件放置在不同的物理设备，这样做可能会提高数据库的性能。例如，当一个磁盘驱动器忙于与高速缓存交换数据库页时，另一个设备可能会正在写入日志文件。

请注意，若要实现这些优点，设备必须是独立的。分区成多个较小的逻辑驱动器的单个磁盘不大可能提供这些优点。

SQL Anywhere 使用四种类型的文件：

1. 数据库 (*.db*)
2. 事务日志 (*.log*)
3. 事务日志镜像 (*.mlg*)
4. 临时文件 (*.tmp*)

数据库文件包含数据库的全部内容。一个文件可以包含一个数据库，或者，您可以添加最多 12 个 dbspace，它们是存有相同数据库的某些部分的附加文件。可以为数据库文件和 dbspaces 选择一个位置。

事务日志文件是在出现故障时恢复数据库信息所必需的。为了提供额外的保护，您可以在第三种文件（称作**事务日志镜像文件**）中维护事务日志的一个副本。SQL Anywhere 会在相同的时间将相同的信息写入这两个文件中的每一个。

提示

将事务日志镜像文件（如果您使用了事务日志镜像文件）放置在物理上独立的驱动器上将有助于保护这些文件以防发生磁盘故障，而 SQL Anywhere 的运行速度也会更快，因为它可以高效地对日志文件和日志镜像文件执行写入操作。要指定事务日志文件和事务日志镜像文件的位置，请使用事务日志实用程序 (dblog) 或 Sybase Central 中的 [**更改日志文件设置向导**]。请参见“事务日志实用程序 (dblog)”一节 《SQL Anywhere 服务器 - 数据库管理》和“更改事务日志的位置”一节 《SQL Anywhere 服务器 - 数据库管理》。

对于排序和形成联合等操作，当 SQL Anywhere 所需的空间大于高速缓存中的可用空间时，将使用**临时文件**。当数据库服务器需要该空间时，通常会集中使用该空间。数据库的总体性能越来越多地依赖于包含临时文件的设备的速度。

提示
如果临时文件位于快速设备上，并且该设备在物理空间上独立于保存数据库文件的设备，则 SQL Anywhere 通常会运行得更快。这是因为许多必须使用临时文件的操作也需要从数据库中检索大量的信息。通过将信息放置在两个单独的磁盘上，这些操作就可以同时进行。

请谨慎选择临时文件的位置。临时文件的位置可在使用 -dt 服务器选项启动数据库服务器时指定。如果在启动数据库服务器时未指定临时文件的位置，则 SQL Anywhere 会依次检查以下环境变量：

1. SATMP
2. TMP
3. TMPDIR
4. TEMP

如果未定义任何环境变量，则在 Windows 中，SQL Anywhere 会将其临时文件放在当前目录中，而在 Unix 中，则会放在 */tmp* 目录中。

如果您的计算机配备有足够数量的快速设备，则可以将这些文件中的每一个分别放置在单独的设备上，从而获得更高的性能。您甚至可以将数据库分为多个位于单独设备上的 dbspace。在这种情况下，请将单独 dbspace 中的表分组，使常见的连接操作从不同的 dbspace 中读取信息。

当您在系统 dbspace 之外的位置创建所有表或索引时，系统 dbspace 仅用于存储检查点日志和系统表。在您由于性能原因希望将检查点日志放在与其余数据库对象位置不同的单独磁盘上时，这很有用。要在另一个 dbspace 中创建基表，更改所有 CREATE TABLE 语句以使用 IN DBSPACE 子句，从而指定另一个替代 dbspace，或者在创建任何表之前更改 default_dbspace 选项的设置。临时表只能在 TEMPORARY dbspace 中创建。请参见“default_dbspace 选项 [数据库]”一节 《SQL Anywhere 服务器 - 数据库管理》和“CREATE TABLE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

有关基表和临时表的缺省 dbspace 的详细信息，请参见“使用附加 dbspace”一节 《SQL Anywhere 服务器 - 数据库管理》。

一种类似的策略是将临时文件和数据库文件放在 RAID 设备或带区集上。虽然这些设备充当逻辑驱动器，但它们会在多个物理驱动器上分配文件并使用多个驱动器头来访问信息，从而显著地提高性能。

您可以在启动数据库服务器时指定 -fc 选项，以在数据库服务器出现文件系统已满时执行回调函数。请参见“-fc 服务器选项”一节 《SQL Anywhere 服务器 - 数据库管理》。

另请参见

- “在查询处理中使用工作表（使用 All-rows 优化目标）”一节第 227 页
- “备份和数据恢复” 《SQL Anywhere 服务器 - 数据库管理》
- “SATMP 环境变量”一节 《SQL Anywhere 服务器 - 数据库管理》

重建数据库

重建数据库是卸载并重装整个数据库的过程，也称为升级数据库文件格式。

重建会删除所有信息（包括数据和模式），并以统一方式将其全部放回到原位置。就像整理磁盘驱动器碎片一样，性能会提高，空间也会被填充。这也会使您有机会更改某些设置。请参见“重建数据库”一节第 709 页。

减少碎片

更改数据库时，自然会出现碎片。如果文件、表或索引出现了过多的碎片，则可能会破坏性能。随着数据库不断增大，减少碎片会变得更加重要。SQL Anywhere 包含一些存储过程，它们会生成有关文件、表和索引碎片的信息。

如果您发现性能显著降低，请考虑：

- 重建数据库以减少表和/或索引碎片，尤其是已经对多个表执行了大量删除/更新/插入操作的情况下。
- 将数据库单独放置在磁盘分区上以减少文件碎片
- 定期运行可用的 Windows 实用程序之一以减少文件碎片
- 对表进行重组以减少数据库碎片
- 使用 REORGANIZE TABLE 语句整理表中行的碎片，或压缩可能因执行 DELETE 而变得分散的索引。对表进行重组可以减少用于存储表及其索引的总页数，而且还可以减少索引树中的级别数。

减少文件碎片

包含碎片的数据库文件会影响数据库服务器的性能。随着数据库不断增大，清理磁盘碎片会变得越来越重要。

当您在 Windows 上启动数据库时，数据库服务器会确定每个 dbspace 中的文件碎片数。如果碎片数大于 1，数据库服务器就会在数据库服务器消息窗口中显示以下信息：**数据库文件 "mydatabase.db" 包含 nnn 个碎片**。

也可以使用 DBFileFragments 数据库属性来获取数据库文件碎片的数量。请参见“数据库属性”一节 《SQL Anywhere 服务器 - 数据库管理》。

要消除文件碎片问题，请将数据库单独放置在一个磁盘分区上，然后定期运行可用的 Windows 磁盘碎片整理实用程序之一。

减少表碎片

当没有连续存储行时，或者当行在多个页之间被拆分时，会出现表碎片。这些行需要附加的页面访问，因此会降低数据库服务器的性能。

碎片对性能的影响在不同情况下各不相同。表的碎片可能非常多，但是如果表装入内存，并且访问表的方式允许对页进行高速缓存，则碎片对性能的影响可能会非常小。另一方面，如果频繁地访问拆分行并且高速缓存不能减少额外 I/O 的开销，则可能需要更多的 I/O 才能访问碎片表，从而导致性能显著降低。

尽管重组表和重建数据库可减少碎片，但此类操作过于频繁或不够频繁也会影响性能。使用下面一节中介绍的工具和方法进行试验，以确定表的可接受碎片程度。

如果减少了碎片但性能仍然很差，则可能存在另外的问题（如不准确的统计信息）。

确定表碎片的程度

可以使用 sa_table_fragmentation 系统过程来获取有关数据库表的碎片程度的信息。要确定是否整理碎片以提高性能，仅运行一次此系统过程是没有用的。而应重建数据库并运行此过程来建立基线结果。然后，在较长的时间内继续定期运行该过程，查找其输出变化与性能测量变化之间的关系。这样，您就可以确定表以什么速率进行碎片化才能达到影响性能的程度，从而确定对表进行碎片整理的最佳频率。

要运行此过程，必须具有 DBA 权限。使用以下语句可调用 sa_table_fragmentation 系统过程：

```
CALL sa_table_fragmentation( [ 'table-name' [, 'owner-name' ] ] );
```

减少碎片的方法

以下方法有助于控制表碎片：

- **使用 PCTFREE**　SQL Anywhere 在每一页上保留了额外的空间，以允许行略微地增大。当对行的更新使行大小超过为它分配的初始空间时，该行将被拆分，而初始行位置将包含一个指针，它指向存储整个行的另一页。例如，如果用 UPDATE 语句填充空行或将新列插入表，可能会导致严重的行拆分。当更多的行存储在单独的页上时，需要更多的时间来访问附加页。

  通过指定应该为将来的更新而保留的空间在表页中所占的百分比，可以减少表中的碎片数。对 PCTFREE 的指定可以用 CREATE TABLE、ALTER TABLE、DECLARE LOCAL TEMPORARY TABLE 或 LOAD TABLE 来设置。

- **重组表**　可以使用 REORGANIZE TABLE 语句来整理特定表的碎片。重组表的操作不会中断数据库访问。

- **重建数据库**　*倘若重建分两步过程执行*（即，将数据卸载并存储到磁盘，然后重装），则重建数据库将整理包括系统表在内的所有表的碎片。以这种方式进行重建还有一个优点，会重排表行以便它们按聚簇索引和主键指定的顺序出现。一步重建（例如，使用 -ar、-an 或 -ac 选项）不会减少表碎片。

另请参见

- “sa_table_fragmentation 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》
- “REORGANIZE TABLE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “卸载实用程序 (dbunload)”一节 《SQL Anywhere 服务器 - 数据库管理》
- “重建实用程序 (rebuild)”一节 《SQL Anywhere 服务器 - 数据库管理》

减少索引碎片和分布偏差

索引用于加快对特定列的搜索速度，但如果对索引表执行了太多的 DELETE 操作，索引就会出现碎片（不紧凑）和分布偏差（不均衡）。

索引密度反映了索引页的平均丰满度。索引分布偏差反映了与平均密度的典型差值。进行选择性估计时，分布偏差量对于优化程序十分重要。

要确定数据库中是否存在其碎片或分布偏差水平无法接受的索引，可使用 [**应用程序分析向导**]。请参见“应用程序分析向导”一节第 161 页。

也可以使用 sa_index_fragmentation 系统过程来查看索引碎片和分布偏差的程度。例如，以下语句调用 sa_index_density 系统过程来检查 Customers 表上的索引。

```
CALL sa_index_density( 'Customers' );
```

TableName	TableId	IndexName	IndexID	IndexType	LeafPages	Density	Skew
Customers	686	CustKey	0	PKEY	1	0.645992	1.002772
Customers	686	IX_cust_name	1	NUI	1	0.789795	1.432239

SQL Anywhere 自动在主键上创建索引。请注意，在 sa_index_density 系统过程的结果中，这些索引的 IndexID 为 0。

当叶页数较低时，无需考虑密度和分布偏差值。密度和分布偏差值仅在叶页数较高时才变得重要。当叶页数较高时，较低的密度值表示存在碎片，而较高的分布偏差值则表示索引分布不均衡。二者都有可能是导致性能低下的因素。通过执行 REORGANIZE TABLE 语句即可解决这两个问题。请参见“REORGANIZE TABLE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

另请参见

- “sa_index_density 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》
- “使用索引”一节第 66 页

减小主键宽度

宽主键由两个或更多个列组成。主键中包含的列越多，对数据库服务器的需求越多。减小主键中的列数可以提高性能。

另请参见

- “管理主键”一节第 22 页
- “使用键提高查询性能”一节第 221 页

减小表宽

在某些表中，组合列的大小（或单一行的大小）超过了数据库页的大小，因而必须在两个或更多个数据库页之间对其进行拆分，这种表被称作宽表。行占用的页数越多，数据库服务器读取每一行所

需的时间越长。如果在使用宽表时发现性能很低，则应考虑进一步规范化表以减少列数。如果这无法实现，较大的数据库页大小可能会有所帮助，尤其是在多数表都很宽的情况下。请参见“在 SQL Anywhere 中创建数据库”第 3 页。

减少客户端和服务器之间的请求

如果发现您的网络等待时间较长，或您的应用程序发送许多游标打开和关闭请求，您可以使用 LazyClose 和 PrefetchOnOpen 网络连接参数来减少客户端和服务器之间的请求数，从而提高性能。请参见“LazyClose 连接参数 [LCLOSE]”一节 《SQL Anywhere 服务器 - 数据库管理》和“PreFetchOnOpen 连接参数”一节 《SQL Anywhere 服务器 - 数据库管理》。

减少开销高的用户定义函数

在必须多次执行高开销的用户定义函数的查询中，减少这些函数可以提高性能。请参见“用户定义的函数简介”一节第 786 页。

替换高开销的触发器

评估触发器的用途以查看一些触发器是否可以由数据库服务器中提供的功能替换。例如，使用最新的更新时间和用户信息更新列的触发器可由数据库服务器中相应的特殊值替换。此外，使用现有触发器上的缺省设置也可以提高性能。请参见“触发器简介”一节第 789 页。

查询结果的策略性排序

减少不必要的数据排序；除非您需要数据按可预知的顺序返回，否则不要在 SELECT 语句中指定 ORDER BY 子句。排序需要额外的时间和资源来处理查询。

有关排序的详细信息，请参见“ORDER BY 子句：查询结果排序”一节第 351 页或“GROUP BY 子句：将查询结果划分为组”一节第 344 页。

指定正确的游标类型

指定正确的游标类型可以提高性能。例如，如果游标是只读的，则将它声明为 [只读] 可以加速优化和执行，因为要生成的资料较少（没有检查约束，等等）。如果游标是可更新的，则可以跳过某些查询重写。此外，如果查询是可更新的，则根据优化程序选择的执行计划，数据库服务器必须使用键集决定的方法。请记住，键集游标开销更大。请参见“选择游标类型”一节 《SQL Anywhere 服务器 - 编程》。

谨慎提供显式选择性估计值

有时，统计信息可能会变得不准确。如果在添加、更新或删除了大量数据后只执行了很少的查询，则最有可能发生这种情况。统计信息不准确或不可用会降低性能。如果 SQL Anywhere 更新统计信息花费的时间太长，请尝试执行 CREATE STATISTICS 或 DROP STATISTICS 以刷新它们。

SQL Anywhere 还会在执行 LOAD TABLE 语句时、查询执行期间和执行更新 DML 语句时更新某些统计信息。

然而，在极少数情况下，这些措施可能无效。如果您知道某条件的成功率与优化程序估计的不一样，可以在搜索条件中显式地提供用户的估计。

虽然用户定义的预计值有时可以提高性能，但是要避免在不断使用的语句中提供显式的用户定义预计值。如果数据改变，则显式估计可能会变得不准确并强制优化程序选择低效率的计划。

对于由于软件选择的访问计划不佳而引发的性能问题，如果您使用了不准确的选择性估计值作为变通解决办法，则可以将 user_estimates 设置为 Off 以忽略这些值。请参见“显式选择性估计”一节《SQL Anywhere 服务器 - SQL 参考》。

关闭自动提交模式

如果应用程序以**自动提交模式**运行，SQL Anywhere 就会将每个语句当作单独的事务。实际上，这等效于在每个命令的末尾附加 COMMIT 语句。

不要以自动提交模式运行，而应该将命令分组，使每个命令组执行一个逻辑任务。如果您禁用了自动提交，则必须在每个逻辑命令组后执行一个显式提交命令。另外，请注意如果逻辑事务太大，则会发生阻塞和死锁。

如果没有使用事务日志文件，则使用自动提交模式的开销会很大。每个语句会强制执行一个检查点操作—该操作可能需要将多页信息写入磁盘。

每个应用程序接口各自有不同的设置自动提交行为的方式。对于 Open Client、ODBC 和 JDBC 接口，自动提交是缺省行为。

有关自动提交的详细信息，请参见“设置自动提交或手工提交模式”一节 《SQL Anywhere 服务器 - 编程》。

使用适当的页大小

选择的页大小会影响数据库的性能。较大的页和较小的页都存在各自的优缺点。

较小的页保存的信息较少，对空间的使用效率较低，特别是当您插入的行略大于页面大小的一半时。但较小的页大小也使 SQL Anywhere 可以使用较少的资源运行，因为它在相同大小的高速缓存中可以存储更多的页。如果您的数据库在内存有限的小型计算机上运行，较小的页将特别有用。如果数据库主要用于从随机位置检索小块信息，较小的页也很有帮助。

较大的页大小有助于 SQL Anywhere 更有效地读取数据库。较大的页大小往往有益于大型数据库和执行顺次表扫描的查询。通常，磁盘的物理设计会使磁盘检索较少的大块比检索较多的小块更有效率。较大的页大小还有其它优点，其中包括提高索引的涵盖范围，进而减少索引级别数并使表可以包括更多的列。

请记住，较大的页大小对内存的需求也更多。此外，除非您能确保总是有较大的数据库服务器高速缓存，否则，对于大多数应用程序，都不建议使用极其大的页大小（16 KB 或 32 KB）。在使用 16 KB 或 32 KB 页大小前，先要调查内存和磁盘空间的增加对性能特性的影响。

数据库服务器的内存使用量与装载的数据库数和数据库的页大小成比例。强烈建议您在选择页大小时进行性能测试（和一般测试）。然后，选择可提供满意结果的最小的页面大小 (>= 4 KB)。如果要在同一服务器上启动大量数据库，那么选择正确（且合理）的页面大小很重要。

您不能更改现有数据库的页大小。而必须新建数据库并使用 dbinit 的 -p 选项来指定页大小。例如，以下命令将创建一个页大小为 4 KB 的数据库。

```
dbinit -p 4096 new.db
```

还可使用包含 PAGE SIZE 子句的 CREATE DATABASE 语句来创建具有新的页大小的数据库。请参见“CREATE DATABASE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

有关较大的页大小的详细信息，请参见“设置最大页面大小”一节 《SQL Anywhere 服务器 - 数据库管理》。

分散读取

如果您使用的是 Windows，则使用最小 4 KB 的页大小使得数据库服务器可将磁盘上数据库页较大的连续部分直接读取到高速缓存中适当的位置（完全不使用 64 KB 缓冲区）。此特性可显著提高性能。

注意
分散读取不再用于远程计算机上的文件，也不再用于使用 UNC 名称（如 *\\mycomputer\myshare\mydb.db*）指定的文件。

使用适当的数据类型

数据类型存储有关特定数据集的信息，包括值的范围、可对值执行的操作以及值在内存中的存储方式。为数据使用适当的数据类型可以提高性能。例如，应避免为仅包含数字数据的值指派 CHAR 数据类型。同时请尽可能选择高效的数据类型，而不要选择开销大的数字和字符串类型。请参见“SQL 数据类型” 《SQL Anywhere 服务器 - SQL 参考》。

使用 AUTOINCREMENT 创建主键

主键值必须唯一。虽然为主键创建唯一值的方法有多种，但最有效的方法是将缺省列值设置为 AUTOINCREMENT。对于要保持唯一值的任何列，您都可以使用此缺省值。使用 AUTOINCREMENT 功能生成主键值比其它方法速度快，原因是该值由数据库服务器生成。请参见“CREATE TABLE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》和“ALTER TABLE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

使用批量操作方法

如果您需要将海量的信息装载到数据库中，则可以借助于为这些任务提供的特殊工具。

如果您将要装载大文件，一种较为有效的方法是在装载数据后在表上创建索引。

有关提高批量操作性能的信息，请参见“批量操作的性能问题”一节第 678 页。

使用延迟提交

向数据库提交更改的速率较高时，事务日志写入的速率就成为决定数据库总体性能的唯一最重要的因素。如果需要提高事务日志性能，可将 delayed_commits 选项设置为 On。此选项设置为 On 时，数据库服务器会立即响应 COMMIT 语句，而不是一直等到 COMMIT 的事务日志条目写入磁盘。此选项设置为 Off 时，应用程序必须等到 COMMIT 写入磁盘。启用 delayed_commits 选项能够避免对部分填充日志页执行多次重新写入，从而降低了事务日志的写入次数，并且可以针对单个连接或所有连接来设置此选项。启用 delayed_commits 选项存在一定风险，即如果在事务日志页刷新到磁盘之前服务器出现故障，则已提交的操作有可能丢失。请参见“delayed_commits 选项 [数据库]”一节 《SQL Anywhere 服务器 - 数据库管理》。

使用内存模式

如果应用程序能够容许丢失最近检查点之后的已提交事务，那么应用程序就可以通过使用内存模式而受益。

此模式对于需要获得较高性能且运行在具有大量可用内存的系统（通常足以在高速缓存中容纳所有的数据库文件）上的应用程序十分有用。

可以选择两种不同的内存模式。在永不写入模式下，已提交事务将不会写入磁盘上的数据库文件中。当指定永不写入模式时，在相同或不同的表上可以有多个并发的 LOAD TABLE 语句处于活动状态。如果数据库关闭或连接中断，则所有更改都会丢失。在仅检查点模式下，数据库服务器不使用事务日志，因而您无法恢复到最近的已提交事务。但是，由于启用了检查点日志，因此数据库可以恢复到最近的检查点。

有关配置内存模式以及确定此模式是否适用于您的应用程序的详细信息，请参见“-im 服务器选项”一节 《SQL Anywhere 服务器 - 数据库管理》。

需要单独授予许可的组成部分
内存模式需要一个单独的许可。请参见“单独授权的组件”一节 《SQL Anywhere 11 - 简介》。

有效使用索引

当执行查询时，SQL Anywhere 会选择如何访问每个表。索引将大大加快访问的速度。当数据库服务器无法找到合适的索引时，它会转为按顺序扫描表—此过程可能需要较长的时间。

例如，假设您需要在一个大型数据库中对员工进行搜索，但只知道他们的名字或姓氏但不知道全名。如果没有索引，SQL Anywhere 会扫描整个表。但是，如果您创建了两个索引（一个首先包含姓氏，另一个首先包含名字），SQL Anywhere 将首先扫描这两个索引，并且通常可以更快地向您返回信息。

正确选择索引可以大大提高性能。在“使用索引”一节第 66 页中介绍了如何创建和管理索引。

使用索引

虽然索引使 SQL Anywhere 能够非常有效地定位信息，但在添加索引时应多加小心。每当您插入、删除或更新行时，由于 SQL Anywhere 还必须更新所有受影响的索引，因此每个索引都会带来额外的工作量。

当索引能够使 SQL Anywhere 更为有效地访问数据时，应考虑添加索引。当索引可以避免不必要地按顺序访问大型表时，尤其应添加索引。但是，如果您在表中添加行时需要更好的性能并且不用考虑快速查找信息方面的问题，则应使用尽可能少的索引。

最好使用 [索引顾问] 来指导您完成为数据库选择一组有效索引的操作。请参见“索引顾问”一节 第 167 页。

聚簇索引

使用聚簇索引有助于存储表中的行，其顺序与索引中行出现的顺序几乎相同。请参见“索引”一节 第 596 页和“使用聚簇索引”一节第 67 页。

使用键提高查询性能

主键和外键虽然主要用于校验，但也可以提高数据库性能。

示例

以下示例说明主键如何加快查询的执行速度。

```
SELECT *
FROM Employees
WHERE EmployeeID = 390;
```

数据库服务器要执行该查询，最简单的方法是查看 Employees 表中的所有 75 行，并检查每行中的雇员 ID 是否为 390。由于只有 75 名雇员，因此这种方法不会花费很长的时间，但对于包含数千个条目的表，顺序搜索可能要花费很长的时间。

每个主键或外键所体现的参照完整性约束是由 SQL Anywhere 借助于索引（用每个主键或外键声明隐式创建）来强制实施的。EmployeeID 列是 Employees 表的主键。通过对应的主键索引，可以快速地检索到雇员编号 390。无论 Employees 表中有 100 行还是有 1000000 行，这种快速搜索方法都将使用大致相同的时间。

系统将自动为主键和外键创建单独的索引。这样，SQL Anywhere 就可以更为有效地执行许多操作。

有关主键和外键工作方式的详细信息，请参见“表间的关系”一节 《SQL Anywhere 11 - 简介》。

使用高速缓存提高性能

数据库高速缓存是内存中的特定区域，它被数据库服务器用来存储数据库页以供重复的快速访问。高速缓存中可访问的页数越多，数据库服务器从磁盘中读取数据所需的次数就越少。由于从磁盘中读取数据是相当慢的操作，因此可用的高速缓存量通常是决定性能的关键因素之一。

数据库启动时，您可以在数据库服务器命令行上指定 -c 选项来控制数据库高速缓存的大小。

数据库服务器消息窗口显示启动时的高速缓存大小，可以使用以下语句来获得当前高速缓存的大小：

```
SELECT PROPERTY( 'CacheSize' );
```

另请参见

- “-c 服务器选项”一节 《SQL Anywhere 服务器 - 数据库管理》
- “-ca 服务器选项”一节 《SQL Anywhere 服务器 - 数据库管理》
- “-ch 服务器选项”一节 《SQL Anywhere 服务器 - 数据库管理》
- “-cl 服务器选项”一节 《SQL Anywhere 服务器 - 数据库管理》

限制高速缓存内存使用

高速缓存的初始、最小和最大大小都可以通过数据库服务器命令行进行控制。

- **初始高速缓存大小** 您可以通过指定数据库服务器 -c 选项更改初始高速缓存的大小。缺省值如下：
  - **Windows Mobile** 相应的公式如下：

    ```
    max( 600 KB, min( dbsize, physical-memory ) );
    ```

    dbsize 是所启动的数据库文件大小总和，*physical-memory* 是计算机上物理内存的 25%。
  - **Windows** 相应的公式如下：

    ```
    max( 2 MB, min( dbsize, physical-memory ) );
    ```

    dbsize 是所启动的数据库文件大小总和，*physical-memory* 是计算机上物理内存的 25%。

    如果在 Windows 上使用了 AWE 高速缓存，则公式如下：

    ```
    min( 100% of available memory-128MB, dbsize );
    ```

    如果该值小于 2 MB，则不使用 AWE 高速缓存。

    有关 AWE 高速缓存的信息，请参见“-cw 服务器选项”一节 《SQL Anywhere 服务器 - 数据库管理》。
  - **Unix** 至少 8 MB。

    有关 Unix 初始高速缓存大小的信息，请参见“在 Unix 上动态调整高速缓存大小”一节 第 224 页。
- **最大高速缓存大小** 您可以通过指定数据库服务器 -ch 选项控制最大高速缓存大小。其缺省值基于由计算机上的物理内存决定的启发式算法。在 Windows Mobile 上，缺省的最大高速缓存大小是可用的程序内存量减去 4 MB。在其它非 Unix 计算机上，缺省值大概是最大非 AWE 高速缓存大小和计算机物理内存的 90% 二者中的较低者。在 Unix 中，缺省的最大高速缓存大小的计算方法如下：
  - 在 32 位 Unix 平台上，它是总物理内存的 90% 或 1,834,880 KB 的较小者。
  - 在 64 位 Unix 平台上，它是总物理内存的 90% 和 8,589,672,320 KB 的较小者。

- **最小高速缓存大小** 您可以通过指定数据库服务器 -cl 服务器选项来控制最小高速缓存大小。缺省情况下，最小高速缓存大小与初始高速缓存大小相同（Windows Mobile 除外）。Windows Mobile 上的缺省最小高速缓存大小为 600 KB。

您也可以使用 -ca 0 服务器选项来禁用动态调整高速缓存大小的功能。

下面的服务器属性可返回有关数据库服务器高速缓存的信息：

- **MinCacheSize** 返回所允许的最小高速缓存大小（以千字节为单位）。
- **MaxCacheSize** 返回所允许的最大高速缓存大小（以千字节为单位）。
- **CurrentCacheSize** 返回当前高速缓存大小（以千字节为单位）。
- **PeakCacheSize** 返回高速缓存在当前会话中已达到的最大值（以千字节为单位）。

有关获取服务器属性值的信息，请参见“数据库服务器属性”一节 《SQL Anywhere 服务器 - 数据库管理》。

另请参见

- “-c 服务器选项”一节 《SQL Anywhere 服务器 - 数据库管理》
- “-ca 服务器选项”一节 《SQL Anywhere 服务器 - 数据库管理》
- “-ch 服务器选项”一节 《SQL Anywhere 服务器 - 数据库管理》
- “-cl 服务器选项”一节 《SQL Anywhere 服务器 - 数据库管理》

动态调整高速缓存大小

可以使用 SQL Anywhere 自动重新设置数据库高速缓存的大小。然而，动态调整高速缓存大小的有效性要受到运行数据库服务器的操作系统的限制，而可用物理内存数量也会对其产生影响。

完全使用**动态调整高速缓存大小**时，内存分配不足不会对数据库服务器的性能造成影响。当数据库服务器可以有效地使用更多内存（只要内存可用）时，高速缓存将增大，当其它应用程序需要高速缓存时，高速缓存将变小。这样可以防止数据库服务器对系统中的其它应用程序产生影响。

通常，动态高速缓存调整功能每分钟评估一次高速缓存需求。然而，当启动新数据库或某一文件显著增大时，可以在随后的 30 秒内将评估间隔增加为每隔 5 秒进行一次。经过最初的三十秒期间后，取样速率会下降到每分钟一次。文件显著增大是指，自数据库启动以后或者自上次发生触发取样速率提高的增长后文件增大了 1/8。此变化可在数据库动态启动时和插入大量数据时更快速地适应高速缓存大小，从而可以进一步提高性能。

利用动态高速缓存调整功能，您无需显式配置数据库高速缓存。

当使用 Address Windowing Extensions (AWE) 高速缓存时，动态调整高速缓存大小的功能将被禁用。不能在 Windows Mobile 上使用 AWE 高速缓存。

有关 AWE 高速缓存的详细信息，请参见“-cw 服务器选项”一节 《SQL Anywhere 服务器 - 数据库管理》。

在 Windows 上动态调整高速缓存大小

在 Windows 和 Windows Mobile 上，数据库服务器将每分钟计算一次高速缓存和运行统计信息并计算出最佳的高速缓存大小。数据库服务器将计算出目标高速缓存大小，该大小使用当前未使用的所有物理内存（保留下来供系统使用的大约 5 MB 内存除外）。目标高速缓存大小决不会小于指定或隐式的最小高速缓存大小。目标高速缓存大小决不会超过指定或隐式的最大高速缓存大小（即所有打开的数据库和临时文件的大小与主堆大小的总和）。

为了避免高速缓存大小波动，数据库服务器会以递增的方式增加高速缓存大小。它不是立即将高速缓存大小调整到目标值，而是按当前高速缓存大小和目标高速缓存大小之间差值的 75% 逐次调整高速缓存大小。

Windows 可以使用 Address Windowing Extensions (AWE) 来支持较大的高速缓存大小，方法是在启动数据库服务器时指定 -cw 命令行选项。AWE 高速缓存不支持动态高速缓存调整。Windows Mobile 不支持 AWE 高速缓存。请参见“-cw 服务器选项”一节 《SQL Anywhere 服务器 - 数据库管理》。

在 Unix 上动态调整高速缓存大小

在 Unix 上，数据库服务器使用交换空间和内存来管理高速缓存大小。交换空间是大多数（但不是全部）Unix 操作系统上的系统范围资源。在本节，内存和交换空间的总和称作**系统资源**。有关详细信息，请参见相应的操作系统文档。

在启动时，数据库会从系统资源中分配指定的最大高速缓存大小。它会将最大高速缓存大小的一部分装载到内存（初始高速缓存大小）中，并保留其余部分作为交换空间。

在数据库服务器关闭之前，数据库服务器所用的系统资源总量是恒定的，但在内存中装入的比例会出现变化。数据库服务器会每分钟计算一次高速缓存和运行统计信息。如果数据库服务器处于繁忙状态并且需要很多内存，它可能会将高速缓存页从交换空间移动到内存中。如果系统中的其它进程需要内存，则数据库服务器可能会将高速缓存页从内存中移出，移至交换空间中。

初始高速缓存大小

缺省情况下，将使用基于可用系统资源的启发式算法来指定初始高速缓存大小。初始高速缓存大小始终小于数据库总大小的 1.1 倍。

如果初始高速缓存大小大于可用系统资源的 3/4，则数据库服务器将退出并报告 [内存不足] 错误。

您可以使用 -c 选项来更改初始高速缓存的大小。请参见“-c 服务器选项”一节 《SQL Anywhere 服务器 - 数据库管理》。

最大高速缓存大小

最大高速缓存必须小于计算机上的可用系统资源。缺省情况下，将使用基于计算机上可用系统资源和总物理内存的启发式算法来指定最大高速缓存大小。高速缓存大小决不会超过指定或隐式的最大高速缓存大小（即所有打开的数据库和临时文件的大小与主堆大小的总和）。

如果您指定的最大高速缓存大小大于可用系统资源，则数据库服务器将退出并报告 [内存不足] 错误。如果您指定的最大高速缓存大小大于可用内存，数据库服务器将发出性能降低警告，但不会退出。

数据库服务器将从系统资源中分配所有*最大*高速缓存大小，并且在数据库服务器退出之前不会放弃该大小。您应该确保所选择的最大高速缓存大小提供了较优的 SQL Anywhere 性能，并且为其它应用程序保留了足够的空间。缺省最大高速缓存大小的公式就是要尽量达到这一平衡的启发式算法。只有当缺省值不适用于您的系统时，才需要调整缺省值。

可以使用 -ch 服务器选项来设置最大高速缓存大小，并限制高速缓存自动增长。有关详细信息，请参见“-ch 服务器选项”一节 《SQL Anywhere 服务器 - 数据库管理》。

最小高速缓存大小

如果指定了 -c 选项，则最小高速缓存大小与初始高速缓存大小相同。如果没有指定 -c 选项，则 UNIX 上的最小高速缓存大小是 8 MB。

您可以使用 -cl 服务器选项来调整最小高速缓存大小。请参见“-cl 服务器选项”一节 《SQL Anywhere 服务器 - 数据库管理》。

监控高速缓存大小

以下统计信息包含在 Windows 性能监控器和数据库的 property 函数中。

- **CurrentCacheSize** 当前高速缓存大小 (KB)
- **MinCacheSize** 允许的最小高速缓存大小 (KB)
- **MaxCacheSize** 允许的最大高速缓存大小 (KB)
- **PeakCacheSize** 高速缓存大小峰值 (KB)

注意
Windows 提供了 Windows 性能监控器。

有关这些属性的详细信息，请参见“数据库服务器属性”一节 《SQL Anywhere 服务器 - 数据库管理》。

有关监控性能的信息，请参见“监控数据库性能”一节第 192 页。

使用高速缓存预热

高速缓存预热旨在帮助减少对数据库执行的初始查询的执行时间。通过在数据库服务器的高速缓存中预装载上次启动数据库时引用的数据库页，可以实现这一点。如果在每次启动数据库时都对其执行相同或相似的查询，则预热高速缓存可以提高性能。

您可以在数据库服务器命令行上控制高速缓存预热设置。当数据库启动并且打开了高速缓存预热时，可能会发生两个活动：数据库页的收集和高速缓存重装（预热）。

对所引用的数据库页的收集由 -cc 数据库服务器选项控制，缺省情况下处于开启状态。启用数据库页收集后，数据库服务器将跟踪数据库启动后所请求的每个数据库页，直到出现下列情况之一：收集的页数达到最大值（该值取决于高速缓存大小和数据库大小），收集率降至最小阀值以下，或数

据库关闭。请注意，数据库服务器控制页的最大数量和收集阈值。完成收集后，会将引用的页记录在数据库中以便在下次启动数据库时使用它们预热高速缓存。

高速缓存预热（重装）在缺省情况下处于开启状态，并且由 -cr 数据库服务器选项控制。为了预热高速缓存，数据库服务器会检查数据库是否包含以前记录的页集合。如果包括，数据库服务器会将相应的页装载到高速缓存中。在高速缓存装载页时，数据库仍旧可以处理请求，但是，如果在数据库中检测到大量 I/O 活动，预热可能会停止。高速缓存预热在这种情况下停止，是为了避免对未包含在重装到高速缓存中的页集中的页进行访问的查询性能降低。如果您希望有关高速缓存预热的消息显示在数据库服务器消息窗口中，可以指定 -cv 选项。

有关用于高速缓存预热的数据库服务器选项的详细信息，请参见“-cc 服务器选项”一节 《SQL Anywhere 服务器 - 数据库管理》、“-cr 服务器选项”一节 《SQL Anywhere 服务器 - 数据库管理》和“-cv 服务器选项”一节 《SQL Anywhere 服务器 - 数据库管理》。

使用压缩功能

通过对一个连接或所有连接启用压缩并调整包压缩的最小大小限值，在某些情况下可以大大提高性能。

为了确定启用压缩是否有益，请在某个生产环境中使用通信压缩之前，对网络和使用的应用程序进行性能分析。

启用压缩会增加数据包中存储的信息量，从而减少了传输一组特定数据所需发送的数据包数量。通过减少数据包的数量，可以更快地传输数据。

通过指定压缩阈值，可以选择希望数据包压缩的最小大小。最佳的压缩阈值可能会受到多种因素的影响，其中包括所用网络的类型和速度。

另请参见

- “调整通信压缩设置以改善性能”一节 《SQL Anywhere 服务器 - 数据库管理》
- “Compress 连接参数 [COMP]”一节 《SQL Anywhere 服务器 - 数据库管理》
- “CompressionThreshold 连接参数 [COMPTH]”一节 《SQL Anywhere 服务器 - 数据库管理》

在校验表时使用 WITH EXPRESS CHECK 选项

如果您发现校验具有较小高速缓存的大型数据库需要很长的时间，则可以使用以下两个选项之一来减少所用的时间。通过使用含 VALIDATE TABLE 语句的 WITH EXPRESS CHECK 选项，或带有校验实用程序 (dbvalid) 的 -fx 选项，都可大大增加表的校验速度。

另请参见

- “在校验数据库时提高性能”一节 《SQL Anywhere 服务器 - 数据库管理》
- “VALIDATE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “校验实用程序 (dbvalid)”一节 《SQL Anywhere 服务器 - 数据库管理》

在查询处理中使用工作表（使用 All-rows 优化目标）

工作表是实例化的临时结果集，这些临时结果集在执行查询的过程中创建。当 SQL Anywhere 确定使用工作表所需的开销少于其它策略时，就会使用工作表。通常，读取前几行所需的时间在使用工作表时要多一些，但是在某些情况下，如果可以使用工作表，检索所有行所需的开销则可能会大大降低。由于存在这种差异，SQL Anywhere 会根据 optimization_goal 设置选择不同的策略。缺省值是前几行 (first-row)。如果该值设置为 first-row，SQL Anywhere 将尝试避免使用工作表。如果该值设置为 All-rows，SQL Anywhere 将在工作表能够减少查询的总执行开销时使用工作表。

有关 optimization_goal 设置的详细信息，请参见“optimization_goal 选项 [数据库]”一节 《SQL Anywhere 服务器 - 数据库管理》。

在以下情况中将使用工作表：

- 当查询包含 ORDER BY、GROUP BY 或 DISTINCT 子句，并且 SQL Anywhere 不使用索引对行进行排序时。如果存在适当的索引且 optimization_goal 设置是 First-row 时，SQL Anywhere 将避免使用工作表。但是，当 optimization_goal 设置为 All-rows 时，使用索引读取查询的所有行可能会比建立工作表并将行排序开销更大。如果优化目标设置为 All-rows，SQL Anywhere 会选择开销较低的策略。对于 GROUP BY 和 DISTINCT，基于散列的算法将使用工作表，但通常在读取查询中的所有行时更为有效。
- 选择了散列连接算法时。在这种情况下，将使用多个工作表来存储中间结果（如果内存不够存放），并使用一个工作表来存储连接的结果。
- 当用敏感值打开游标时。在这种情况下，将创建一个工作表来保存基表的行标识符和主键。当向前从查询中读取行时，将填充该工作表。但是，如果从游标中读取最后一行，则将填充整个表。
- 当用不敏感的语义打开游标时。在这种情况下，将在查询打开时用查询结果填充工作表。
- 当多行 UPDATE 操作正在执行，并且所更新的列出现在 Update 命令的 WHERE 子句中，或出现在用于 Update 操作的索引中时
- 当多行 UPDATE 或 DELETE 操作在 WHERE 子句中有一个引用所修改的表的子查询时
- 当从 SELECT 语句中执行 INSERT，并且该 SELECT 语句引用插入表时
- 当执行多行 INSERT、UPDATE 或 DELETE 操作，并且在表中定义可能在操作过程中触发的相应触发器时

在这些情况下，操作所影响的记录会进入工作表。在某些情况（如键集决定的游标）下，将在工作表上建立临时索引。在查询结果出现之前，将所需记录提取到工作表中的操作可能会需要大量的时间。通过创建可用来在上述第一种情况下执行排序的索引，可以减少检索前几行所需的时间。但是，如果使用工作表，则可以减少读取所有行所需的时间，因为它支持基于散列和合并排序的查询算法。这些算法采用顺序 I/O，这比用于索引扫描的随机 I/O 要快一些。

优化程序将分析每个查询，以确定工作表是否能够提供最佳性能。用户无需执行任何操作即可利用这些优化。

注意

上述的 INSERT、UPDATE 和 DELETE 情况通常不会有性能问题，因为它们通常是一次性操作。但是，如果出现问题，您可能还能够改写该命令以避免冲突并避免建立工作表。这并不总是可能的。

应用程序分析教程

目录

使用应用程序分析教程来学习如何使用 [**应用程序分析向导**] 和 [**数据库跟踪向导**] 分析常见的性能问题，这些问题包括死锁、语句执行速度缓慢、索引碎片、表碎片和过程执行速度缓慢。

小心

这些教程使用的是您创建的测试数据库 *app_profiling.db*，而不是示例数据库 (*demo.db*)。不要使用示例数据库完成教程。

必须具有 PROFILE 权限才能执行应用程序分析。要获得应用程序分析教程，请以具有 DBA 权限的用户身份进行连接。

教程：诊断死锁

使用本节中的教程来学习如何利用 [**数据库跟踪向导**] 来查看数据库中可能发生的死锁。可以使用 [**数据库跟踪向导**] 分析在哪些情况下发生死锁以及导致死锁的连接。

当两个或多个事务互相阻塞时，就会发生死锁。例如，事务 A 需要访问表 B，但表 B 被事务 B 锁定。事务 B 需要访问表 A，但表 A 被事务 A 锁定。这时就发生循环阻塞冲突。

当返回 SQLCODE -306 和 -307 时，就明显表示发生死锁。为解决死锁，SQL Anywhere 会自动回退形成该死锁的最后一条语句。如果不断回退语句，则会出现性能问题。

第 1 课：创建测试数据库

按照以下过程使用示例数据库中的数据创建测试数据库 *app_profiling.db*。该测试数据库用于所有应用程序分析教程。

♦ 创建测试数据库

1. 创建目录 *C:\AppProfilingTutorial*。
2. 打开命令提示符，键入以下命令来创建测试数据库 *app_profiling.db*。*samples-dir* 是您的示例目录的位置：

   ```
   dbunload -c "UID=DBA;PWD=sql;DBF=samples-dir\demo.db" -an C:
   \AppProfilingTutorial\app_profiling.db
   ```

   例如，在默认位置安装了 Windows XP 和 SQL Anywhere 的计算机上，该命令为：

   ```
   dbunload -c "UID=DBA;PWD=sql;DBF=C:\Documents and Settings\All Users
   \Documents\SQL Anywhere 11\Samples\demo.db" -an C:\AppProfilingTutorial
   \app_profiling.db
   ```

   有关 *samples-dir* 位置的详细信息，请参见“示例目录”一节 《SQL Anywhere 服务器 - 数据库管理》。

提示
在应用程序分析教程中，跟踪信息就存储在要进行分析的同一数据库 (*app_profiling.db*) 中。但如果您要进行分析的数据库承受的负荷很重，则应考虑将跟踪数据存储到另一个单独数据库中，以避免影响生产数据库的性能。

另请参见

- “应用程序分析向导”一节第 161 页
- “使用诊断跟踪进行高级应用程序分析”一节第 171 页
- “PROFILE 特权”一节 《SQL Anywhere 服务器 - 数据库管理》

第 2 课：创建死锁

本教程假定您已创建了测试数据库。如果尚未创建测试数据库，请参见“第 1 课：创建测试数据库”一节第 230 页。

提示
可以将本教程中的 SQL 语句复制并粘贴到 Interactive SQL 中。

♦ 形成死锁

1. 启动 Sybase Central，并使用用户 ID **DBA** 和口令 **sql** 连接到测试数据库 *app_profiling.db*。

   如果不熟悉如何启动 Sybase Central 和连接到数据库，请参见“连接到本地数据库”一节《SQL Anywhere 服务器 - 数据库管理》。

2. 在左窗格中，单击 *app_profiling - DBA*，然后选择 [**文件**] » [**打开 Interactive SQL**]。

   Interactive SQL 启动并连接到 *app_profiling.db* 数据库。

3. 在 Interactive SQL 中，运行以下 SQL 语句：

   a. 创建两个表

   ```
   CREATE TABLE "DBA"."deadlock1" (
      "id" UNSIGNED BIGINT NOT NULL DEFAULT AUTOINCREMENT,
      "val" CHAR(1) );
   CREATE TABLE "DBA"."deadlock2" (
      "id" UNSIGNED BIGINT NOT NULL DEFAULT AUTOINCREMENT,
      "val" CHAR(1) );
   ```

   b. 向各表中插入值

   ```
   INSERT INTO "deadlock1"("val") VALUES('x');
   INSERT INTO "deadlock2"("val") VALUES('x');
   ```

   c. 创建两个过程，稍后用它们形成死锁

   ```
   CREATE PROCEDURE "DBA"."proc_deadlock1"(  )
      BEGIN
         LOCK TABLE "DBA"."deadlock1" IN EXCLUSIVE MODE;
         WAITFOR DELAY '00:00:20:000';
         UPDATE deadlock2 SET val='y';
      END;
   CREATE PROCEDURE "DBA"."proc_deadlock2"(  )
      BEGIN
         LOCK TABLE "DBA"."deadlock2" IN EXCLUSIVE MODE;
         WAITFOR DELAY '00:00:20:000';
         UPDATE deadlock1 SET val='y';
      END;
   ```

   d. 提交对数据库所做的更改

   ```
   COMMIT;
   ```

4. 退出 Interactive SQL。

第 3 课：捕获死锁数据

[**数据库跟踪向导**] 可用于创建诊断跟踪会话。该跟踪会话将捕获死锁数据。

♦ **捕获死锁数据**

1. 在 Sybase Central 中，选择 [**模式**] » [**应用程序分析**]。

   如果出现 [**应用程序分析向导**]，则单击 [**取消**]。
2. 启动 [**数据库跟踪向导**]。
   a. 在左窗格中，单击 *app_profiling - DBA*，然后选择 [**文件**] » [**跟踪**]。
   b. 在 [**欢迎**] 页面上，单击 [**下一步**]。
   c. 在 [**跟踪详细信息级别**] 页面上，选择 [**高详细信息（建议短期、集中监控使用）**]，然后单击 [**下一步**]。
   d. 在 [**编辑跟踪级别**] 页面上，单击 [**下一步**]。
   e. 在 [**创建外部数据库**] 页面上，选择 [**不创建新数据库。我将使用现有跟踪数据库**]，然后单击 [**下一步**]。
   f. 在 [**启动跟踪**] 页面上，选择 [**在此数据库中保存跟踪数据**]。
   g. 若不想对存储的跟踪数据量设置任何限制，请选择 [**无限制**]，然后单击 [**完成**]。
   h. 单击 [**完成**]。
3. 形成死锁。
   a. 在 Sybase Central 的左窗格中，选择 *app_profiling - DBA* 数据库，然后选择 [**文件**] » [**打开 Interactive SQL**]。

      Interactive SQL 启动并连接到 *app_profiling - DBA* 数据库。
   b. 重复前面的步骤，打开另一个 Interactive SQL 窗口。
   c. 在一个 Interactive SQL 窗口中，运行以下 SQL 语句：

      ```
      CALL "DBA"."proc_deadlock1"();
      ```
   d. 在 20 秒内于第二个 Interactive SQL 窗口中运行以下 SQL 语句：

      ```
      CALL "DBA"."proc_deadlock2"();
      ```

      片刻之后，将出现 [**ISQL 错误**] 窗口，表明已检测到死锁。这是因为 proc_deadlock1 需要访问 deadlock2 表，而 deadlock2 表被 proc_deadlock2 锁定。同时 proc_deadlock2 需要访问 deadlock1 表，但 deadlock1 表被 proc_deadlock1 锁定。
   e. 单击 [**确定**]。
4. 关闭这两个 Interactive SQL 窗口。
5. 要停止跟踪会话，在 Sybase Central 中选择 *app_profiling - DBA* 数据库，然后选择 [**文件**] » [**跟踪**] » [**停止跟踪并保存**]。

第 4 课：查看阻塞的连接数据

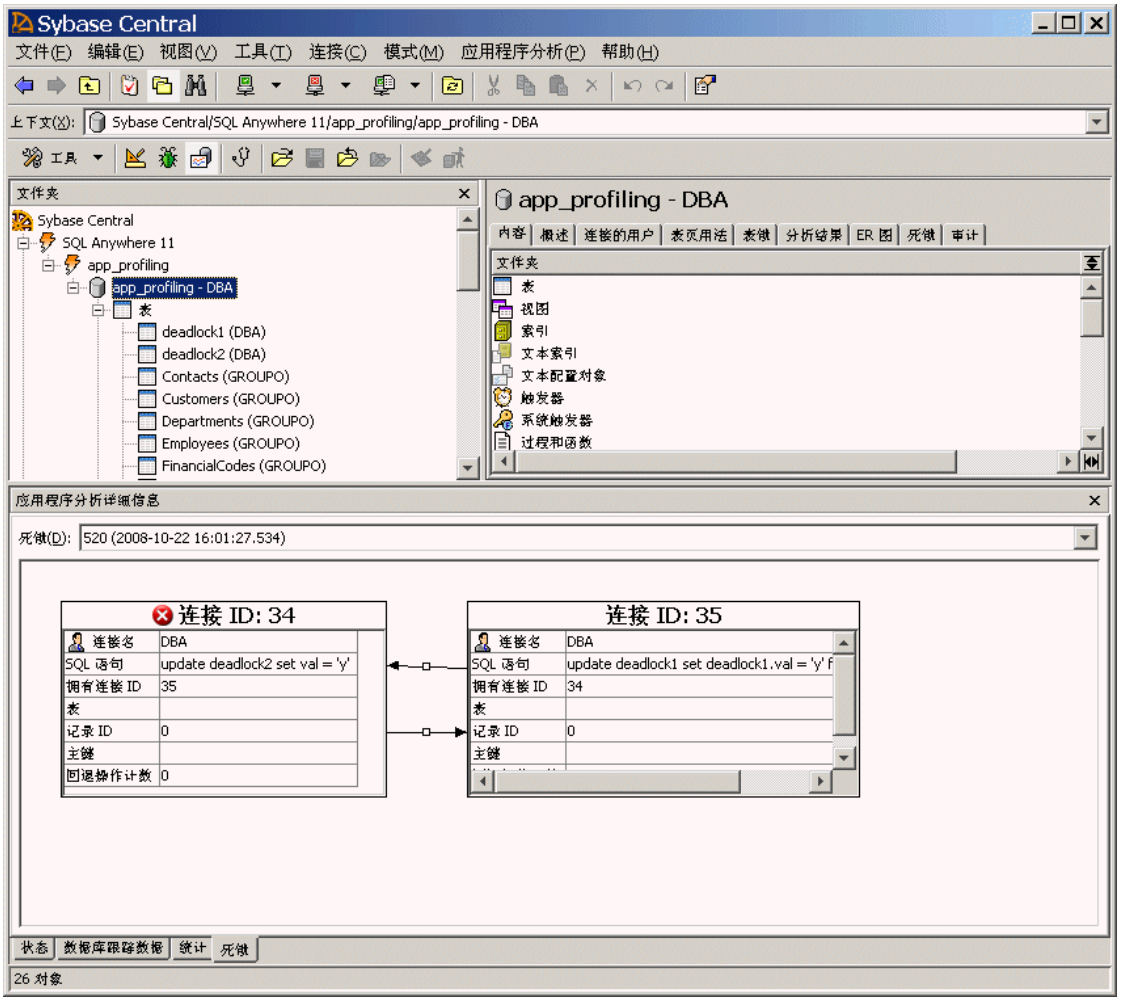
[**应用程序分析**] 模式提供死锁所涉及的连接的图形表示。它还提供 [**连接块**] 选项卡，该选项卡提供有关阻塞的连接的其它信息。

♦ **查看阻塞的连接数据**

1. 打开在跟踪会话期间创建的分析文件。
   a. 在 Sybase Central 中，选择 [**应用程序分析**] » [**打开分析文件或连接到跟踪数据库**]。
   b. 选择 [**在跟踪数据库中**]。
   c. 单击 [**打开**]。
   d. 单击 [**标识**] 选项卡，然后在 [**用户 ID**] 字段中键入 **DBA**，在 [**口令**] 字段中键入 **sql**。
   e. 单击 [**数据库**] 选项卡，在 [**数据库文件**] 字段中，浏览至 *app_profiling - DBA*，然后选中它。
   f. 单击 [**确定**]。
2. 查看死锁的图形表示。
   a. 在 [**应用程序分析详细信息**] 窗格中，单击 [**状态**] 选项卡，然后从 [**记录会话 ID**] 列表中选择时间最近的 ID。

      如果未出现 [**应用程序分析详细信息**] 窗格，则选择 [**视图**] » [**应用程序分析详细信息**]。
   b. 在 [**应用程序分析详细信息**] 窗格的底部，单击 [**死锁**] 选项卡。将出现时间最近的死锁。单击 [**死锁**] 列表来查看其它死锁。

   下图显示了 UPDATE 语句是如何形成死锁条件的。



死锁中涉及的各个连接由具有以下字段的表来表示：

- **连接名称** 此字段显示打开连接的用户 ID。
- **SQL 语句** 此字段显示死锁中涉及的实际语句。在本例中，死锁由 UPDATE 语句引起，在您从每个 Interactive SQL 实例执行的过程中即可找到该语句。
- **拥有连接 ID** 此字段显示阻塞当前连接的那个连接的 ID。
- **记录 ID** 此字段显示当前连接发生阻塞时所在行的 ID。
- **回退操作计数** 此字段显示由于死锁而必须回退的操作数。在本例中，过程仅包含 UPDATE 语句，因此计数为 0。

第 5 课：查看死锁数据

使用以下过程查看其它死锁数据，如死锁发生的频率和持续的时间等。请使用 [**连接块**] 选项卡来查看在数据库跟踪会话期间记录的所有死锁的列表。

♦ 查看死锁数据

1. 在 [**应用程序分析详细信息**] 窗格中，单击 [**数据库跟踪数据**] 选项卡。
2. 单击 [**连接块**] 选项卡，该选项卡就在 [**数据库跟踪数据**] 选项卡的上方。

   将出现 [**连接块**] 选项卡，显示阻塞时间、解除阻塞时间和各个连接受阻塞的持续时间。

另请参见

- “事务阻塞和死锁”一节第 118 页
- “选择诊断跟踪级别”一节第 173 页
- “死锁”一节第 118 页
- “使用诊断跟踪进行高级应用程序分析”一节第 171 页

教程：诊断执行速度慢的语句

利用本节中的教程来学习如何使用 [**数据库跟踪向导**] 来查看语句的执行时间，以及如何识别那些看起来运行缓慢的语句。

当数据库服务器处理某语句的时间比较长时，该语句就出现了执行速度慢的问题。处理时间长可能是由于若干问题造成，例如数据库设计不正确、索引使用不当、索引和表碎片、高速缓存大小过小等等。语句的构成不周密也可能导致执行速度慢，否则会使用更有效的快捷方式来实现结果。

本教程并不说明如何重写执行速度慢的语句，因为每个语句都会有特殊的要求。但本教程会说明在何处查找执行时间，以及在使用替代语法在重写查询时，如何比较执行时间。

另请参见

- “查询数据”第 253 页
- “连接：从多个表检索数据”第 361 页
- “使用子查询”第 471 页

第 1 课：创建诊断跟踪会话

[**数据库跟踪向导**] 用于创建诊断跟踪会话。跟踪会话可捕获处理中的语句数据（包括持续时间）。

本教程假定您已创建了测试数据库。如果尚未创建测试数据库，请参见“第 1 课：创建测试数据库”一节第 230 页。

提示
可以将本教程中的 SQL 语句复制并粘贴到 Interactive SQL 中。

♦ 创建诊断跟踪会话

1. 启动 Sybase Central，并使用用户 ID **DBA** 和口令 **sql** 连接到测试数据库 *app_profiling.db*。

   如果不熟悉如何启动 Sybase Central 和连接到数据库，请参见“连接到本地数据库”一节《SQL Anywhere 服务器 - 数据库管理》。

2. 启动 [**数据库跟踪向导**]。
   a. 在 Sybase Central 中，选择 [**模式**] » [**应用程序分析**]。如果出现 [**应用程序分析向导**]，则单击 [**取消**]。
   b. 选择 [**文件**] » [**配置和启动跟踪**]。
   c. 在 [**欢迎**] 页面上，单击 [**下一步**]。
   d. 在 [**跟踪详细信息级别**] 页面上，选择 [**高详细信息（建议短期、集中监控使用）**]，然后单击 [**下一步**]。
   e. 在 [**编辑跟踪级别**] 页面上，单击 [**下一步**]。
   f. 在 [**创建外部数据库**] 页面上，选择 [**不创建新数据库**]，然后单击 [**下一步**]。
   g. 在 [**启动跟踪**] 页面上，选择 [**在此数据库中保存跟踪数据**]。

h. 若不想对存储的跟踪数据量设置任何限制，请选择 [**无限制**]，然后单击 [**完成**]。

3. 在 Sybase Central 的左窗格中，选择 *app_profiling - DBA* 数据库，然后选择 [**文件**] » [**打开 Interactive SQL**]。

   Interactive SQL 启动并连接到 *app_profiling - DBA* 数据库。

4. 在 Interactive SQL 中，运行以下 SQL 语句。

```
SELECT SalesOrderItems.ID, LineID, ProductID, SalesOrderItems.Quantity,
ShipDate
FROM SalesOrderItems, SalesOrders
WHERE SalesOrders.CustomerID = 105 AND
   SalesOrderItems.ID=SalesOrders.ID;
```

5. 在 Interactive SQL 中，运行以下 SQL 语句。此查询返回的结果与上一查询相同，但使用的是不相关的子查询。

```
SELECT *
FROM SalesOrderItems
WHERE SalesOrderItems.ID IN (
   SELECT SalesOrders.ID
   FROM SalesOrders
   WHERE SalesOrders.CustomerID = 105 );
```

6. 退出 Interactive SQL。

7. 在 Sybase Central 中，选中该数据库，然后选择 [**文件**] » [**跟踪**] » [**停止跟踪并保存**] 来停止跟踪会话。

有关 [**数据库跟踪向导**] 的信息，请参见“使用诊断跟踪进行高级应用程序分析”一节第 171 页。

第 2 课：查看由数据库服务器处理的语句

通过使用位于 Sybase Central 的 [**应用程序分析**] 窗格中的 [**概览**] 和 [**详细信息**] 选项卡，您可以确定数据库服务器在哪些语句上花费的处理时间最多。

♦ 查看由数据库服务器处理的语句

1. 打开分析文件。

   a. 在 Sybase Central 中，选择 [**模式**] » [**应用程序分析**]。如果出现 [**应用程序分析向导**]，则单击 [**取消**]。

   b. 选择 [**应用程序分析**] » [**打开分析文件或连接到跟踪数据库**]。

   c. 选择 [**在跟踪数据库中**]，然后单击 [**打开**]。

   d. 单击 [**标识**] 选项卡，然后在 [**用户 ID**] 字段中键入 **DBA**，在 [**口令**] 字段中键入 **sql**。

   e. 单击 [**数据库**] 选项卡，在 [**数据库文件**] 字段中，浏览至 *app_profiling - DBA*，然后选中它。

   f. 单击 [**确定**]。

      如果窗口底部未出现 [**应用程序分析详细信息**] 窗格，则选择 [**视图**] » [**应用程序分析详细信息**]。

2. 检查在跟踪会话期间处理的语句的执行时间。

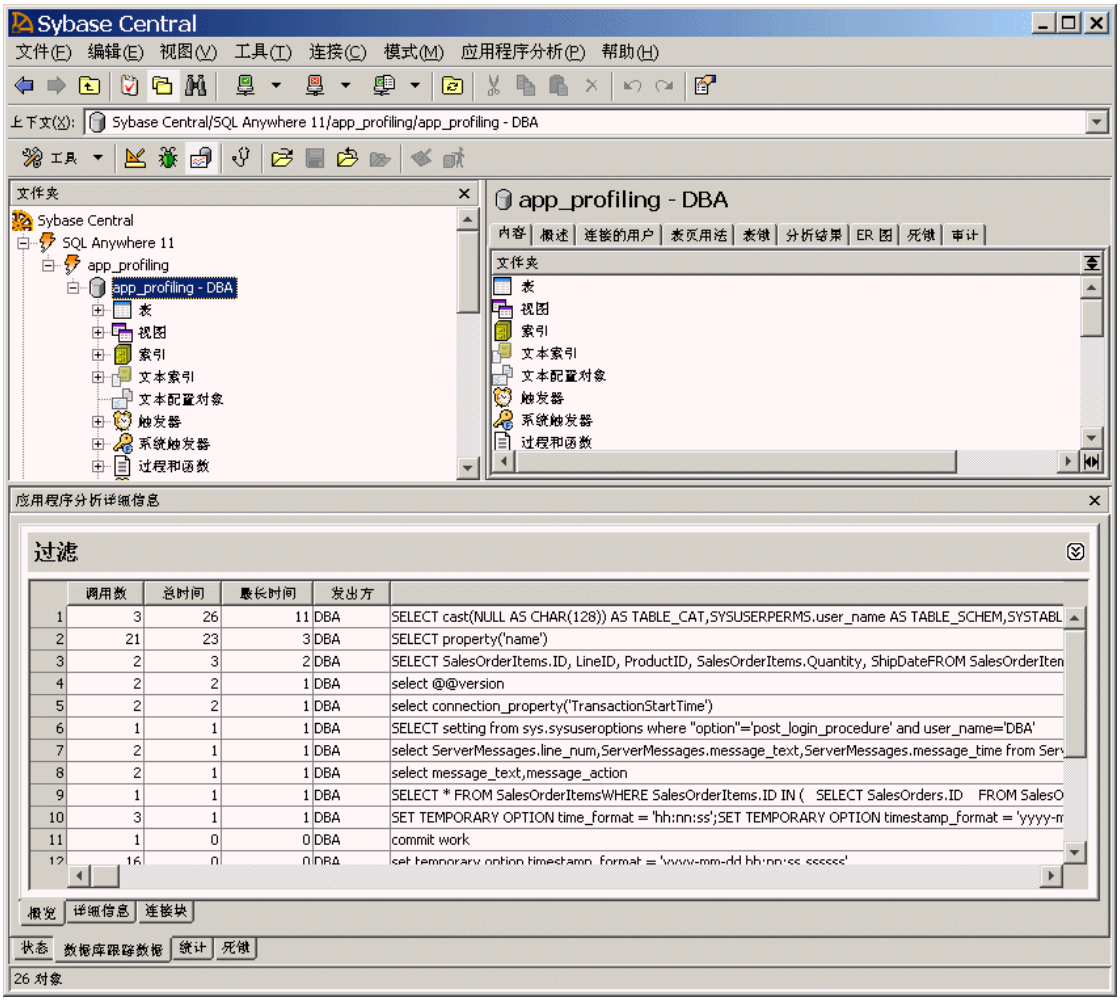
a. 在 [**应用程序分析详细信息**] 窗格的 [**状态**] 选项卡上，从 [**记录会话 ID**] 字段中选择时间最近的 ID（最高编号），然后单击 [**数据库跟踪数据**] 选项卡。

b. 将出现该会话的数据。

   在 [**概览**] 选项卡中，出现在会话期间执行的 SQL 语句。您还会看到更多其它语句。这是因为您自动执行的语句导致了其它语句的执行（例如，触发器）。

   [**概览**] 选项卡将相似的语句组合到一起，并汇总调用总数和处理这些语句所花费的总时间。SELECT、INSERT、UPDATE 和 DELETE 语句按它们所引用的表、列及表达式组合到一起。其它语句则作为一个整体组合到一起（例如，所有 CREATE TABLE 语句在 [**概览**] 选项卡中以单个条目的形式显示）。语句也可能在 [**概览**] 选项卡中显示为高成本语句，因为该语句是一个高成本语句，也可能是因为它执行频率比较高。

   使用 [**总时间**] 列和 [**最长时间**] 列来检查您在本教程中先前执行的两个查询的执行时间。第一个查询显示执行总时间为 20 毫秒。第二个查询显示的执行时间较短（16 毫秒），这说明使用了不相关子查询的第二个查询可能是要使用的更有效的语法。



3. 要在 [**概览**] 选项卡中查看有关任一 SQL 语句的其它信息，请右键单击该语句并选择 [**显示所选概览 SQL 语句的详细 SQL 语句**]。
   - 要查看执行该语句的连接的相关信息，请右键单击该语句并选择 [**查看所选语句的连接详细信息**]。
   - 要查看该语句使用的执行计划，请在 [**详细信息**] 选项卡中右键单击该语句，然后选择 [**查看所选语句的更多 SQL 语句详细信息**]。

     将出现 [**SQL 语句详细信息**] 窗口，显示该语句的全文及该语句所处上下文的相关详细信息。请注意，显示的语句文本可能与您执行的原始 SQL 语句不符。[**SQL 语句详细信息**] 窗口会以语句的重写形式显示该语句，因为它是由数据库服务器处理的。例如，视图上的查询可能看上去截然不同，因为视图定义通常在执行查询时被优化程序重写。

     要查看执行计划，请单击 [**查询信息**] 选项卡。

有关执行计划中显示的项的详细信息，请参见“读取执行计划”一节第 569 页。

有关相关子查询和不相关子查询的信息，请参见“使用子查询”第 471 页。

有关使用 [**概览**] 和 [**详细信息**] 选项卡的信息，请参见“执行请求跟踪分析”一节第 185 页。

教程：诊断索引碎片

利用本节中的教程来学习如何使用 [**应用程序分析向导**] 来确定您的数据库是否有不可接受的索引碎片级别。

创建索引后，将读取表数据并按照逻辑顺序将索引的值记录到索引页中。随着数据在表中的变化，可以在现有值之间插入新索引值。要保持索引值的逻辑顺序，数据库服务器可能需要创建新索引页来容纳被移动的现有值。通常，新页面与最初存储这些值的页面不相邻。这种索引页顺序的累积递降被称为索引碎片。

出现索引碎片的表现是，对于那些连续在其中插入、更新和删除大块的行的表，通常执行的查询在表上的执行时间变长。

第 1 课：建立索引碎片

本教程假定您已创建了测试数据库。如果尚未创建测试数据库，请参见“第 1 课：创建测试数据库”一节第 230 页。

提示
可以将本教程中的 SQL 语句复制并粘贴到 Interactive SQL 中。

♦ 建立索引碎片

1. 启动 Sybase Central，并使用用户 ID **DBA** 和口令 **sql** 连接到测试数据库 *app_profiling.db*。

   如果不熟悉如何启动 Sybase Central 和连接到数据库，请参见“连接到本地数据库”一节《SQL Anywhere 服务器 - 数据库管理》。

2. 在左窗格中，选择 *app_profiling* - *DBA* 数据库，然后选择 [**文件**] » [**打开 Interactive SQL**]。

   Interactive SQL 启动并连接到 *app_profiling* - *DBA* 数据库。

3. 在 Interactive SQL 中，运行以下 SQL 语句以引入索引碎片。这些语句可能需要几分钟来完成。

```
CREATE TABLE fragment ( id INT );
CREATE INDEX idx_fragment ON fragment ( id );
INSERT INTO fragment SELECT * FROM sa_rowgenerator ( 0, 100000 );
DELETE FROM fragment WHERE MOD ( id, 2 ) = 0;
INSERT INTO fragment SELECT * FROM sa_rowgenerator ( 0, 100000 );
INSERT INTO fragment SELECT * FROM sa_rowgenerator ( 0, 100000 );
COMMIT;
```

4. 退出 Interactive SQL。

第 2 课：识别索引碎片

使用此过程来识别索引碎片并学习在何处查找索引碎片警告。建议您定期检查产品数据库中的碎片警告。

注意
您在上一过程中执行的语句引入了索引碎片。但是，在某些系统中产生的碎片可能不足以产生此过程中描述的警告及建议。

♦ **识别索引碎片**

1. 在 Sybase Central 中，选择 [**模式**] » [**应用程序分析**]。

   如果未出现 [**应用程序分析向导**]，则选择 [**应用程序分析**] » [**打开应用程序分析向导**]。
2. 在 [**欢迎**] 页面上，单击 [**下一步**]。
3. 在 [**分析选项**] 页面中，选择 [**基于数据库模式的总体数据库性能**]，然后单击 [**下一步**]。
4. 在 [**分析文件**] 页面的 [**将分析保存到以下文件**] 字段中，键入 *C:\AppProfilingTutorial*。
5. 单击 [**完成**]。

   [**应用程序分析详细信息**] 窗格中出现建议的列表。
6. 要查看更多详细信息，双击 [**碎片索引**]。将出现 [**建议**] 窗口，其中包含您可以运行的用来解析索引碎片的 SQL 语句。

第 3 课：检查表的索引密度

要定期检查一个表的索引密度，请运行 sa_index_density 系统过程。密度值范围介于 0 和 1 之间。该值越接近 1，说明索引碎片越小。该值小于 0.5 则说明索引碎片级别会影响性能。

在 Interactive SQL 中，运行以下 SQL 语句来查看本教程中引入到碎片表中的索引碎片：

```
CALL sa_index_density( 'fragment' );
```

TableName	TableId	IndexName	IndexId	IndexType	LeafPages	Density
fragment	736	idx_fragment	1	NUI	1,177	0.597509

您的结果可能不同，但 [**密度**] 列的值应大约为 0.6。

在 Interactive SQL 中，运行以下 SQL 语句来提高索引的密度：

```
ALTER INDEX idx_fragment ON fragment REBUILD;
```

另请参见

- “sa_index_density 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》
- “REORGANIZE TABLE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “重建索引”一节第 70 页
- “减少索引碎片和分布偏差”一节第 216 页
- “应用程序分析”一节第 161 页

教程：诊断表碎片

利用本节中的教程来学习如何使用 [**应用程序分析向导**] 来确定您的数据库是否有不可接受的表碎片级别。

表数据存储在数据库页面中。针对表执行数据修改语言 (DML) 语句（如 INSERT、UPDATE 和 DELETE）时，各行可能不会连续存储，可能会在多个页之间拆分。即使 CPU 活动性很高，表碎片也可能对需要扫描表的查询的性能造成负面影响。

第 1 课：建立表碎片

本教程假定您已创建了测试数据库。如果尚未创建测试数据库，请参见“第 1 课：创建测试数据库”一节第 230 页。

提示
可以将本教程中的 SQL 语句复制并粘贴到 Interactive SQL 中。

♦ 建立表碎片

1. 启动 Sybase Central，并使用用户 ID **DBA** 和口令 **sql** 连接到测试数据库 *app_profiling.db*。

   如果不熟悉如何启动 Sybase Central 和连接到数据库，请参见“连接到本地数据库”一节《SQL Anywhere 服务器 - 数据库管理》。

2. 在左窗格中，选择 *app_profiling* - *DBA* 数据库，然后选择 [**文件**] » [**打开 Interactive SQL**]。

   Interactive SQL 启动并连接到 *app_profiling* - *DBA* 数据库。

3. 在 Interactive SQL 中，运行以下 SQL 语句以引入表碎片：

   a. 创建表：

```
CREATE TABLE "DBA"."tablefrag" (
"id" UNSIGNED BIGINT NOT NULL DEFAULT AUTOINCREMENT,
"val1" LONG VARCHAR NULL,
"val2" LONG VARCHAR NULL,
"val3" LONG VARCHAR NULL,
"val4" LONG VARCHAR NULL,
"val5" LONG VARCHAR NULL,
"val6" LONG VARCHAR NULL,
"val7" LONG VARCHAR NULL,
"val8" LONG VARCHAR NULL,
"val9" LONG VARCHAR NULL,
"val10" LONG VARCHAR NULL,
PRIMARY KEY ( id ) );
```

   b. 创建将值插入该表的过程：

```
CREATE PROCEDURE "DBA"."proc_tablefrag"( )
   BEGIN
      DECLARE I INTEGER;
      SET I = 0;
      WHILE I < 1000
         LOOP
            INSERT INTO "DBA"."tablefrag" ( "val1" )
```



```
            VALUES('a');
            SET I = I + 1;
        END LOOP;
    END;
```

c. 插入值：

```
CALL proc_tablefrag( );
```

d. 更新表中的值：

```
UPDATE "DBA"."tablefrag"
SET "val1" = 'abcdefghijklmnopqrstuvwxyz0123456789',
    "val2" = 'abcdefghijklmnopqrstuvwxyz0123456789',
    "val3" = 'abcdefghijklmnopqrstuvwxyz0123456789',
    "val4" = 'abcdefghijklmnopqrstuvwxyz0123456789',
    "val5" = 'abcdefghijklmnopqrstuvwxyz0123456789',
    "val6" = 'abcdefghijklmnopqrstuvwxyz0123456789',
    "val7" = 'abcdefghijklmnopqrstuvwxyz0123456789',
    "val8" = 'abcdefghijklmnopqrstuvwxyz0123456789',
    "val9" = 'abcdefghijklmnopqrstuvwxyz0123456789',
    "val10" = 'abcdefghijklmnopqrstuvwxyz0123456789';
```

e. 提交对数据库所做的更改：

```
COMMIT;
```

4. 退出 Interactive SQL。

第 2 课：识别表碎片

使用此过程来识别表碎片及学习如何查找表碎片警告。建议您定期检查产品数据库中的碎片警告。

注意
您在上一过程中执行的语句引入了表碎片。但是，在某些系统中产生的表碎片可能不足以产生此过程中描述的警告及建议。

♦ 识别表碎片

1. 在 Sybase Central 中，选择 [**模式**] » [**应用程序分析**]。

   如果未出现 [**应用程序分析向导**]，则选择 [**应用程序分析**] » [**打开应用程序分析向导**]。
2. 在 [**分析选项**] 页面中，选择 [**基于数据库模式的总体数据库性能**]。
3. 在 [**分析文件**] 页面中，将分析文件保存在相应目录下。例如，*C:\AppProfilingTutorial*。
4. 单击 [**完成**]。

   [**应用程序分析详细信息**] 窗格中出现建议的列表。
5. 要查看更多详细信息，双击 [**碎片表**]。将出现 [**建议**] 窗口，其中包含您可以运行的用来解析表碎片的 SQL 语句。

第 3 课：检查表碎片

要检查是否存在表碎片（例如，`CALL sa_table_fragmentation( 'tablefrag' );`），请运行 sa_table_fragmentation 系统过程。如果每行的分段数大于 1.1，则存在表碎片。等级较高的碎片可能会对性能造成负面影响。请参见“sa_table_fragmentation 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》。

您在本教程中创建的表应具有的碎片值约为 1.9。

在 Interactive SQL 中，运行以下 SQL 语句以减少表碎片：

```
REORGANIZE TABLE tablefrag;
```

请参见“REORGANIZE TABLE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

另请参见

- “减少表碎片”一节第 214 页
- “应用程序分析”一节第 161 页

教程：使用过程分析进行基线比较

使用本节中的教程来学习如何使用 [**应用程序分析向导**] 来创建基线，在改进性能时可以使用该基线进行比较。

过程分析可提供过程、用户定义的函数、事件、系统触发器和触发器的执行时间测量值。您可将保存的结果设置为基线，对过程进行增量更改，然后在每次更改后运行该过程。这使您可将新结果与基线进行比较。

第 1 课：创建基线过程

本教程假定您已创建了测试数据库。如果尚未创建测试数据库，请参见“第 1 课：创建测试数据库”一节第 230 页。

提示
可以将本教程中的 SQL 语句复制并粘贴到 Interactive SQL 中。

♦ 创建基线过程

1. 启动 Sybase Central，并使用用户 ID **DBA** 和口令 **sql** 连接到测试数据库 *app_profiling - DBA*。

   如果不熟悉如何启动 Sybase Central 和连接到数据库，请参见“连接到本地数据库”一节《SQL Anywhere 服务器 - 数据库管理》。

2. 在左窗格中，选择 *app_profiling - DBA* 数据库，然后选择 [**文件**] » [**打开 Interactive SQL**]。

   Interactive SQL 启动并连接到 *app_profiling - DBA* 数据库。

3. 在 Interactive SQL 中，运行以下 SQL 语句：

   a. 创建表：

   ```
   CREATE TABLE table1 (
   Count INT );
   ```

   b. 创建基线过程：

   ```
   CREATE PROCEDURE baseline( )
      BEGIN
         INSERT table1
            SELECT COUNT (*)
               FROM rowgenerator r1, rowgenerator r2,
                    rowgenerator r3
            WHERE r3.row_num < 5;
      END;
   ```

   c. 提交对数据库所做的更改：

   ```
   COMMIT;
   ```

4. 关闭 Interactive SQL。

第 2 课：对基线过程运行更新的过程

- **对基线过程运行更新的过程**

1. 在 Sybase Central 中，选择 [**模式**] » [**应用程序分析**]。

   如果未出现 [**应用程序分析向导**]，则选择 [**应用程序分析**] » [**打开应用程序分析向导**]。
2. 在 [**欢迎**] 页面上，单击 [**下一步**]。
3. 在 [**分析选项**] 页面上，选择 [**存储过程、函数、触发器或事件执行时间**]。
4. 单击 [**完成**]。

   数据库服务器开始过程分析。
5. 在 Sybase Central 的左窗格中，双击 [**过程和函数**]。
6. 右键单击该基线过程，然后选择 [**从 Interactive SQL 执行**]。过程分析即被启用，这样，就可捕获到过程的执行详细信息。
7. 关闭 Interactive SQL。
8. 查看分析结果。
   a. 在 Sybase Central 的左窗格中，选择基线过程。
   b. 单击右窗格中的 [**分析结果**] 选项卡。如果未出现任何结果，则选择 [**视图**] » [**刷新文件夹**]。

      将出现基线过程中各行的执行时间。
9. 保存分析结果。
   a. 右键单击数据库并选择 [**属性**]。
   b. 单击 [**分析设置**] 选项卡。
   c. 选择 [**将数据库中的当前分析信息保存到以下分析日志文件**]，然后为分析日志文件指定一个位置和文件名。
   d. 单击 [**应用**]。不要关闭属性窗口。

      刚刚收集的过程分析信息保存到指定的分析日志文件 (*.plg*) 中。
10. 启用相对分析日志文件的基线比较
    a. 在 [**App_Profiling - DBA 数据库属性**] 窗口的 [**分析设置**] 选项卡中，选择 [**将以下分析日志文件中的分析信息用作比较的基线**]。
    b. 浏览至您创建的分析日志文件，并选中它。
    c. 单击 [**应用**]。
    d. 单击 [**确定**] 以关闭 [**App_Profiling - DBA 数据库属性**] 窗口。
11. 对基线过程进行更改。
    a. 在 Sybase Central 中，选择 [**模式**] » [**设计**]。
    b. 在左窗格中，浏览至 [**过程和函数**] 中的基线过程，并选中它。
    c. 在右窗格的 [**SQL**] 选项卡上，删除现有的 INSERT 语句。

d. 将以下 SQL 语句复制和粘贴到该过程中：

```
INSERT table1
   SELECT COUNT ( * ) FROM rowgenerator r1, rowgenerator r2,
rowgenerator r3
   WHERE r3.row_num < 250;
```

e. 选择 [**文件**] » [**保存**]。

12. 在 [**过程和函数**] 中，右键单击该基线过程，然后选择 [**从 Interactive SQL 执行**]。
13. 该过程完成后，退出 Interactive SQL。

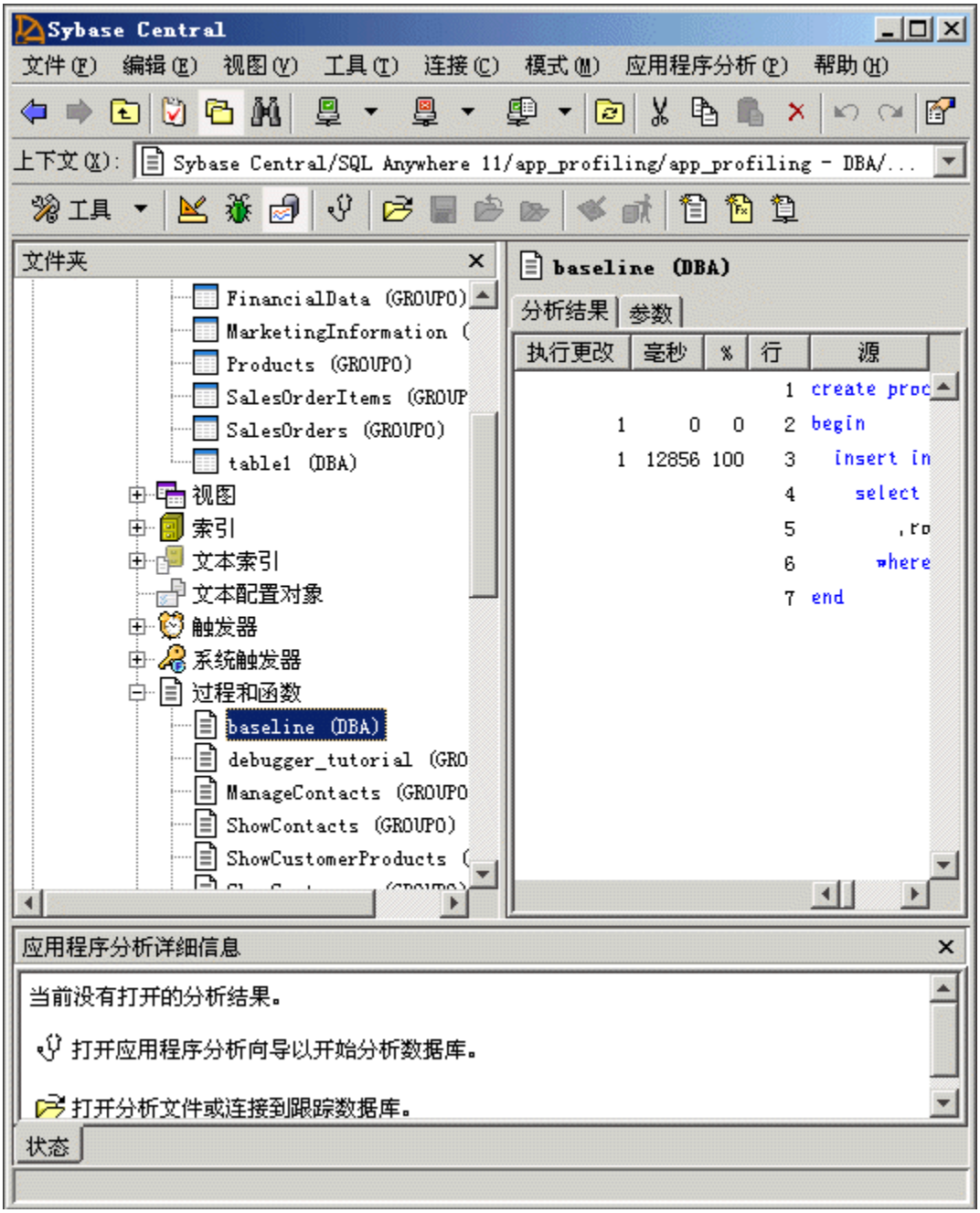
第 3 课：比较过程分析结果

♦ **比较过程分析结果**

1. 在 Sybase Central 中，选择 [**模式**] » [**应用程序分析**]。

   如果出现 [**应用程序分析向导**]，则单击 [**取消**]。
2. 在 Sybase Central 左窗格的 [**过程和函数**] 中，单击该基线过程。
3. 在右窗格中，单击 [**分析结果**] 选项卡。
4. 选择 [**视图**] » [**刷新文件夹**]。

   出现两个新列，[**Execs.+/-**] 和 [**ms. +/-**]。



[**Execs.+/-**] 和 [**ms. +/-**] 列是将分析日志文件中的统计信息与过程的最近一次执行期间捕获的统计信息进行比较的结果。具体来说，它们分别将过程中每一行代码的执行次数和执行持续时间进行了比较。

通常，您会关注 [**ms. +/-**] 列，该列指明您是否改进了过程中代码行的执行时间。减号和红色字体表示时间加快。没有符号和绿色字体则表示时间减慢。在本教程中，[**ms. +/-**] 列中的值应该是一个加

号连同一个以绿色字体显示的执行时间。INSERT 语句在更新后过程中的执行速度比其在基线过程中的执行速度慢。

另请参见

- “分析过程分析的结果”一节第 165 页
- “[应用程序分析] 模式中的过程分析”一节第 162 页

查询和修改数据

本节介绍了如何查询和修改数据，包括如何使用连接。它包括几个有关查询的章节（从简单到复杂），以及有关插入、删除和更新数据的信息。本节还深入探讨如何创建返回多维结果的分析查询。

查询数据

目录

查询请求数据库中的数据并接收结果。此过程也称为数据检索。所有 SQL 查询都是使用 SELECT 语句表达的。可以使用 SELECT 语句在一个或多个表中检索所有行或行的子集，以及在一个或多个表中检索所有列或列的子集。

查询和 SELECT 语句

SELECT 语句从数据库中检索信息以供客户端应用程序使用。SELECT 语句也称为**查询**。信息以结果集的形式传送到客户端应用程序。然后，客户端可以对结果集进行处理。例如，Interactive SQL 在 [结果] 窗格中显示结果集。结果集由一组行组成，就像数据库中的表一样。

SELECT 语句包含**子句**，这些子句是定义要返回的结果范围的命令。在下面的 SELECT 语法中，每个新行都是一个单独的子句。这里只列出比较常见的子句。

SELECT *select-list*
[**FROM** *table-expression*]
[**WHERE** *search-condition*]
[**GROUP BY** *column-name*]
[**HAVING** *search-condition*]
[**ORDER BY** { *expression* | *integer* }]

SELECT 语句中的子句如下：

- SELECT 子句指定您要检索的列。它是 SELECT 语句中唯一必需的子句。
- FROM 子句指定从中提取列的表。在所有从表中检索数据的查询中都需要该子句。没有 FROM 子句的 SELECT 语句具有不同的含义，本章不讨论它们。

  虽然大多数查询都是对表执行操作，但是查询也可以从具有列和行的其它对象（包括视图、其它查询（派生表）和存储过程结果集）中检索数据。请参见“FROM 子句”一节 《SQL Anywhere 服务器 - SQL 参考》。
- WHERE 子句指定表中您要查看的行。
- GROUP BY 子句用于集合数据。
- HAVING 子句指定要在其上收集集合数据的行。
- ORDER BY 子句对结果集中的行进行排序。（缺省情况下，以没有任何意义的顺序从关系数据库中返回行。）

  有关 GROUP BY、HAVING 和 ORDER BY 子句的信息，请参见“对查询结果进行汇总、分组和排序”第 339 页。

大多数子句是可选的，但是如果包括这些子句，它们就必须以正确的顺序出现。

请参见“SELECT 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

SQL 查询

本文档中，显示 SELECT 语句和其它 SQL 语句时，每个子句都在单独一行上，SQL 关键字以大写字母显示。这样做是便于阅读语句，但并不是必需的。可以使用大写或小写字母输入 SQL 关键字，也可以在语句中的任何位置处使用换行符。

关键字和换行符

例如，以下 SELECT 语句从 Contacts 表中查找居住在 California 的联系人的名和姓。

```
SELECT GivenName, Surname
FROM Contacts
WHERE State = 'CA';
```

如果按以下方式输入该语句，虽然不便于阅读，但同样有效：

```
SELECT GivenName,
Surname from Contacts
WHERE State
 = 'CA';
```

字符串和标识符的区分大小写

在 SQL Anywhere 数据库中，标识符（如表名、列名等）是不区分大小写的。

缺省情况下，字符串是不区分大小写的，因此 'CA'、'ca'、'cA' 和 'Ca' 都是等同的，但是如果将数据库创建为区分大小写的数据库，则字符串的大小写就有意义了。SQL Anywhere 示例数据库是不区分大小写的。

另请参见

- “创建数据库”一节 《SQL Anywhere 服务器 - 数据库管理》
- “初始化实用程序 (dbinit)”一节 《SQL Anywhere 服务器 - 数据库管理》
- “区分大小写”一节第 622 页

限定标识符

如果引用的对象存在不明确性，则可以限定数据库标识符的名称。例如，SQL Anywhere 示例数据库中多个表都具有名为 City 的列，因此您可能必须用表的名称限定对 City 的引用。在一个较大的数据库中，可能还需要使用表的所有者的名称来标识该表。

```
SELECT Contacts.City
FROM Contacts
WHERE State = 'CA';
```

由于本节中的示例涉及单表查询，因此通常不用表的名称或表所有者的名称对语法模型和示例中的列名进行限定。

省去这些元素是为了便于阅读；包括这些限定符也没有任何错误。

结果集中的行顺序

结果集中的行顺序没有意义。无法保证行以什么顺序从数据库返回，该顺序也没有意义。如果您要以特定的顺序检索行，则必须在查询中指定顺序。

选择列表：指定列

选择列表由从中查询数据的一个或多个对象组成。选择列表通常由一系列使用逗号分隔的列名组成，或将星号用作速记代表所有列。一般地说，选择列表可以包括一个或多个由逗号分隔的表达式。列表中最后一列之后没有逗号，如果列表中只有一列，也不使用逗号。

选择列表的一般语法如下：

SELECT *expression* [, *expression*]...

如果列表中的任何表名或列名不符合有效标识符的规则，则您必须将该标识符括在双引号中。

选择列表表达式可以包括 *（所有列）、一组列名、字符串、列标题和包括算术运算符的表达式。还可以包括集合函数，相关内容在“对查询结果进行汇总、分组和排序”第 339 页中进行了介绍。

有关表达式的详细信息，请参见“表达式”一节 《SQL Anywhere 服务器 - SQL 参考》。

从表中选择所有列

在 SELECT 语句中，星号 (*) 具有特殊含义。它表示 FROM 子句指定的所有表中的所有列名。如果您要查看某个表中的所有列，则可以使用星号节省输入时间并避免键入错误。

当使用 SELECT * 时，列返回的顺序是创建表时定义这些列的顺序。

选择表中所有列的语法是：

```
SELECT *
FROM table-expression;
```

SELECT * 查找当前在表中的所有列，因此表结构中的更改（例如添加、删除或重命名列）将自动修改 SELECT * 的结果。逐个列出列可以使您更精确地控制结果。

示例

以下语句检索 Departments 表中的所有列。未包括 WHERE 子句；因此，此语句检索表中的每一行：

```
SELECT *
FROM Departments;
```

结果如下：

DepartmentID	DepartmentName	DepartmentHeadID
100	R & D	501
200	Sales	902
300	Finance	1293
400	Marketing	1576

DepartmentID	DepartmentName	DepartmentHeadID
...	...	...

在 SELECT 关键字后面按顺序列出表中所有列名将得到完全相同的结果：

```
SELECT DepartmentID, DepartmentName, DepartmentHeadID
FROM Departments;
```

像列名一样，可以使用表名对 "*" 进行限定，如在下面的查询中：

```
SELECT Departments.*
FROM Departments;
```

从表中选择特定的列

可以通过紧接在 SELECT 关键字之后列出列来限制 SELECT 语句检索的列。此 SELECT 语句的语法如下：

SELECT *column-name* [, *column-name*]...
FROM *table-name*

在语法中，*column-name* 和 *table-name* 应该用要查询的列名和表名替换。

例如：

```
SELECT Surname, GivenName
FROM Employees;
```

投影和限制

投影是表中各列的子集。**限制**（也称作**选择**）是表中各行的子集（基于某些条件）。

例如，以下 SELECT 语句在 SQL Anywhere 示例数据库中检索价格超过 $15 的所有产品的名称和价格：

```
SELECT Name, UnitPrice
FROM Products
WHERE UnitPrice > 15;
```

此查询同时使用投影 (`SELECT Name, UnitPrice`) 和限制 (`WHERE UnitPrice > 15`)。

重新排列列的顺序

列出列名的顺序决定列显示的顺序。以下两个示例显示如何指定显示的列顺序。两个示例都查找并显示 Departments 表所有五行中的部门名称和标识号，但是显示顺序不同。

```
SELECT DepartmentID, DepartmentName
FROM Departments;
```

DepartmentID	DepartmentName
100	R & D

DepartmentID	DepartmentName
200	Sales
300	Finance
400	Marketing
...	...

```
SELECT DepartmentName, DepartmentID
FROM Departments;
```

DepartmentName	DepartmentID
R & D	100
Sales	200
Finance	300
Marketing	400
...	...

连接

连接通过比较每个表中各列的值，将两个或多个表中的各行链接起来。例如，您可能想要为发运的商品件数超过一打的所有订单项选择订单项标识号和产品名称：

```
SELECT SalesOrderItems.ID, Products.Name
FROM Products JOIN SalesOrderItems
WHERE SalesOrderItems.Quantity > 12;
```

Products 表和 SalesOrderItems 表基于两个表间的外键关系连接在一起。

请参见“连接：从多个表检索数据”第 361 页。

重命名查询结果中的列

查询结果由一组列组成。缺省情况下，每个列的标题都是选择列表中提供的表达式。

当显示查询结果时，每个列的缺省标题都是创建该列时赋予它的名称。可以指定不同的列标题（即**别名**），如下所示：

SELECT *column-name* [**AS**] *alias*

提供别名可以产生更加易读的结果。例如，您可以在部门列表中将 DepartmentName 更改为 Department，如下所示：


```
SELECT DepartmentName AS Department,
   DepartmentID AS "Identifying Number"
FROM Departments;
```

Department	Identifying Number
R & D	100
Sales	200
Finance	300
Marketing	400
...	...

在别名中使用空格和关键字

DepartmentID 的别名 Identifying Number 括在双引号中，因为它是一个标识符。如果要在别名中使用关键字，您也可以使用双引号。例如，如果没有引号，则以下查询无效：

```
SELECT DepartmentName AS Department,
   DepartmentID AS "integer"
FROM Departments;
```

如果您想要确保与 Adaptive Server Enterprise 兼容，则应使用 30 个字节（或更少）的带引号的别名。

查询结果中的字符串

大多数 SELECT 语句产生的结果只包含 FROM 子句指定的表中的数据。但是，通过将字符串括在单引号中并用逗号将它们与选择列表中的其它元素分开，也可以在查询结果中显示这些字符串。字符串中若要包括引号，您需要在它前面再放一个引号。例如：

```
SELECT 'The department''s name is' AS "Prefix",
   DepartmentName AS Department
FROM Departments;
```

Prefix	Department
The department's name is	R & D
The department's name is	Sales
The department's name is	Finance
The department's name is	Marketing
The department's name is	Shipping

计算 SELECT 列表中的值

选择列表中的表达式可以比单纯的列名或字符串复杂。例如，您可以使用选择列表中数字列中的数据执行计算。

算术运算

为了说明您可以在选择列表中执行的数字运算，我们从 SQL Anywhere 示例数据库中产品的名称、库存数量和单价列表开始。

```
SELECT Name, Quantity, UnitPrice
FROM Products;
```

Name	Quantity	UnitPrice
Tee Shirt	28	9
Tee Shirt	54	14
Tee Shirt	75	14
Baseball Cap	112	9
...	...	...

假设惯例是在某种产品的库存为十件时补充产品库存。以下查询列出在再次订购前每种产品必须卖出的数量：

```
SELECT Name, Quantity - 10
   AS "Sell before reorder"
FROM Products;
```

Name	Sell before reorder
Tee Shirt	18
Tee Shirt	44
Tee Shirt	65
Baseball Cap	102
...	...

您还可以结合多个列中的值。以下查询列出库存中每种产品的总价值：

```
SELECT Name, Quantity * UnitPrice AS "Inventory value"
FROM Products;
```

Name	Inventory value
Tee Shirt	252.00
Tee Shirt	756.00
Tee Shirt	1050.00
Baseball Cap	1008.00
...	...

算术运算符优先级

当表达式中有多个算术运算符时，先计算乘法、除法和模，然后计算减法和加法。当表达式中的所有算术运算符优先级级别相同时，执行顺序为从左到右。括号中的表达式的优先级高于所有其它运算符。

例如，以下 SELECT 语句计算库存中每种产品的总价值，然后从该值中减去 5 美元。

```
SELECT Name, Quantity * UnitPrice - 5
FROM Products;
```

要确保结果正确，请尽量使用括号。以下查询与上一个查询具有相同的含义且给出的结果也相同，但语法更加精确：

```
SELECT Name, ( Quantity * UnitPrice ) - 5
FROM Products;
```

另请参见“运算符优先级”一节 《SQL Anywhere 服务器 - SQL 参考》。

字符串运算

您可以使用字符串连接运算符来连接字符串。您可以使用 ||（与 SQL/2003 兼容）或 +（受 Adaptive Server Enterprise 支持）作为连接运算符。例如，以下语句检索 GivenName 和 Surname 的值，并在结果中将二者连接在一起：

```
SELECT EmployeeID, GivenName || ' ' || Surname AS Name
FROM Employees;
```

EmployeeID	Name
102	Fran Whitney
105	Matthew Cobb
129	Philip Chin
148	Julie Jordan
...	...

日期和时间运算

虽然可以在日期和时间列上使用运算符，但此操作通常涉及函数的使用。请参见“SQL 函数”《SQL Anywhere 服务器 - SQL 参考》。

有关计算列的其它注意事项

- **可以赋予列一个别名** 缺省情况下，列名是选择列表中列出的表达式，但是对于计算列，使用表达式作为名称十分麻烦，并且不能提供足够的信息。
- **可以使用其它运算符** 可使用乘法运算符将多个列组合在一起。您可以使用其它运算符，包括标准的算术运算符以及逻辑运算符和字符串运算符。

  例如，以下查询列出所有客户的全名：

  ```
  SELECT ID, (GivenName || ' ' || Surname ) AS "Full name"
  FROM Customers;
  ```

  || 运算符连接字符串。在本查询中，列的别名具有空格，因此必须用双引号括起来。此规则不仅适用于列别名，还适用于数据库中的表名和其它标识符。请参见“运算符”一节 《SQL Anywhere 服务器 - SQL 参考》。
- **可以使用函数** 除了组合列之外，您可以使用多种内置函数生成所需的结果。

  例如，以下查询以大写字母列出产品名称：

  ```
  SELECT ID, UCASE( Name )
  FROM Products;
  ```

ID	UCASE(Products.name)
300	TEE SHIRT
301	TEE SHIRT
302	TEE SHIRT
400	BASEBALL CAP
...	...

请参见“SQL 函数”《SQL Anywhere 服务器 - SQL 参考》。

消除重复查询结果

可选的 DISTINCT 关键字可以消除 SELECT 语句结果中重复的行。如果不指定 DISTINCT，您将得到所有行，其中包括重复行。或者，您可以在选择列表前指定 ALL 以得到所有行。为了和 SQL 的其它实现兼容，SQL Anywhere 语法允许使用 ALL 以显式请求所有行。ALL 是缺省设置。

例如，如果您在不使用 DISTINCT 的情况下搜索 Contacts 表中的所有城市，您会得到 60 行：

```
SELECT City
FROM Contacts;
```


您可以使用 DISTINCT 消除重复条目。以下查询只返回 16 行：

```
SELECT DISTINCT City
FROM Contacts;
```

NULL 值是重复的

DISTINCT 关键字将 NULL 值视为是相互重复的。换句话说，当 SELECT 语句中包括 DISTINCT 时，不管遇到多少个 NULL 值，结果中只返回一个 NULL。请参见“排除不必要的 DISTINCT 条件”一节第 514 页。

FROM 子句：指定表

每个涉及表、视图或存储过程中数据的 SELECT 语句都需要有 FROM 子句。

FROM 子句可以包括链接两个或多个表的 JOIN 条件，也可以包括与其它查询（派生表）的连接。有关这些功能的信息，请参见“连接：从多个表检索数据”第 361 页。

限定表名

在 FROM 子句中，始终允许使用表和视图的全称语法，例如：

```
SELECT select-list
FROM owner.table-name;
```

只有当对象所有者的用户 ID 与当前连接的用户 ID 不同，或者所有者的用户 ID 不是当前连接的用户 ID 所属组的名称时，才有必要限定表、视图和过程名称。

使用相关名

将相关名赋予某个表名，可提高可读性，并且在每次引用该表名时不必输入完整的表名。您可以在 FROM 子句中指派相关名，方法是在表名之后输入该相关名，如下所示：

```
SELECT d.DepartmentID, d.DepartmentName
FROM Departments d;
```

使用相关名时，对该表的其它所有引用（例如，在 WHERE 子句中）都*必须*使用相关名而不是表名。相关名必须符合有效标识符的规则。

请参见“FROM 子句”一节 《SQL Anywhere 服务器 - SQL 参考》。

查询派生表

派生表是通过计算查询表达式从一个或多个表直接或间接派生的表。派生表在 SELECT 语句的 FROM 子句中定义。

查询派生表的作用方式与查询视图相同。即，派生表的值在评估派生表定义时决定。但是，派生表与视图不同，因为派生表的定义不存储在数据库中。派生表与基表和临时表不同，因为它们不被实例化，并且不能在对其进行定义的查询之外引用它们。

以下查询使用派生表 (my_drv_tbl) 保存每个部门的最高薪水。然后，将派生表中的数据与 Employees 表相连接，以得到赚取这些薪水的雇员的姓。

```
SELECT Surname,
   my_drv_tbl.max_sal AS Salary,
   my_drv_tbl.DepartmentID
FROM Employees e,
  ( SELECT MAX( Salary ) AS max_sal, DepartmentID
      FROM Employees
      GROUP BY DepartmentID ) my_drv_tbl
  WHERE e.Salary = my_drv_tbl.max_sal
  AND e.DepartmentID = my_drv_tbl.DepartmentID
ORDER BY Salary DESC;
```


Surname	Salary	DepartmentID
Shea	138948.00	300
Scott	96300.00	100
Kelly	87500.00	200
Evans	68940.00	400
Martinez	55500.80	500

以下示例创建一个对 Products 表中的项目进行排序的派生表 (MyDerivedTable)，然后查询该派生表以返回三个最低开销项目：

```
SELECT TOP 3 *
       FROM ( SELECT Description,
                     Quantity,
                     UnitPrice,
                     RANK() OVER ( ORDER BY UnitPrice ASC )
                  AS Rank
                  FROM Products ) AS MyDerivedTable
ORDER BY Rank;
```

另请参见： “FROM 子句”一节 《SQL Anywhere 服务器 - SQL 参考》.

查询除表之外的对象

FROM 子句中最常用的元素是表名。但是，也可以从结构与表类似的其它数据库对象—即，一组明确定义的行和列中查询行。例如，可以查询视图，还可以查询返回结果集的存储过程。

例如，以下语句查询名为 ShowCustomerProducts 的存储过程的结果集。

```
SELECT *
FROM ShowCustomerProducts( 149 );
```

WHERE 子句：指定行

SELECT 语句中的 WHERE 子句指定具体检索哪些行的搜索条件。搜索条件也称为**谓语**。一般格式为：

SELECT *select-list*
FROM *table-list*
WHERE *search-condition*

WHERE 子句中的搜索条件包括以下内容：

- **比较运算符** （=、<、> 等）例如，您可以列出收入超过 $50,000 的所有雇员：

```
SELECT Surname
   FROM Employees
   WHERE Salary > 50000;
```

- **范围** （BETWEEN 和 NOT BETWEEN）例如，您可以列出收入在 $40,000 到 $60,000 的所有雇员：

```
SELECT Surname
   FROM Employees
   WHERE Salary BETWEEN 40000 AND 60000;
```

- **列表** （IN、NOT IN）例如，您可以列出居住在 Ontario、Quebec 或 Manitoba 的所有客户：

```
SELECT CompanyName, State
   FROM Customers
   WHERE State IN( 'ON', 'PQ', 'MB');
```

- **字符匹配** （LIKE 和 NOT LIKE）例如，您可以列出电话号码以 415 开头的所有客户。（电话号码在数据库中作为字符串存储）：

```
SELECT CompanyName, Phone
   FROM Customers
   WHERE Phone LIKE '415%';
```

- **未知值** （IS NULL 和 IS NOT NULL）例如，您可以列出有经理的所有部门：

```
SELECT DepartmentName
   FROM Departments
   WHERE DepartmentHeadID IS NOT NULL;
```

- **组合** （AND、OR）例如，您可以列出收入超过 $50,000 并且名字以字母 A 开头的所有雇员。

```
SELECT GivenName, Surname
   FROM Employees
   WHERE Salary > 50000
   AND GivenName like 'A%';
```

有关搜索条件的完整语法，请参见“搜索条件”一节 《SQL Anywhere 服务器 - SQL 参考》。

在 WHERE 子句中使用比较运算符

可以在 WHERE 子句中使用比较运算符。这些运算符遵循以下语法：

WHERE *expression comparison-operator expression*

请参见“比较运算符”一节 《SQL Anywhere 服务器 - SQL 参考》和“表达式”一节 《SQL Anywhere 服务器 - SQL 参考》。

有关比较的说明

- **排序顺序** 比较字符数据时，< 表示在排序顺序中靠前，而 > 则表示在排序顺序中靠后。排序顺序是由创建数据库时选择的归类决定的。您可以通过对数据库运行 dbinfo 实用程序查找归类：

  ```
  dbinfo -c "uid=DBA;pwd=sql"
  ```

  也可以通过转到 [**数据库属性**] 窗口的 [扩展信息] 选项卡从 Sybase Central 查找归类。

- **尾随空白** 当创建数据库时，您可以指示是否忽略尾随空白或不忽略尾随空白以进行比较。

  缺省情况下，创建数据库时不忽略尾随空白。例如，'Dirk' 和 'Dirk ' 不同。您可以创建具有空格填充的数据库，以便忽略尾随空白。

- **比较日期** 比较日期时，< 表示日期较早，> 表示日期较晚。

- **区分大小写** 当创建数据库时，您可以指示字符串比较是否区分大小写。

  缺省情况下，创建数据库时不区分大小写。例如，'Dirk' 和 'DIRK' 相同。可以创建区分大小写的数据库。

这里是一些使用比较运算符的 SELECT 语句：

```
SELECT *
   FROM Products
   WHERE Quantity < 20;
SELECT E.Surname, E.GivenName
   FROM Employees E
   WHERE Surname > 'McBadden';
SELECT ID, Phone
   FROM Contacts
   WHERE State  != 'CA';
```

NOT 运算符

NOT 运算符对表达式取非。以下两个查询中的任何一个都会查找所有价格为 $10（或不到）的 T 恤衫和棒球帽。但是，请注意否定逻辑运算符 (NOT) 和否定比较运算符 (!>) 之间的差别。

```
SELECT ID, Name, Quantity
   FROM Products
   WHERE (name = 'Tee Shirt' OR name = 'BaseBall Cap')
   AND NOT UnitPrice > 10;
SELECT ID, Name, Quantity
   FROM Products
   WHERE (name = 'Tee Shirt' OR name = 'BaseBall Cap')
   AND UnitPrice !> 10;
```

在 WHERE 子句中使用范围

BETWEEN 关键字指定包含的范围，将搜索该范围的上限值、下限值以及介于这两个值之间的值。

♦ **列出价格在 $10 到 $15（含）之间的所有产品：**

● 输入以下查询：

```
SELECT Name, UnitPrice
   FROM Products
   WHERE UnitPrice BETWEEN 10 AND 15;
```

Name	UnitPrice
Tee Shirt	14
Tee Shirt	14
Baseball Cap	10
Shorts	15

您可以使用 NOT BETWEEN 查找不在该范围中的所有行。

♦ **列出价格低于 $10 或高于 $15 的所有产品：**

● 执行以下查询：

```
SELECT Name, UnitPrice
   FROM Products
   WHERE UnitPrice NOT BETWEEN 10 AND 15;
```

Name	UnitPrice
Tee Shirt	9
Baseball Cap	9
Visor	7
Visor	7
...	...

在 WHERE 子句中使用列表

IN 关键字可用于选择与一组值中任何一个值匹配的值。该表达式可以是一个常量或一个列名，而列表可以是一组常量或一个子查询（较常见）。

例如，在不使用 IN 的情况下，如果需要一个所有居住在 Ontario、Manitoba 或 Quebec 的客户姓名和州的列表，可输入以下查询：

```
SELECT CompanyName, State
   FROM Customers
   WHERE State = 'ON' OR State = 'MB' OR State = 'PQ';
```


但是，如果使用 IN，将得到相同的结果。IN 关键字后面的各项必须使用逗号分隔并括在括号中。在字符、日期或时间值的两边使用单引号。例如：

```
SELECT CompanyName, State
   FROM Customers
   WHERE State IN( 'ON', 'MB', 'PQ');
```

或许 IN 关键字最重要的用法是用在嵌套查询（也称作子查询）中。

WHERE 子句中的匹配字符串

模式匹配是一种识别字符数据的通用方式。在 SQL 中，LIKE 关键字用于搜索模式。模式匹配使用通配符来匹配不同的字符组合。

LIKE 关键字指示其后的字符串是一个匹配模式。LIKE 与字符数据配合使用。

LIKE 的语法为：

expression [**NOT**] **LIKE** *match-expression*

将要进行匹配的表达式与匹配表达式进行比较，匹配表达式可以包含以下特殊符号：

符号	含义
%	匹配具有任意数目字符（没有字符或多个字符）的任意字符串
_	匹配任意一个字符
[指定符]	方括号中的指定符可以采用以下形式： ● **范围** 范围以格式 *rangespec1-rangespec2* 表示，其中 *rangespec1* 表示字符范围的开始，连字符表示范围，而 *rangespec2* 则表示字符范围的结束 ● **集合** 集合可包括任何一组离散的值并以任何顺序组成。例如，[a2bR]。 请注意范围 [a-f] 以及集合 [abcdef] 和 [fcbdae] 都返回相同一组值。
[^指定符]	指定符前面的脱字符 (^) 表示不包含。[^a-f] 表示不在范围 a-f 中；[^a2bR] 表示不是 a、2、b 和 R。

您可以将列数据与常量、变量或包含该表中所显示的通配符的其它列匹配。如果使用常量，您应该将匹配子串和字符串括在单引号中。

示例

以下所有示例都将 LIKE 与 Contacts 表中的 Surname 列配合使用。查询格式如下：

```
SELECT Surname
   FROM Contacts
   WHERE Surname LIKE match-expression;
```

第一个示例输入以下内容

```
SELECT Surname
   FROM Contacts
   WHERE Surname LIKE 'Mc%';
```

匹配表达式	说明	返回值
'Mc%'	搜索所有以字母 Mc 开头的姓名	McEvoy
'%er'	搜索所有以字母 er 结尾的姓名	Brier, Miller, Weaver, Rayner
'%en%'	搜索所有包含字母 en 的姓名。	Pettengill, Lencki, Cohen
'_ish'	搜索所有以字母 ish 结尾的四个字母的姓名。	Fish
'Br[iy][ae]r'	搜索 Brier、Bryer、Briar 或 Bryar。	Brier
'[M-Z]owell'	搜索所有以 owell 结尾、以从 M 到 Z 范围内的某个字母开头的姓名。	Powell
'M[^c]%'	搜索所有以 M 开头、第二个字母不是 c 的名称	Moore, Mulley, Miller, Masalsky

通配符需要使用 LIKE

使用通配符时如果没有 LIKE，则会将该通配符视为**字符串文字**，而不是模式：它们只代表它们自己的值。以下查询试图查找只由四个字符 415% 组成的所有电话号码。不查找以 415 开头的电话号码。

```
SELECT Phone
   FROM Contacts
   WHERE Phone = '415%';
```

另请参见“字符串文字”一节 《SQL Anywhere 服务器 - SQL 参考》 。

将 LIKE 用于日期和时间值

可以将 LIKE 用于日期和时间字段以及字符数据。在您将 LIKE 与日期和时间值一起使用时，日期将被转换为标准的 DATETIME 格式，然后转换为 VARCHAR。

这便是搜索 DATETIME 值时使用 LIKE 的一个功能。由于日期和时间条目可能包含多种日期格式，所以必须小心编写等同性测试才能成功。

例如，如果将值 9:20 和当前日期插入到一个名为 arrival_time 的列中，则以下子句无法找到该值，原因是该条目中既有日期又有时间：

```
WHERE arrival_time = '9:20'
```

但是，下面的子句将会找到值 9:20：

```
WHERE arrival_time LIKE '%09:20%'
```


使用 NOT LIKE

对于 NOT LIKE，您可以使用的通配符与可用于 LIKE 的通配符相同。若要查找 Contacts 表中地区号不是 415 的所有电话号码，您可以使用以下这两个查询中的任何一个：

```
SELECT Phone
   FROM Contacts
   WHERE Phone NOT LIKE '415%';

SELECT Phone
   FROM Contacts
   WHERE NOT Phone LIKE '415%';
```

使用下划线

另一个可以与 LIKE 一起使用的特殊字符是 _（下划线）字符，下划线只与一个字符匹配。例如，模式 'BR_U%' 与所有以 BR 开头并以 U 作为第四个字母的名称匹配。在 Braun 中，_ 字符与字母 A 匹配，% 与 N 匹配。请参见“LIKE 搜索条件”一节 《SQL Anywhere 服务器 - SQL 参考》。

字符串和引号

当输入或搜索字符和日期数据时，您必须将其括在单引号中，如以下示例所示。

```
SELECT GivenName, Surname
   FROM Contacts
   WHERE GivenName = 'John';
```

如果 quoted_identifier 数据库选项设置为 Off（缺省设置为 On），则您还可以使用双引号将字符或日期数据括起来。

♦ **为当前用户 ID 将 quoted_identifier 选项设置为 OFF：**

● 输入以下命令：

```
SET OPTION quoted_identifier = 'Off';
```

提供 quoted_identifier 选项是为了与 Adaptive Server Enterprise 兼容。缺省情况下，Adaptive Server Enterprise 选项 quoted_identifier 设为 Off，而 SQL Anywhere 选项 quoted_identifier 设为 On。请参见“quoted_identifier 选项 [兼容性]”一节 《SQL Anywhere 服务器 - 数据库管理》。

字符串中的引号

在字符条目中指定文字引号有两种方法。第一种方法是使用两个连续的引号。例如，如果您以一个单引号开始字符条目并想包括一个单引号作为条目的一部分，则请使用两个单引号：

```
'I don''t understand.'
```

对于双引号（quoted_identifier 设置为 Off），指定：

```
"He said, ""It is not really confusing."""
```

第二种方法只有在 quoted_identifier 设置为 Off 时才适用，该方法是将一种引号括在另一种引号中。换句话说，将包含双引号的条目括在单引号中，反之亦然。以下是一些示例：

```
'George said, "There must be a better way."'
"Isn't there a better way?"
'George asked, "Isn''t there a better way?"'
```

未知值：NULL

列中的 NULL 意味着用户或应用程序未在该列中进行输入。即，该列的数据值未知或不可用。

NULL 不同于零（数字值）或空格（字符值）。相反，NULL 值允许您将数字列有意的零输入或字符列有意的空白输入和未输入区别开来，未输入对数字列和字符列来说都是 NULL。

输入 NULL

只能在允许 NULL 值的列中输入 NULL。在创建表时决定该列是否可以接受 NULL 值。假定该列可以接受 NULL 值，则插入 NULL：

- **缺省值** 未输入任何数据并且列没有任何其它缺省设置。
- **显式输入** 可以显式插入不带引号的单词 NULL。如果在字符列中键入的单词 NULL 带有引号，则将其视为数据，而不是 NULL 值。

例如，Departments 表的 DepartmentHeadID 列允许 NULL 值。您可以为没有经理的部门输入两行，如下所示：

```
INSERT INTO Departments (DepartmentID, DepartmentName)
   VALUES (201, 'Eastern Sales')
INSERT INTO Departments
   VALUES (202, 'Western Sales', null);
```

返回 NULL 值

NULL 值被返回到客户端应用程序以进行显示，就像其它值一样。例如，以下示例说明 NULL 值如何在 Interactive SQL 中显示：

```
SELECT *
FROM Departments;
```

DepartmentID	DepartmentName	DepartmentHeadID
100	R & D	501
200	Sales	904
300	Finance	1293
400	Marketing	1576
500	Shipping	703
201	Eastern Sales	(null)
202	Western Sales	(null)

基于 NULL 对列进行测试

您可以使用 IS NULL 搜索条件将列值与 NULL 进行比较，并根据比较结果选择列值或执行特定的操作。只有返回值 TRUE 的列才被选中或引发指定的操作；那些返回 FALSE 或 UNKNOWN 的列不会被选中，也不引发指定的操作。

以下示例仅选择 UnitPrice 小于 $15 或者为 NULL 的行：

```
SELECT Quantity, UnitPrice
   FROM Products
   WHERE UnitPrice < 15
   OR UnitPrice IS NULL;
```

任何值与 NULL 的比较结果都是 UNKNOWN，这是因为不能确定 NULL 是等于（或不等于）给定的值还是等于（或不等于）另一个 NULL。

有些条件从不返回 TRUE，因此使用这些条件的查询不会返回结果集。例如，永远无法确定以下比较为 TRUE，这是因为 NULL 表示具有未知值：

```
WHERE column1 > NULL
```

此逻辑还适用于在 WHERE 子句中使用两个列名（即连接两个表）的情况。包含条件 `WHERE column1 = column2` 的子句不返回列中包含 NULL 的行。

您还可以使用这些模式查找 NULL 或非 NULL：

```
WHERE column_name IS NULL
```

```
WHERE column_name IS NOT NULL
```

例如：

```
WHERE advance < $5000
OR advance IS NULL
```

请参见“NULL 值”一节 《SQL Anywhere 服务器 - SQL 参考》。

NULL 的属性

下面详述 NULL 值的属性。

- **FALSE 和 UNKNOWN 之间的差异** 虽然 FALSE 和 UNKNOWN 都不返回值，但是 FALSE 和 UNKNOWN 之间存在着重要的逻辑差异；FALSE 的相反值 ("NOT FALSE") 是 TRUE，而 UNKNOWN 的相反值则不表示任何已知内容。例如，`1 = 2` 计算得到 FALSE，而 `1 != 2`（1 不等于 2）计算得到 TRUE。

  但是，如果比较中包括 NULL，则您不能对表达式取非以得到相对的一组行或相反的真假值。UNKNOWN 值保持 UNKNOWN。

- **用某个值替代 NULL 值** 可以通过 ISNULL 内置函数用某个特定的值替代 NULL 值。替代的目的只是为了进行显示，未影响实际列值。语法是：

  ISNULL(*expression*, *value*)

例如，使用以下语句从 Departments 中选择所有行，然后将列 DepartmentHeadID 中的所有 NULL 值显示为值 -1。

```
SELECT DepartmentID,
       DepartmentName,
       ISNULL( DepartmentHeadID, -1 ) AS DepartmentHead
   FROM Departments;
```

- **计算得到 NULL 的表达式** 如果所有操作数都为 NULL 值，则使用含算术或位运算符的表达式计算得到 NULL。例如，如果 column1 为 NULL，则 `1 + column1` 计算得到 NULL。请参见“算术运算符”一节 《SQL Anywhere 服务器 - SQL 参考》和“位运算符”一节 《SQL Anywhere 服务器 - SQL 参考》。
- **连接字符串和 NULL** 如果您将字符串和 NULL 连接，则表达式计算得到字符串。例如，下面的语句返回字符串 abcdef：

```
SELECT 'abc' || NULL || 'def';
```

使用逻辑运算符连接条件

逻辑运算符 AND、OR 和 NOT 可用来在 WHERE 子句中连接搜索条件。当语句中使用多个逻辑运算符时，通常先计算 AND 运算符，然后再计算 OR 运算符。您可以使用括号更改执行的顺序。

使用 AND

AND 运算符连接两个或多个条件并且只有当所有条件都为真时才返回结果。例如，以下查询只查找联系人的姓是 Purcell 并且联系人的名是 Beth 的那些行。

```
SELECT *
   FROM Contacts
   WHERE GivenName = 'Beth'
      AND Surname = 'Purcell';
```

使用 OR

OR 运算符连接两个或多个条件，并且当*任何*一个条件为真时，它就会返回结果。以下查询搜索 GivenName 列中包含 Elizabeth 各种变体的那些行。

```
SELECT *
   FROM Contacts
   WHERE GivenName = 'Beth'
      OR GivenName = 'Liz';
```

使用 NOT

NOT 运算符对它后面的表达式取非。以下查询列出所有不居住在 California 的联系人：

```
SELECT *
   FROM Contacts
   WHERE NOT State = 'CA';
```


比较搜索条件中的日期

可以使用等号以外的运算符选择一组满足搜索条件的行。不等式运算符（< 和 >）可用于比较数字、日期、甚至是字符串。

♦ **列出所有生于 1964 年 3 月 13 日以前的雇员：**

- 在 Interactive SQL 中，执行以下查询：

```
SELECT Surname, BirthDate
   FROM Employees
   WHERE BirthDate < 'March 13, 1964'
   ORDER BY BirthDate DESC;
```

Surname	BirthDate
Ahmed	1963-12-12
Dill	1963-07-19
Rebeiro	1963-04-12
Garcia	1963-01-23
Pastor	1962-07-14
...	...

注意

- **自动转换为日期** SQL Anywhere 数据库服务器知道 BirthDate 列包含日期，并自动将字符串 `'March 13, 1964'` 转换为日期。
- **指定日期的方式** 指定日期有多种方式。例如：

```
'March 13, 1964'
'1964/03/13'
'1964-03-13'
```

  可以通过设置 date_order 数据库选项配置查询中日期的解释。请参见“date_order 选项 [数据库]”一节 《SQL Anywhere 服务器 - 数据库管理》。

  yyyy/mm/dd 或 *yyyy-mm-dd* 格式的日期总是会被明确地识别为日期，而与 date_order 设置无关。

- **其它比较运算符** SQL Anywhere 支持若干个比较运算符。请参见“比较运算符”一节 《SQL Anywhere 服务器 - SQL 参考》。

通过发音匹配行

使用 SOUNDEX 函数，您可以通过发音匹配行。例如，假设有一个电话留言找姓名发音类似 "Ms. Brown" 的人。可以执行以下查询来搜索姓名发音与 Brown 类似的雇员。

♦ **列出姓发音与 Brown 类似的雇员**

● 在 Interactive SQL 中，执行以下查询：

```
SELECT Surname, GivenName
   FROM Employees
   WHERE SOUNDEX( Surname ) = SOUNDEX( 'Brown' );
```

Surname	GivenName
Braun	Jane

SOUNDEX 使用的算法使该函数主要用于语言为英语的数据库。请参见“SOUNDEX 函数 [String]”一节 《SQL Anywhere 服务器 - SQL 参考》。

ORDER BY 子句：对结果进行排序

除非另有要求，否则数据库服务器按照没有任何意义的顺序返回表中的行。通常，以一种更有意义的顺序查看表中的行将十分有用。例如，您可能要按字母顺序查看产品。

您可以通过在 SELECT 语句末尾添加 ORDER BY 子句对结果集中的行进行排序。此 SELECT 语句的语法如下：

SELECT *column-name-1*, *column-name-2*,...
FROM *table-name*
ORDER BY *order-by-column-name*

必须用要查询的列名和表名分别替换 *column-name-1*、*column-name-2* 和 *table-name*，并用该表中的某列替换 *order-by-column-name*。像以前一样，您可以使用星号作为表中所有列的简写形式。

♦ **按字母顺序列出产品**

- 在 Interactive SQL 中，执行以下查询：

```
SELECT ID, Name, Description
   FROM Products
   ORDER BY Name;
```

ID	Name	Description
400	Baseball Cap	Cotton Cap
401	Baseball Cap	Wool cap
700	Shorts	Cotton Shorts
600	Sweatshirt	Hooded Sweatshirt
...	...	...

注意

- **子句的顺序有意义** ORDER BY 子句必须在 FROM 子句和 SELECT 子句之后。
- **您可以指定升序或降序** 缺省顺序是升序。您可以通过在子句的末尾添加关键字 DESC 指定降序，如在下面的查询中所示：

```
SELECT ID, Quantity
   FROM Products
   ORDER BY Quantity DESC;
```

ID	Quantity
400	112
700	80

ID	Quantity
302	75
301	54
600	39
...	...

- **您可以根据多个列进行排序** 以下查询先按大小（按字母顺序）排序，然后按名称进行排序：

```
SELECT ID, Name, Size
   FROM Products
   ORDER BY Size, Name;
```

ID	Name	Size
600	Sweatshirt	Large
601	Sweatshirt	Large
700	Shorts	Medium
301	Tee Shirt	Medium
...	...	...

- **ORDER BY 列不一定在选择列表中** 以下查询通过单价对产品进行排序，即使价格不包括在结果集中

```
SELECT ID, Name, Size
   FROM Products
   ORDER BY UnitPrice;
```

ID	Name	Size
500	Visor	One size fits all
501	Visor	One size fits all
300	Tee Shirt	Small
400	Baseball Cap	One size fits all
...	...	...

- **如果您不使用 ORDER BY 子句并且多次执行查询，您可能会得到不同的结果** 这是因为 SQL Anywhere 可能以不同的顺序返回相同的结果集。当缺少 ORDERBY 子句时，SQL Anywhere 以

最有效的顺序返回行。这意味着结果集的外观可能随您上一次访问行的时间和其它因素的不同而有所变化。确保以特定的顺序返回行的唯一方法是使用 ORDER BY。

使用索引提高 ORDER BY 的性能

有时候，SQL Anywhere 数据库服务器执行具有 ORDER BY 子句的查询有多种可能的方式。您可以使用索引使数据库服务器更有效地对表进行搜索。

包含 WHERE 和 ORDER BY 子句的查询

可以以多种可能方式执行的查询的示例是同时具有 WHERE 子句和 ORDER BY 子句的查询。

```
SELECT *
   FROM Customers
   WHERE ID > 300
   ORDER BY CompanyName;
```

在这个示例中，SQL Anywhere 必须在两种策略之间进行选择：

1. 按照公司名称的顺序检查整个 Customers 表，查看每行的客户 ID 是否大于 300。
2. 使用 ID 列上的键只读取 ID 大于 300 的公司。然后，需要按照公司名称对结果进行排序。

如果只有很少的 ID 值大于 300，则第二种策略较好，这是因为只对很少的行进行扫描并迅速进行排序。如果大多数 ID 值都大于 300，则第一种策略较好，这是因为不需要进行排序。

解决问题

创建一个有关 ID 和 CompanyName 的两列索引就可以解决上面示例中的问题。SQL Anywhere 可以使用此索引从表中以正确的顺序选择行。但是，请记住，索引占用数据库文件中的空间，保持索引为最新还需要一些系统开销。不要无限制地创建索引。请参见“使用索引”一节第 221 页。

集合函数

有些查询会检查表中数据的某些方面，这些方面反映组行（而不是单行）的属性。例如，您可能要计算客户为某一订单支付的平均款额，或者需要了解有多少雇员为各个部门工作。对于这些类型的任务，您可以使用**集合**函数和 GROUP BY 子句。

集合函数为一组行返回一个值。如果没有 GROUP BY 子句，集合函数将为满足查询其它方面的所有行返回一个值。

♦ **列出公司的雇员数**

● 在 Interactive SQL 中，执行以下查询：

```
SELECT COUNT( * )
   FROM Employees;
```

COUNT(*)
75

结果集仅包含一列（标题为 COUNT(*)）和一行（包含雇员的总人数）。

♦ **列出公司的雇员数以及年纪最大和最小的雇员的出生日期**

● 在 Interactive SQL 中，执行以下查询：

```
SELECT COUNT( * ), MIN( BirthDate ), MAX( BirthDate )
   FROM Employees;
```

COUNT(*)	MIN(Employees.BirthDate)	MAX(Employees.BirthDate)
75	1936-01-02	1973-01-18

函数 COUNT、MIN 和 MAX 称作**集合函数**。集合函数用于汇总信息。其它集合函数包括统计函数，如 AVG、STDDEV 和 VARIANCE。除 COUNT 之外的所有函数都需要一个参数。请参见“集合函数”一节 《SQL Anywhere 服务器 - SQL 参考》。

将集合函数应用于分组数据

除了提供有关整个表的信息之外，集合函数还可应用于行组。GROUP BY 子句将行分组，而集合函数将为每一行组返回一个值。

示例

♦ **列出销售代表和每位销售代表获得的订单数**

● 在 Interactive SQL 中，执行以下查询：

```
SELECT SalesRepresentative, COUNT( * )
   FROM SalesOrders
```

```
GROUP BY SalesRepresentative
ORDER BY SalesRepresentative;
```

SalesRepresentative	count(*)
129	57
195	50
299	114
467	56
...	...

GROUP BY 子句通知 SQL Anywhere 对以其它方式返回的所有行的集合进行分区。每个分区（即组）中的所有行在指定的列中都具有相同的值。每个唯一值或值集仅对应于一个组。在本例中，每组中的所有行都具有相同的 SalesRepresentative 值。

诸如 COUNT 等集合函数适用于每个组中的行。因此，此结果集会显示每个组中行的总数。查询的结果由对应于每个销售代表 ID 号的行组成。每行都包含销售代表 ID 以及该销售代表获得的销售订单总数。

每当使用 GROUP BY 时，结果表中都有一行对应于在 GROUP BY 子句中指定的每个列或列集。请参见“GROUP BY 子句：将查询结果划分为组”一节第 344 页。

GROUP BY 常见错误

GROUP BY 的一个常见错误是试图获取无法正确放入一个组中的信息。例如，下面的查询会生成一个错误。

```
SELECT SalesRepresentative, Surname, COUNT( * )
   FROM SalesOrders KEY JOIN Employees
   GROUP BY SalesRepresentative;
```

之所以会报告错误 [**对选择列表中 'Surname' 的函数或列引用还必须出现在 GROUP BY 中**]，是因为 SQL Anywhere 无法确定对于给定 ID 的雇员，每个结果行是否都具有相同的姓氏。

要解决此问题，请将该列添加到 GROUP BY 子句。

```
SELECT SalesRepresentative, Surname, COUNT( * )
   FROM SalesOrders KEY JOIN Employees
   GROUP BY SalesRepresentative, Surname
   ORDER BY SalesRepresentative;
```

如果这不适用，则可以改用集合函数来仅选择一个值：

```
SELECT SalesRepresentative, MAX( Surname ), COUNT( * )
   FROM SalesOrders KEY JOIN Employees
   GROUP BY SalesRepresentative
   ORDER BY SalesRepresentative;
```

MAX 函数从每个组的详细信息行中选择 Surname 最大值（按字母顺序排在最后的姓氏）。由于只能有一个不同的最大值，所以该语句有效。在本例中，相同的 Surname 会出现在一个组中的每个详细信息行上。

限制组

您已经知道了如何使用 WHERE 子句来限制结果集中的行。您可以使用 HAVING 子句来限制组中的行。

♦ **列出其订单数超过 55 的所有销售代表**

● 在 Interactive SQL 中，执行以下查询：

```
SELECT SalesRepresentative, COUNT( * ) AS orders
FROM SalesOrders KEY JOIN Employees
GROUP BY SalesRepresentative
HAVING count( * ) > 55
ORDER BY orders DESC;
```

SalesRepresentative	orders
299	114
129	57
1142	57
467	56

另请参见“HAVING 子句：选择数据组”一节第 349 页。

组合 WHERE 和 HAVING 子句

有时，您可以使用 WHERE 子句或 HAVING 子句指定同一组行。在这种情况下，其中一种方法并不比另一种方法更有效。优化程序始终会自动分析您输入的每条语句，并选择执行该语句的有效方法。最好使用能够最清晰地描述所需结果的语法。通常，这意味着在前面的子句中消除不需要的行。

示例

要列出其订单数超过 55 且 ID 大于 1000 的所有销售代表，请输入以下语句。

```
SELECT SalesRepresentative, COUNT( * )
FROM SalesOrders
WHERE SalesRepresentative > 1000
GROUP BY SalesRepresentative
HAVING count( * ) > 55
ORDER BY SalesRepresentative;
```

以下语句会产生相同的结果。

```
SELECT SalesRepresentative, COUNT( * )
FROM SalesOrders
GROUP BY SalesRepresentative
HAVING count( * ) > 55 AND SalesRepresentative > 1000
ORDER BY SalesRepresentative;
```


SQL Anywhere 会检测到这两条语句都描述了相同的结果集，从而高效地执行各条语句。

全文搜索

全文搜索可在数据库中快速查找某个术语（单词）的所有实例，而不必扫描表行，也不必知道术语存储在哪一列。全文搜索使用**文本索引**进行操作。文本索引在索引列中存储术语的位置信息。使用文本索引查找包含术语的行会比扫描表中的每一行要快，其原因就是使用常规索引查找包含给定值的行会比较快的那些原因。请参见“文本索引”一节第 298 页。

全文搜索使用 CONTAINS 搜索条件。全文搜索与使用谓语（例如 LIKE、REGEXP 和 SIMILAR TO）进行的搜索不同，因为全文搜索的匹配是基于术语，而不是基于模式。请参见“CONTAINS 搜索条件”一节 《SQL Anywhere 服务器 - SQL 参考》。

全文搜索中的字符串比较使用数据库的所有常规归类设置。例如，如果将数据库配置为不区分大小写，则全文搜索将不区分大小写。请参见“了解归类”一节 《SQL Anywhere 服务器 - 数据库管理》。

除非特别说明，否则全文搜索可利用 SQL Anywhere 支持的所有国际功能。请参见“SQL Anywhere 的国际功能”一节 《SQL Anywhere 服务器 - 数据库管理》。

要在包含中文、日文和朝鲜文 (CJK) 数据的数据库中执行全文搜索，请参见白皮书《Performing Full Text Searches on Chinese, Japanese, and Korean Data in SQL Anywhere 11》，该白皮书可在网址 http://www.sybase.com/detail?id=1061814 获得。

执行全文查询

通过在 SELECT 语句的 FROM 子句中使用 CONTAINS 子句，或者在 WHERE 子句中使用 CONTAINS 搜索条件（谓语），可以执行全文查询。两种方法都返回相同的行；但 CONTAINS 子句还会返回匹配行的分数。

例如，以下两个语句查询 MarketingInformation 表中的 Description 列，并返回 Description 列中包含术语 **cotton** 的值所在的行。第二个语句也会为匹配行返回分数。

```
SELECT *
   FROM MarketingInformation
   WHERE CONTAINS ( Description, 'cotton' );

SELECT *
   FROM MarketingInformation
   CONTAINS ( Description, 'cotton' );
```

请参见“FROM 子句”一节 《SQL Anywhere 服务器 - SQL 参考》和“CONTAINS 搜索条件”一节 《SQL Anywhere 服务器 - SQL 参考》。

文本配置对象

文本配置对象控制构建或刷新文本索引时要包含在索引中的术语，以及如何解释全文查询。各个文本配置对象的设置均在 ISYSTEXTCONFIG 系统表中存储为一行。

数据库服务器创建或刷新文本索引时，将使用创建文本索引期间指定的文本配置对象的设置。如果在创建文本索引时未指定文本配置对象，则数据库服务器将根据要建立索引的列的数据类型来选择某个缺省文本配置对象。SQL Anywhere 提供了两种缺省文本配置对象。请参见“文本配置对象示例”一节第 291 页。

要查看现有文本配置对象的设置，请查询 SYSTEXTCONFIG 系统视图。请参见“SYSTEXTCONFIG 系统视图”一节 《SQL Anywhere 服务器 - SQL 参考》。

文本配置对象设置

SQL Anywhere 提供两种缺省文本配置对象，即与非 NCHAR 数据一起使用的 default_char 以及 default_nchar。有关其设置的信息，请参见“缺省文本配置对象”一节第 291 页。

下表说明了文本配置对象设置、各种设置如何影响索引内容以及如何解释全文搜索查询。有关文本配置对象的示例以及它们对文本索引和全文搜索的影响，请参见“文本配置对象示例”一节第 291 页。

- **术语断开器算法 (TERM BREAKER)** TERM BREAKER 设置指定将字符串分解为术语时所使用的算法。可以选择 GENERIC（缺省设置）以存储术语，或选择 NGRAM 以存储 n 元语法词。**n 元语法词** 是一组长度为 *n* 的字符，其中 *n* 是 MAXIMUM TERM LENGTH 的值。

  无论指定哪种术语断开器，数据库服务器都将术语插入文本索引时术语的原始位置信息记录在文本索引中。如果是 n 元语法词，则存储 n 元语法词的位置信息，而不是原始术语的位置信息。

TERM BREAKER 对文本索引的影响	TERM BREAKER 对查询术语的影响
GENERIC 文本索引 构建 GENERIC 文本索引（缺省设置）时，数据库服务器会将出现在非字母数字字符之间的字母数字字符组处理为术语。定义术语后，超出术语长度设置的术语和在非索引字表中找到的术语将只进行计数而不插入到文本索引当中。 GENERIC 文本索引的性能可能要高于 NGRAM 文本索引。但无法对 GENERIC 文本索引执行模糊搜索。 **NGRAM 文本索引** 构建 NGRAM 文本索引时，数据库服务器会将非字母数字字符之间的任意一组字母数字字符处理为术语。定义术语后，数据库服务器将术语分解为 n 元语法词。这样便可放弃长度小于 *n* 的术语以及非索引字表中的 n 元语法词。 例如，对于 MAXIMUM TERM LENGTH 为 3 的 NGRAM 文本索引，字符串 'my red table' 在文本索引中表示为以下 n 元语法词：red tab abl ble。	**GENERIC 文本索引** 查询 GENERIC 文本索引时，将按照术语已建立索引的方式来处理查询字符串中的术语。通过比较查询术语和文本索引中的术语来执行匹配。 **NGRAM 文本索引** 查询 NGRAM 文本索引时，将按照术语已建立索引的方式来处理查询字符串中的术语。通过比较查询术语中的 n 元语法词和索引术语中的 n 元语法词来执行匹配。

- **最小术语长度设置 (MINIMUM TERM LENGTH)** MINIMUM TERM LENGTH 设置指定插入到索引中的术语或在全文查询中搜索的术语的最小长度（以字符为单位）。MINIMUM TERM LENGTH 不可用于 NGRAM 文本索引。

  MINIMUM TERM LENGTH 对于前缀搜索有特殊含义。请参见“前缀搜索”一节第 308 页。

  MINIMUM TERM LENGTH 的值必须大于 0。如果将其设置为大于 MAXIMUM TERM LENGTH，则 MAXIMUM TERM LENGTH 会自动调整为与 MINIMUM TERM LENGTH 相等。

  MINIMUM TERM LENGTH 的缺省值从缺省文本配置对象的设置中获得，通常为 1。请参见“缺省文本配置对象”一节第 291 页。

MINIMUM TERM LENGTH 对文本索引的影响	MINIMUM TERM LENGTH 对查询术语的影响
GENERIC 文本索引 对于 GENERIC 文本索引，长度小于 MINIMUM TERM LENGTH 的单词将不会包含在文本索引之中。 **NGRAM 文本索引** 对于 NGRAM 文本索引，将忽略此设置。	**GENERIC 文本索引** 查询 GENERIC 文本索引时，将忽略长度小于 MINIMUM TERM LENGTH 的查询术语，因为它们不会在文本索引中存在。 **NGRAM 文本索引** MINIMUM TERM LENGTH 的设置对 NGRAM 文本索引上的全文查询没有影响。

- **最大术语长度设置 (MAXIMUM TERM LENGTH)** 根据术语断开器算法的不同，MAXIMUM TERM LENGTH 设置的使用方法也不同。

MAXIMUM TERM LENGTH 的值必须小于或等于 60。如果将其设置为小于 MINIMUM TERM LENGTH，则 MINIMUM TERM LENGTH 会自动调整为与 MAXIMUM TERM LENGTH 相等。

此设置的缺省值从缺省文本配置对象的设置中获得，通常为 20。请参见“缺省文本配置对象”一节第 291 页。

MAXIMUM TERM LENGTH 对文本索引的影响	MAXIMUM TERM LENGTH 对查询术语的影响
GENERIC 文本索引 对于 GENERIC 文本索引，MAXIMUM TERM LENGTH 指定插入到文本索引中的术语的最大长度（以字符为单位）。 **NGRAM 文本索引** 对于 NGRAM 文本索引，MAXIMUM TERM LENGTH 用于确定术语分解后形成的 n 元语法词的长度。如何选择适当的 MAXIMUM TERM LENGTH 长度取决于相应的语言。对于英文，典型值为 4 个或 5 个字符；对于中文，典型值为 2 个或 3 个字符。	**GENERIC 文本索引** 对于 GENERIC 文本索引，将忽略长度大于 MAXIMUM TERM LENGTH 的查询术语，因为它们不会在文本索引中存在。 **NGRAM 文本索引** 对于 NGRAM 文本索引，查询术语将分解成长度为 *n* 的 n 元语法词，其中 *n* 与 MAXIMUM TERM LENGTH 的值相同。然后数据库服务器使用 n 元语法词搜索文本索引。长度小于 MAXIMUM TERM LENGTH 的术语将被忽略，因为它们与文本索引中的 n 元语法词不匹配。

- **非索引字表设置 (STOPLIST)** 非索引字表设置指定不得对其建立索引的术语。

  此设置的缺省值从缺省文本配置对象的设置中获得，通常含有空的非索引字表。请参见“缺省文本配置对象”一节第 291 页。

STOPLIST 对文本索引的影响	STOPLIST 对查询术语的影响
GENERIC 文本索引 对于 GENERIC 文本索引，非索引字表中的术语不会插入到文本索引之中。 **NGRAM 文本索引** 对于 GENERIC 文本索引，由非索引字表中的术语形成的 n 元语法词不会包含在文本索引之中。	**GENERIC 文本索引** 对于 GENERIC 文本索引，将忽略非索引字表中的查询术语，因为它们不会在文本索引中存在。 **NGRAM 文本索引** 非索引字表中的术语分解为 n 元语法词，然后将 n 元语法词用于非索引字表。类似地，查询术语将分解为 n 元语法词，然后删除任何与非索引字表中的 n 元语法词相匹配的项目，因为它们不会在文本索引中存在。

请认真考虑是否将术语置于非索引字表中。尤其注意不要将具有非字母数字字符（如撇号和横线）的单词包括在内。这些字符用作术语断开器。例如，单词 you'll（必须指定为 `['you''ll']`）将被分解为 you 和 ll，并在非索引字表中存储为这两个术语。随后如果全文搜索 `['you']` 或 `['they''ll']`，则会受到负面影响。

对于 NGRAM 文本索引，非索引字表可能导致意外的结果，因为所储存的非索引字表实际上是 n 元语法词的形式，而不是指定的实际非索引字表术语形式。例如，对于 MAXIMUM TERM LENGTH 为 3 的 NGRAM 文本索引，如果指定 `[STOPLIST 'there']`，则以下 n 元语法词将存储为非索引字表：the her ere。这会影响查询任何包含 n 元语法词 the、her 和 ere 的术语的能力。

注意
指定字符串文字时的限制同样适用于非索引字表。例如，必须对撇号进行转义等。有关格式化字符串文字的详细信息，请参见“字符串文字”一节 《SQL Anywhere 服务器 - SQL 参考》。

示例目录包含装载多种语言的非索引字表的示例代码。建议仅对 GENERIC 文本索引使用这些示例非索引字表。有关示例目录的位置，请参见“示例目录”一节 《SQL Anywhere 服务器 - 数据库管理》。

另请参见

- “缺省文本配置对象”一节第 291 页
- “创建文本配置对象”一节第 288 页
- “变更文本配置对象”一节第 289 页
- “文本配置对象示例”一节第 291 页
- “模糊搜索”一节第 314 页
- “文本索引”一节第 298 页
- “CONTAINS 搜索条件”一节 《SQL Anywhere 服务器 - SQL 参考》
- “CREATE TEXT CONFIGURATION 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “ALTER TEXT CONFIGURATION 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “DROP TEXT CONFIGURATION 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “SYSTEXTIDX 系统视图”一节 《SQL Anywhere 服务器 - SQL 参考》

创建文本配置对象

使用 SQL 语句创建文本配置对象时，使用另一文本配置对象作为模板。然后更改配置设置并使用新的文本配置创建自己的文本索引。

在 Sybase Central 中创建文本配置对象时，可以使用 [**创建文本配置对象向导**] 在创建期间配置设置。

♦ 创建文本配置对象 (SQL)

1. 以具有 DBA 权限的用户身份连接到数据库。
2. 执行 CREATE TEXT CONFIGURATION 语句。

   例如，以下语句将 default_char 文本配置对象用作模板来创建名为 myTxtConfig 的文本配置对象：

   ```
   CREATE TEXT CONFIGURATION myTxtConfig FROM default_char;
   ```

♦ 创建文本配置对象 (Sybase Central)

1. 以具有 DBA 权限的用户身份连接到数据库。
2. 右击 [**文本配置对象**]，然后选择 [**新建**] » [**文本配置对象**]。
3. 请按照 [**创建文本配置对象向导**] 中的说明进行操作。

   新文本配置对象随即出现在 [**文本配置对象**] 窗格中。

另请参见

- “全文搜索”一节第 284 页
- “文本配置对象设置”一节第 285 页
- “文本配置对象示例”一节第 291 页
- “查看数据库中的文本配置对象”一节第 290 页
- “CREATE TEXT CONFIGURATION 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “ALTER TEXT CONFIGURATION 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “缺省文本配置对象”一节第 291 页

变更文本配置对象

文本索引依赖于创建索引时使用的文本配置对象。因此，必须先截断相关文本索引，然后才可以更改文本配置对象。由于 IMMEDIATE REFRESH 文本索引无法截断，因此必须先删除该索引，然后才能更改文本配置对象。

♦ 更改文本配置对象 (SQL)

1. 以具有 DBA 权限的用户身份或以文本配置对象所有者的身份连接到数据库。
2. 执行 ALTER TEXT CONFIGURATION 语句。例如，以下语句变更 myTxtConfig 文本配置对象的最小术语长度：

```
ALTER TEXT CONFIGURATION myTxtConfig
MINIMUM TERM LENGTH 2;
```

♦ 变更文本配置对象 (Sybase Central)

1. 以具有 DBA 权限的用户身份或以文本配置对象所有者的身份连接到数据库。
2. 在左窗格中，单击 [**文本配置对象**]。
3. 右击文本配置对象，然后选择 [**属性**]。
4. 编辑文本配置对象属性，并单击 [**确定**]。

另请参见

- “全文搜索”一节第 284 页
- “文本配置对象设置”一节第 285 页
- “文本配置对象示例”一节第 291 页
- “查看数据库中的文本配置对象”一节第 290 页
- “CREATE TEXT CONFIGURATION 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “ALTER TEXT CONFIGURATION 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “缺省文本配置对象”一节第 291 页

数据库选项和文本配置对象

创建文本配置对象时，date_format、time_format 和 timestamp_format 数据库选项的当前设置将与文本配置对象一同存储。这是因为创建和刷新依赖于文本配置对象的文本索引时，这些设置会影响字符串转换。

将设置与文本配置对象一同存储允许您更改这些数据库选项的设置，而无需更改存储在相关文本索引中的数据的格式。

如果要更改文本索引中表示日期和时间的字符串的格式，必须执行以下操作：

1. 删除文本索引和文本配置对象。
2. 将数据库选项更改为所需的格式。
3. 创建文本配置对象。
4. 使用新文本配置对象创建文本索引。

注意
创建或刷新文本索引时，必须将 conversion_error 选项设置为 ON。

另请参见

- “文本配置对象设置”一节第 285 页
- “date_format 选项 [数据库]”一节 《SQL Anywhere 服务器 - 数据库管理》
- “time_format 选项 [兼容性]”一节 《SQL Anywhere 服务器 - 数据库管理》
- “timestamp_format 选项 [兼容性]”一节 《SQL Anywhere 服务器 - 数据库管理》
- “conversion_error 选项 [兼容性]”一节 《SQL Anywhere 服务器 - 数据库管理》

查看数据库中的文本配置对象

可以通过 Sybase Central 或 SQL 语句来查看有关数据库中文本配置对象的信息。

♦ 查看文本配置对象的设置 (Sybase Central)

1. 以具有 DBA 权限的用户身份或以文本配置对象所有者的身份连接到数据库。
2. 在左窗格中，单击 [**文本配置对象**]。
3. 双击文本配置对象以查看其设置。

♦ 查看文本配置对象的设置 (SQL)

1. 以具有 DBA 权限的用户身份或以文本配置对象所有者的身份连接到数据库。
2. 查询 SYSTEXTCONFIG 系统视图，语句如下：

```
SELECT * FROM SYSTEXTCONFIG;
```


另请参见

- “文本配置对象设置”一节第 285 页
- “SYSTEXTCONFIG 系统视图”一节 《SQL Anywhere 服务器 - SQL 参考》

文本配置对象示例

有关文本配置对象设置的详细说明以及查询文本索引时不同设置如何影响文本索引内容和所返回结果的信息，请参见“文本配置对象设置”一节第 285 页。

要查看数据库中所有文本配置对象以及它们包含的设置的列表，请查询 SYSTEXTCONFIG 系统视图（例如 [`SELECT * FROM SYSTEXTCONFIG`]）。请参见“SYSTEXTCONFIG 系统视图”一节《SQL Anywhere 服务器 - SQL 参考》。

缺省文本配置对象

SQL Anywhere 提供两种缺省文本配置对象，即 **default_nchar** 和 **default_char**，分别用于 NCHAR 和非 NCHAR 数据。这些配置在首次尝试创建文本配置对象或文本索引时创建。如果因误操作而删除了某个配置，则下次尝试创建文本配置对象或文本索引时将重新创建该配置。

下表所示为在安装时 default_char 和 default_nchar 的设置。选择这些设置是因为它们能够最好地适合大多数基于字符的语言。强烈建议不要更改缺省文本配置对象中的设置。

设置	安装值
TERM BREAKER	0 (GENERIC)
MINIMUM TERM LENGTH	1
MAXIMUM TERM LENGTH	20
STOPLIST	（空）

如果删除了某个缺省文本配置对象，则下次创建文本索引或文本配置对象时将自动重新创建该对象。请参见“DROP TEXT CONFIGURATION 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

文本配置对象示例

下表说明了不同文本配置对象的设置、各种设置如何影响索引内容以及如何解释全文查询字符串。所有示例均使用字符串 `'I'm not sure I understand'`。

配置设置	建立索引的术语	查询解释
TERM BREAKER GENERIC MINIMUM TERM LENGTH 1 MAXIMUM TERM LENGTH 20 STOPLIST "	`I m not sure I understand`	`"I m" AND not AND sure AND I AND understand'`
TERM BREAKER GENERIC MINIMUM TERM LENGTH 2 MAXIMUM TERM LENGTH 20 STOPLIST 'not and'	`sure understand`	`'sure AND understand'.`

TERM BREAKER NGRAM
MAXIMUM TERM LENGTH 3
STOPLIST 'not and'
	`sur ure und nde der ers rst sta tan`	`'sur AND ure AND und AND nde AND der AND ers AND rst AND sta AND tan'.`
		在模糊搜索的情况下：
		`'sur OR ure OR und OR nde OR der OR ers OR rst OR sta OR tan'`

TERM BREAKER GENERIC MINIMUM TERM LENGTH 1 MAXIMUM TERM LENGTH 20 STOPLIST 'not and'	`I m sure I understand`	`'"I m" AND sure AND I AND understand'.`

配置设置	建立索引的术语	查询解释
TERM BREAKER NGRAM MAXIMUM TERM LENGTH 20 STOPLIST 'not and'	不对任何内容建立索引，因为没有长度等于或大于 20 个字符的术语。 这说明了 MAXIMUM TERM LENGTH 对 GENERIC 和 NGRAM 文本索引的不同影响；对于 NGRAM 文本索引，MAXIMUM TERM LENGTH 设置插入到文本索引中的 n 元语法词的长度。	搜索将返回空结果集，因为通过查询字符串无法形成 20 个字符的 n 元语法词。

字符串解释示例

下表提供如何解释文本配置对象字符串设置的示例。

“解释的字符串”列中括号内的数字表示为各个术语存储的位置信息。文档中这些数字用于说明目的。实际存储的术语不包括括号内数字。

配置设置	String	解释的字符串
TERM BREAKER GENERIC MINIMUM TERM LENGTH 3 MAXIMUM TERM LENGTH 20	`'w*'`	`'"w*(1)"'`
	`'we*'`	`'"we*(1)"'`
	`'wea*'`	`'"wea*(1)"'`
	`'we* -the'`	`'"we*(1)" -"the(1)"'`
	`'we* the'`	`"we*(1)" & "the(1)"'`
	`'for* | wonderl*'`	`'"for*(1)" | "wonderl*(1)"'`
	`'wonderlandwonderlandwonderland*'`	`''`
	`'"tr* weather"'`	`'"weather(1)"'`
	`'"tr* the weather"'`	`'"the(1) weather(2)"'`
	`'"wonderlandwonderlandwonderland* wonderland"'`	`'"wonderland(1)"'`

配置设置	String	解释的字符串
	'"wonderlandwonderlandwonderland* weather"'	'"weather(1)"'
	'"the_wonderlandwonderlandwonderland* weather"'	'"the(1) weather(3)"'
	'the_wonderlandwonderlandwonderland* weather'	'"the(1)" & "weather(1)"'
	'"light_a* the end" & tunnel'	'"light(1) the(3) end(4)" & "tunnel(1)"'
	light_b* the end" & tunnel'	'"light(1) the(3) end(4)" & "tunnel(1)"'
	'"light_at_b* end"'	'"light(1) end(4)"'
	'and-te*'	'"and(1) te*(2)"'
	'a_long_and_t* & journey'	'"long(2) and(3) t*(4)" & "journey(1)"'
TERM BREAKER NGRAM MAXIMUM TERM LENGTH 3	'w*'	'"w*(1)"'
	'we*'	'"we*(1)"'
	'wea*	'"wea(1)"'
	'we* -the'	'"we*(1)" -"the(1)"'
	'we* the'	'"we*(1)" & "the(1)"'
	'for | la*'	'"for(1)" | "la*(1)"'
	'weath*'	'"wea(1) eat(2) ath(3)"'
	'"ful weat*"'	'"ful(1) wea(2) eat(3)"'

配置设置	String	解释的字符串
	'"wo* la*"'	'"wo*(1)" & "la*(2)"'
	'"la* won* "'	'"la*(1)" & "won(2)"'
	'"won* weat*"'	'"won(1)" & "wea(2) eat(3)"'
	'"won* weat"'	'"won(1)" & "wea(2) eat(3)"'
	'"wo* weat*"'	'"wo*(1)" & "wea(2) eat(3)"'
	'"weat* wo* "'	'"wea(1) eat(2)" & "wo*(3)"'
	'"wo* weat"'	'"wo*(1)" & "wea(2) eat(3)"'
	'"weat wo* "'	'"wea(1) eat(2) wo*(3)"'
	'w* NEAR[1] f*'	'"w*(1)" & "f*(1)"'
	'weat* NEAR[1] f*'	"wea(1) eat(2)" & "f*(1)"'
	'f* NEAR[1] weat*'	'"f*(1)" & "wea(1) eat(2)"'
	'weat NEAR[1] f*'	'"wea(1) eat(2)" & "f*(1)"'
	'f* NEAR[1] weat'	'"f*(1)" & "wea(1) eat(2)"'
	'for NEAR[1] weat*'	'"for(1)" & "wea(1) eat(2)"'
	'weat* NEAR[1] for'	'"wea(1) eat(2)" & "for(1)"'
	'and_tedi*'	'"and(1) ted(2) edi(3)"'

配置设置	String	解释的字符串
	'and-t*'	'"and(1) t*(2)"'
	'"and_tedi*"'	'"and(1) ted(2) edi(3)"'
	'"and-t*"'	'"and(1) t*(2)"'
	'"ligh* at_the_end of_the tun* nel"'	'"lig(1) igh(2)" & ("the(4) end(5) the(7) tun(8)" & "nel(9)")'
	'"ligh* at_the_end_of_the_tun* nel"'	'"lig(1) igh(2)" & ("the(4) end(5) the(7) tun(8)" & "nel(9)")'
	'"at_the_end of_the tun* ligh* nel"'	'"the(2) end(3) the(5) tun(6)" & ("lig(7) igh(8)" & "nel(9)")'
	'l* NEAR[1] and_t*'	"l*(1)" & "and(1) t*(2)"'
	'long NEAR[1] and_t*'	'"lon(1) ong(2)" & "and(1) t*(2)"'
	'end NEAR[3] tunne*'	'"end(1)" & "tun(1) unn(2) nne(3)"'
TERM BREAKER NGRAM MAXIMUM TERM LENGTH 3 SKIPPED TOKENS IN TABLE AND IN QUERIES	'"cat in a hat"'	'"cat(1) hat(4)"'
	'"cat in_a hat"'	'"cat(1) hat(4)"'
	'"cat_in_a_hat"'	'"cat(1) hat(4)"'
	'"cat_in a_hat"'	'"cat(1) hat(4)"'
	'cat in a hat'	'"cat(1)" & "hat(1)"'
	'cat in_a hat'	'"cat(1)" & "hat(1)"'
	'"ice hat"'	'"ice(1) hat(2)"'

配置设置	String	解释的字符串
	'ice NEAR[1] hat'	'"ice(1)" NEAR[1] "hat(1)"'
	'ear NEAR[2] hat'	'"ear(1)" NEAR[2] "hat(1)"'
	'"ear a hat"'	'"ear(1) hat(3)"'
	'"cat hat"'	'"cat(1) hat(2)"'
	'cat NEAR[1] hat'	'"cat(1)" NEAR[1] "hat(1)"'
	'ear NEAR[1] hat'	'"ear(1)" NEAR[1] "hat(1)"'
	'"ear hat"'	'"ear(1) hat(2)"'
	'"wear a a hat"'	'"wea(1) ear(2) hat(5)"'

另请参见

- “sa_char_terms 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》
- “sa_nchar_terms 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》

文本索引

在执行全文搜索时，搜索的是**文本索引**，而不是表中的行。因此，必须先在要搜索的列上创建文本索引，之后才能执行全文搜索。文本索引在索引列中存储术语的位置信息。使用文本索引的查询会比必须扫描表中所有值的查询要快。

创建文本索引时，可以指定创建和刷新文本索引时使用的**文本配置对象**。文本配置对象包含影响索引构建方式的设置。如果不指定文本配置对象，数据库服务器将使用缺省配置对象。请参见“文本配置对象”一节第 285 页。

也可以指定文本索引的**刷新类型**。刷新类型定义了刷新文本索引的频率。文本索引刷新的时间越近，返回的结果就越精确。但是，刷新需要时间，而且会降低性能。例如，如果将文本索引配置为在每次更改基础数据时都进行刷新，则索引表的频繁更新可能会影响性能。请参见“文本索引刷新类型”一节第 298 页。

要查看现有文本索引的设置，请使用 sa_text_index_stats 系统过程。请参见“sa_text_index_stats 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》。

文本索引刷新类型

在创建文本索引时，还必须选择一种**刷新类型**。支持的文本索引刷新类型有三种：立即、自动和手动。在创建时定义文本索引的刷新类型。可以在创建文本索引后更改刷新类型，但快速文本索引除外。

有关如何设置刷新类型的信息，请参见“CREATE TEXT INDEX 语句”一节 《SQL Anywhere 服务器 - SQL 参考》和“ALTER TEXT INDEX 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

- **立即刷新（缺省值）** 立即刷新文本索引在基础表中的数据发生更改时进行刷新，仅当数据必须保持最新或索引列相对较短时才建议使用。

  文本索引的缺省刷新类型是立即刷新。

  无法将自动或手动刷新文本索引更改为立即刷新文本索引。相反，必须先将其删除，然后重新创建为立即刷新文本索引。

  立即刷新文本索引支持所有隔离级别。它们会在创建时填充，并在此初始刷新过程中保持该表的独占锁。

- **自动刷新** 自动刷新文本索引会按照所指定的时间间隔自动进行刷新，因此建议在可以接受某些数据过时的情况下使用。过时索引的查询返回自上次刷新后未更改的匹配行。因此，查询不会返回自上次刷新后所插入、删除或更新的行。

  如果满足以下任一条件，则自动刷新文本索引的刷新频率也可能要比所指定的间隔更频繁：距离上次刷新的时间大于刷新间隔，或者所有待执行行的总长度（sa_text_index_stats 系统过程返回的 pending_length）超过总索引大小（sa_text_index_stats 返回的 doc_length）的 20%。

  自动刷新文本索引使用隔离级别 0 进行刷新。

  自动刷新文本索引在创建时不包含数据，因此仅在首次刷新后才可用（首次刷新通常在创建文本索引后的一分钟之内进行）。也可以使用 REFRESH TEXT INDEX 语句手动刷新自动刷新文本索引。

自动刷新文本索引在重装过程中不会进行刷新，除非为 dbunload 指定 -g 选项。请参见“卸载实用程序 (dbunload)”一节 《SQL Anywhere 服务器 - 数据库管理》。

- **手动刷新** 手动刷新文本索引仅当您刷新它们时才会刷新，因此建议在以下情况下使用：基础表中的数据很少更改、可接受较大程度的数据过时，或者要在某个事件后或满足某个条件后进行刷新。过时索引的查询返回自上次刷新后未更改的匹配行。因此，查询不会返回自上次刷新后所插入、删除或更新的行。

  您可以定义自己的策略以刷新手动刷新文本索引。例如，可以使用过程来刷新所有手动刷新文本索引，该过程使用以参数形式传递的刷新间隔，以及与用于自动刷新文本索引的规则相类似的规则。

  在以下示例中，替换 *text-index-name*、*table-owner* 和 *table-name*。

```
CREATE PROCEDURE refresh_manual_text_indexes(
   refresh_interval UNSIGNED INT )
BEGIN
 FOR lp1 AS c1 CURSOR FOR
   SELECT ts.*
   FROM SYS.SYSTEXTIDX ti JOIN sa_text_index_stats( ) ts
   ON ( ts.index_id = ti.index_id )
   WHERE ti.refresh_type = 1 -- manual refresh indexes only
 DO
   BEGIN
    IF last_refresh IS null
    OR cast(pending_length as float) / (
       IF doc_length=0 THEN NULL ELSE doc_length ENDIF) > 0.2
    OR DATEDIFF( MINUTE, CURRENT TIMESTAMP, last_refresh )
       > refresh_interval THEN
     EXECUTE IMMEDIATE 'REFRESH TEXT INDEX ' || text-index-name || ' ON "'
     || table-owner || '"."' || table-name || '"';
    END IF;
   END;
  END FOR;
END;
```

  您可以随时使用 sa_text_index_stats 系统过程决定是否需要刷新，以及刷新应该是完整重建还是增量更新。请参见“sa_text_index_stats 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》。

  手动刷新文本索引在创建时不包含数据，仅当对其进行刷新后才可用。要刷新手动刷新文本索引，请使用 REFRESH TEXT INDEX 语句。

  手动刷新文本索引在重装过程中不会进行刷新，除非为 dbunload 指定 -g 选项。请参见“卸载实用程序 (dbunload)”一节 《SQL Anywhere 服务器 - 数据库管理》。

另请参见

- “全文搜索”一节第 284 页
- “创建文本索引”一节第 300 页
- “文本配置对象设置”一节第 285 页
- “sa_text_index_stats 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》
- “REFRESH TEXT INDEX 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “isolation_level 选项 [数据库] [兼容性]”一节 《SQL Anywhere 服务器 - 数据库管理》

创建文本索引

可以在任何类型的列上创建文本索引。类型不是 VARCHAR 或 NVARCHAR 的列会在建立索引时转换为字符串。请参见“数据类型转换”一节 《SQL Anywhere 服务器 - SQL 参考》。

文本索引会占用磁盘空间，并需要刷新。应仅在需要用来支持查询的列上创建文本索引。

不能在实例化视图、常规视图或临时表上创建文本索引。

不要创建多个引用某一列的文本索引，因为这样可能会返回意外的结果。

♦ 创建文本索引 (Sybase Central)

1. 以具有 DBA 权限的用户身份，或以要创建文本索引的表的所有者身份连接到数据库。
2. 单击 [**文本索引**] 选项卡。
3. 选择 [**文件**] » [**新建**] » [**文本索引**]。
4. 按照 [**创建索引向导**] 中的说明操作。

   新文本索引随即出现在 [**文本索引**] 选项卡上。它还会出现在 [**文本索引**] 文件夹中。
5. 如果创建立即刷新文本索引，将自动向其中填充数据。对于其它刷新类型，必须右键单击文本索引并选择 [**刷新数据**] 来刷新该文本索引。

♦ 创建新文本索引 (SQL)

1. 以具有 DBA 权限的用户身份，或以要创建文本索引的表的所有者身份连接到数据库。
2. 执行 CREATE TEXT INDEX 语句。请参见“CREATE TEXT INDEX 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。
3. 如果创建立即刷新文本索引，将自动向其中填充数据。对于其它刷新类型，必须通过执行 REFRESH TEXT INDEX 语句来刷新文本索引。请参见“REFRESH TEXT INDEX 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

另请参见

- “文本索引刷新类型”一节第 298 页
- “全文搜索”一节第 284 页

刷新文本索引

只能刷新定义为 AUTO REFRESH 和 MANUAL REFRESH 的文本索引。

♦ 刷新文本索引 (Sybase Central)

1. 以具有 DBA 权限的用户身份，或以要构建文本索引的表的所有者身份连接到数据库。
2. 在左窗格中，单击 [**文本索引**]。
3. 右击文本索引，然后选择 [**刷新数据**]。

4. 为刷新选择隔离级别，然后单击 [**确定**]。

♦ **刷新文本索引 (SQL)**

1. 以具有 DBA 权限的用户身份，或以要构建文本索引的表的所有者身份连接到数据库。
2. 执行 REFRESH TEXT INDEX 语句。

   例如，以下语句刷新 demo 数据库中 Products 表的 Description 列上名为 txt_index_manual 的文本索引：

```
REFRESH TEXT INDEX txt_index_manual ON Products
   WITH ISOLATION LEVEL 0;
```

   请参见“REFRESH TEXT INDEX 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

另请参见

- “文本索引刷新类型”一节第 298 页
- “全文搜索”一节第 284 页

更改文本索引概述

可以变更文本索引的以下特性：

- **刷新类型** 可以将刷新类型从 AUTO REFRESH 更改为 MANUAL REFRESH，反之亦然。使用 ALTER TEXT INDEX 语句的 REFRESH 子句更改刷新类型。请参见“ALTER TEXT INDEX 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

  不能将文本索引更改为 IMMEDIATE REFRESH，也不能从 IMMEDIATE REFRESH 更改为其它类型；要进行此更改，必须删除该文本索引，然后重新创建它。请参见“DROP TEXT INDEX 语句”一节 《SQL Anywhere 服务器 - SQL 参考》和“CREATE TEXT INDEX 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

- **名称** 可以使用 ALTER TEXT INDEX 语句的 RENAME 子句对文本索引重命名。请参见“ALTER TEXT INDEX 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

- **内容** 除列列表之外，控制为哪个目标建立索引的设置存储在文本配置对象中。如果要更改为其建立索引的目标，则变更文本索引所引用的文本配置对象。必须在能够变更文本配置对象之前截断相关文本索引，在变更文本配置对象之后刷新文本索引。对于立即刷新文本索引，必须先删除文本索引，然后在更改文本配置对象之后再重新创建该文本索引。

  请参见：

  - “文本配置对象”一节第 285 页
  - “TRUNCATE TEXT INDEX 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
  - “ALTER TEXT INDEX 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
  - “REFRESH TEXT INDEX 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
  - “sa_refresh_text_indexes 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》

不能变更文本索引来引用其它文本配置对象。如果希望让文本索引引用另一个文本配置对象，则删除该文本索引，然后通过指定新文本配置对象重新创建它。

变更文本索引

可以更改文本索引的名称或更改其刷新类型。

♦ 变更文本索引的刷新类型 (Sybase Central)

1. 以具有 DBA 权限的用户身份，或以要构建文本索引的表的所有者身份连接到数据库。
2. 在左窗格中，单击 [**文本索引**]。
3. 右击文本索引，然后选择 [**属性**]。
4. 编辑文本索引属性，并单击 [**确定**]。

♦ 变更文本索引的刷新类型 (SQL)

1. 以具有 DBA 权限的用户身份或以文本索引所有者身份连接到数据库。
2. 执行 ALTER TEXT INDEX 语句。请参见“ALTER TEXT INDEX 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

♦ 重命名文本索引 (Sybase Central)

1. 以具有 DBA 权限的用户身份，或以要构建文本索引的表的所有者身份连接到数据库。
2. 在左窗格中，单击 [**文本索引**]。
3. 右击文本索引，然后选择 [**属性**]。
4. 单击 [**常规**] 选项卡，然后为文本索引键入新名称。
5. 单击 [**确定**]。

♦ 重命名文本索引 (SQL)

1. 以具有 DBA 权限的用户身份或以文本索引所有者身份连接到数据库。
2. 执行 ALTER TEXT INDEX 语句。请参见“ALTER TEXT INDEX 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

另请参见

- “文本索引”一节第 298 页
- “全文搜索”一节第 284 页

查看数据库中的文本索引

可以通过 Sybase Central 或 SQL 语句来查看有关数据库中文本索引的信息。

♦ 查看数据库中的文本索引 (Sybase Central)

1. 以具有 DBA 权限的用户身份或以文本索引所有者身份连接到数据库。
2. 在左窗格中，单击 [**文本索引**]。
3. 要查看文本索引中的术语，在左窗格中双击文本索引，然后在右窗格中选择 [**词汇**] 选项卡。
4. 要查看文本索引设置（如刷新类型或该索引所引用的文本配置对象），右键单击文本索引，然后选择 [**属性**]。

♦ 查看数据库中的文本索引 (SQL)

1. 以具有 DBA 权限的用户身份或以文本索引所有者身份连接到数据库。
2. 调用 sa_text_index_stats 系统过程，如下所示：

```
CALL sa_text_index_stats( );
```

另请参见

- “sa_text_index_stats 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》

全文搜索的类型

使用全文搜索，可以搜索**术语**、**短语**（术语序列）或**前缀**。还可以将多个术语、短语或前缀合并成布尔表达式，或者要求通过邻近搜索使表达式出现在彼此附近。

通过在 SELECT 语句的 FROM 子句中使用 CONTAINS 子句，或者在 WHERE 子句中使用 CONTAINS 子句，可以执行全文搜索。还可以将全文搜索作为 IF 搜索条件的一部分执行（例如，`SELECT IF CONTAINS...`）。

术语和短语搜索

执行术语列表的全文搜索时，除非多个术语处于一个短语之中，否则术语的顺序不重要。如果将术语放在一个短语之中，数据库服务器将严格按照指定术语时的相同顺序和相对位置来查找这些术语。

执行术语或短语搜索时，如果术语因为超出术语长度设置或位于非索引字表之中而从查询中删除，则返回的行数可能与预期不符。这是因为从查询中删除术语相当于更改搜索条件。例如，如果搜索短语 [`'"grown cotton"'`] 并且 grown 位于非索引字表中，将得到所有包含 cotton 的索引行。

只要视作 CONTAINS 子句语法关键字的术语在短语之中，即可搜索它们。

术语搜索

在 demo 数据库中，MarketingInformation 表的 Description 列中创建了名为 MarketingTextIndex 的文本索引。以下语句查询 MarketingInformation.Description 列，并返回 Description 列中包含术语 **cotton** 的值所在的行。

```
SELECT ID, Description
   FROM MarketingInformation
   WHERE CONTAINS ( Description, 'cotton' );
```

ID	Description
906	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Visor</title></head><body lang=EN-US> Lightweight 100% organically grown cotton construction. Shields against sun and precipitation.cotton Metallic ions in the fibers inhibit bacterial growth, and help neutralize odor. </body></html>
908	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Sweatshirt</title></head><body lang=EN-US> Lightweight 100% organically grown cotton hooded sweatshirt with taped neck seams. Comes pre-washed for softness and to lessen shrinkage. </body></html>

ID	Description
909	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Sweatshirt</title></head><body lang=EN-US> Top-notch construction includes durable topstitched seams for strength with low-bulk, resilient rib-knit collar, cuffs and bottom. An 80% cotton/20% polyester blend makes it easy to keep them clean. </body></html>
910	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Shorts</title></head><body lang=EN-US> These quick-drying cotton shorts provide all day comfort on or off the trails. Now with a more comfortable and stretchy fabric and an adjustable drawstring waist. </body></html>

以下示例查询 MarketingInformation 表，并为每一行返回一个指示 Description 列中的值是否包含术语 **cotton** 的值。

```
SELECT ID, IF CONTAINS ( Description, 'cotton' )
      THEN 1
      ELSE 0
      ENDIF AS Results
   FROM MarketingInformation;
```

ID	Results
901	0
902	0
903	0
904	0
905	0
906	1
907	0
908	1
909	1
910	1

下一示例在 MarketingInformation 表中查询 Description 列中含有术语 **cotton** 的项目，然后显示各个匹配结果的分数。

```
SELECT ID, ct.score, Description
   FROM MarketingInformation CONTAINS ( MarketingInformation.Description,
'cotton' ) as ct
   ORDER BY ct.score DESC;
```

ID	score	Description
908	0.9461597363521859	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Sweatshirt</title></head><body lang=EN-US> Lightweight 100% organically grown cotton hooded sweatshirt with taped neck seams. Comes pre-washed for softness and to lessen shrinkage. </body></html>
910	0.9244136988525732	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Shorts</title></head><body lang=EN-US> These quick-drying cotton shorts provide all day comfort on or off the trails. Now with a more comfortable and stretchy fabric and an adjustable drawstring waist. </body></html>
906	0.9134171046194403	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Visor</title></head><body lang=EN-US> Lightweight 100% organically grown cotton construction. Shields against sun and precipitation. Metallic ions in the fibers inhibit bacterial growth, and help neutralize odor. </body></html>
909	0.8856420222728282	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Sweatshirt</title></head><body lang=EN-US> Top-notch construction includes durable topstitched seams for strength with low-bulk, resilient rib-knit collar, cuffs and bottom. An 80% cotton/20% polyester blend makes it easy to keep them clean. </body></html>

有关在查询的 FROM 子句中使用 CONTAINS 时对结果进行计分的详细信息，请参见“对全文搜索结果进行计分”一节第 316 页。

短语搜索

对短语执行全文搜索时，将该短语括在双引号中。如果某一列包含具有指定顺序和相对位置的术语，则该列是匹配的。

不能指定 CONTAINS 关键字（如 AND 或 FUZZY ）作为要搜索的术语，除非将它们放在短语之内（允许只包含一个术语的短语）。例如，即使 NOT 是 CONTAINS 关键字，以下语句也是可接受的。有关 CONTAINS 关键字和特殊字符的列表，请参见“CONTAINS 搜索条件”一节 《SQL Anywhere 服务器 - SQL 参考》。

```
SELECT * FROM table-name CONTAINS ( Remarks, '"NOT"' );
```

如果特殊字符位于短语之中，则不会将其解释为特殊字符，但星号例外。

短语不能用作邻近搜索的参数。

以下语句在 MarketingInformation.Description 中查询短语 ["grown cotton"]，然后显示各个匹配结果的分数：

```
SELECT ID, ct.score, Description
   FROM MarketingInformation CONTAINS ( MarketingInformation.Description,
'"grown cotton"' ) as ct
   ORDER BY ct.score DESC;
```

ID	score	Description
908	1.6619019465461564	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Sweatshirt</title></head><body lang=EN-US> Lightweight 100% organically grown cotton hooded sweatshirt with taped neck seams. Comes pre-washed for softness and to lessen shrinkage. </body></html>
906	1.6043904700786786	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Visor</title></head><body lang=EN-US> Lightweight 100% organically grown cotton construction. Shields against sun and precipitation. Metallic ions in the fibers inhibit bacterial growth, and help neutralize odor. </body></html>

前缀搜索

全文搜索功能允许搜索某一术语的开头部分。这称为**前缀搜索**。要执行前缀搜索，应指定要搜索的前缀，后跟一个星号。这称为**前缀术语**。

CONTAINS 子句的关键字不能用于前缀搜索，除非它们位于短语之中。有关 CONTAINS 关键字的列表，请参见“CONTAINS 搜索条件”一节 《SQL Anywhere 服务器 - SQL 参考》。

也可以在查询字符串中指定多个前缀术语，并将其包括在短语中（例如 `['"shi* fab"']`。）

有关前缀搜索的完整语法限制，请参见“对于星号 (*) 允许的语法”一节 《SQL Anywhere 服务器 - SQL 参考》。

以下示例在 MarketingInformation 表中查询以前缀 **shi** 开头的项目：

```
SELECT ID, ct.score, Description
   FROM MarketingInformation CONTAINS ( MarketingInformation.Description,
'shi*' ) AS ct
   ORDER BY ct.score DESC;
```

ID	score	Description
906	2.295363835537917	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Visor</title></head><body lang=EN-US> Lightweight 100% organically grown cotton construction. Shields against sun and precipitation. Metallic ions in the fibers inhibit bacterial growth, and help neutralize odor. </body></html>
901	1.6883275743936228	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Tee Shirt</title></head><body lang=EN-US> We've improved the design of this perennial favorite. A sleek and technical shirt built for the trail, track, or sidewalk. UPF rating of 50+. </body></html>

ID	score	Description
903	1.6336529491832605	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Tee Shirt</title></head><body lang=EN-US> A sporty, casual shirt made of recycled water bottles. It will serve you equally well on trails or around town. The fabric has a wicking finish to pull perspiration away from your skin. </body></html>
902	1.6181703448678983	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Tee Shirt</title></head><body lang=EN-US> This simple, sleek, and lightweight technical shirt is designed for high-intensity workouts in hot and humid weather. The recycled polyester fabric is gentle on the earth and soft against your skin. </body></html>

ID 906 的分数最高，因为术语 shield 在文本索引中的出现频率低于 shirt。

对 GENERIC 文本索引执行前缀搜索

对于 GENERIC 文本索引，前缀搜索的行为如下所示：

- 如果前缀术语的长度大于 MAXIMUM TERM LENGTH，则将其从查询字符串中删除，因为文本索引中不能存在长度超过 MAXIMUM TERM LENGTH 的术语。因此，对于 MAXIMUM TERM LENGTH 为 3 的文本索引，搜索 ['red appl*'] 相当于搜索 ['red']。
- 如果前缀术语的长度小于 MINIMUM TERM LENGTH，并且该术语不是短语搜索的一部分，则前缀搜索将正常进行。因此，对于 MINIMUM TERM LENGTH 为 5 的 GENERIC 文本索引，搜索 ['macintosh a*'] 将返回包含 macintosh 以及任何以 a 开头且长度大于等于 5 的术语的索引行。
- 如果前缀术语的长度小于 MINIMUM TERM LENGTH，但术语是短语搜索的一部分，则前缀术语将从查询中删除。因此，对于 MINIMUM TERM LENGTH 为 5 的 GENERIC 文本索引，搜索 ['"macintosh appl* turnover"'] 相当于搜索其后为任何后跟 turnover 的术语的 macintosh。不会查找包含 ["macintosh turnover"] 的行；macintosh 和 turnover 之间必须存在术语。

对 NGRAM 文本索引执行前缀搜索

对于 NGRAM 文本索引，前缀搜索可能返回意外的结果，因为 NGRAM 文本索引仅包含 n 元语法词，而不包含任何有关术语开头的信息。查询术语也将分解为 n 元语法词，然后使用 n 元语法词而不是查询术语执行搜索。因此，应注意以下行为：

- 如果前缀术语的长度小于 n 元语法词的长度 (MAXIMUM TERM LENGTH)，则查询将返回所有包含以该前缀术语开头的 n 元语法词的索引行。例如，对于 3 元语法词文本索引，搜索 ['ea*'] 将返回所有包含以 ea 开头的 n 元语法词的索引行。因此，如果对术语 weather 和 fear 建立索引，则行将被视为匹配，因为它们各自的 n 元语法词分别包含 eat 和 ear。
- 如果前缀术语的长度大于 n 元语法词长度，并且不是短语的一部分，也不是邻近搜索中的参数，则前缀术语将转换为 n 元语法词短语，并将星号删除。例如，对于 3 元语法词文本索引，搜索 ['purple blac*'] 相当于搜索 ['"pur urp rpl ple" AND "bla lac"']。
- 如果是短语，那么还会发生以下行为：
  - 如果前缀术语是短语中的唯一术语，则将其转换为 n 元语法词短语，并删除星号。例如，对于 3 元语法词文本索引，搜索 ['"purpl*"'] 相当于搜索 ['"pur urp rpl"']。
  - 如果前缀术语在短语中处于最后位置，则删除星号并将术语转换为 n 元语法词短语。例如，对于 3 元语法词文本索引，搜索 ['"purple blac*"'] 相当于搜索 ['"pur urp rpl ple bla lac"']。
  - 如果前缀术语不在短语的最后位置，则短语分解为由 AND 连接的多个短语。例如，对于 3 元语法词文本索引，搜索 ['"purp* blac*"'] 相当于搜索 ['"pur urp" AND "bla lac"']。
- 如果前缀术语是邻近搜索中的参数，则邻近搜索转换为 AND。例如，对于 3 元语法词文本索引，搜索 ['red NEAR[1] appl*'] 相当于搜索 ['red AND "app ppl"']。

另请参见

- “文本索引”一节第 298 页
- “CONTAINS 搜索条件”一节 《SQL Anywhere 服务器 - SQL 参考》

邻近搜索

全文搜索功能允许搜索某一列中相互邻近的多个术语。这称为**邻近搜索**。要执行邻近搜索，应指定关键字 NEAR 或代字号 (~) 在其中间的两个术语。

对于 NEAR 关键字，可以指定一个整型变量来定义最大距离。例如，*term1* NEAR[5] *term2* 查找位于 *term2* 的五个术语范围内的 *term1* 的实例。术语的顺序并不重要；'term1 NEAR term2' 等效于 'term2 NEAR term1'。

如果不指定距离，数据库服务器将使用 10 作为缺省距离。

还可以指定代字号 (~) 取代 NEAR 关键字。例如，'term1 ~ term2'。但是，在使用代字号格式时无法指定距离；此时应用十个术语的缺省值。

不能将短语指定为邻近搜索中的参数。

在邻近搜索中，如果指定前缀术语作为参数，则邻近搜索转换为 AND 表达式。例如，对于 3 元语法词文本索引，搜索 ['red NEAR[1] appl*'] 相当于搜索 ['red AND "app ppl"']。由于这不再是邻近搜索，所以在 CONTAINS 子句中指定了多列的情况下搜索不再限于单一列。

示例

假定要在 MarketingInformation.Description 中搜索含有术语 skin 的 10 个术语中的术语 fabric。此时可以执行以下语句。

```
SELECT ID, "contains".score, Description
   FROM MarketingInformation CONTAINS ( Description, 'fabric ~ skin' );
```

ID	score	Description
902	1.5572371866083279	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Tee Shirt</title></head><body lang=EN-US> This simple, sleek, and lightweight technical shirt is designed for high-intensity workouts in hot and humid weather. The recycled polyester fabric is gentle on the earth and soft against your skin. </body></html>

由于缺省距离即为 10 个术语，因此不必指定距离。但如果将距离增加一个术语，将返回另一行：

```
SELECT ID, "contains".score, Description
   FROM MarketingInformation CONTAINS ( Description, 'fabric NEAR[11] skin' );
```

ID	score	Description
903	1.5787803210404958	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Tee Shirt</title></head><body lang=EN-US> A sporty, casual shirt made of recycled water bottles. It will serve you equally well on trails or around town. The fabric has a wicking finish to pull perspiration away from your skin. </body></html>
902	2.163125855043747	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Tee Shirt</title></head><body lang=EN-US> This simple, sleek, and lightweight technical shirt is designed for high-intensity workouts in hot and humid weather. The recycled polyester fabric is gentle on the earth and soft against your skin. </body></html>

ID 903 的分数较高，因为其术语间的距离较近。

布尔搜索

在执行全文搜索时，可以指定由布尔运算符分隔的多个术语。在执行全文搜索时，SQL Anywhere 支持以下布尔运算符：AND、OR 和 AND NOT。

有关布尔搜索语法的详细信息，请参见“CONTAINS 搜索条件”一节 《SQL Anywhere 服务器 - SQL 参考》。

在全文搜索中使用 AND 运算符

使用 AND 运算符得到的匹配行是包含在 AND 两侧指定的两个术语的行。对于 AND 运算符，还可以使用和符号 (&)。如果指定的术语之间没有任何运算符，则暗指采用 AND 运算符。

术语的列出顺序并不重要。

例如，以下各个语句都可在 MarketingInformation.Description 中查找包含术语 **fabric** 和以 **ski** 开头的术语的行：

```
SELECT *
   FROM MarketingInformation
   WHERE CONTAINS ( MarketingInformation.Description, 'ski* AND fabric' );
SELECT *
   FROM MarketingInformation
   WHERE CONTAINS ( MarketingInformation.Description, 'fabric & ski*' );
SELECT *
   FROM MarketingInformation
   WHERE CONTAINS ( MarketingInformation.Description, 'ski* fabric' );
```

在全文搜索中使用 OR 运算符

使用 OR 运算符得到的匹配行是至少包含在 OR 两侧指定的任一搜索术语的行。对于 OR 运算符，还可以使用竖线 (|)；两者是等同的。

术语的列出顺序并不重要。

例如，以下任一语句都可返回 MarketingInformation.Description 中包含术语 **fabric** 或以 **ski** 开头的术语的行：

```
SELECT *
   FROM MarketingInformation
   WHERE CONTAINS ( MarketingInformation.Description, 'ski* OR fabric' );
SELECT *
   FROM MarketingInformation
   WHERE CONTAINS ( MarketingInformation.Description, 'fabric | ski*' );
```

在全文搜索中使用 AND NOT 运算符

使用 AND NOT 运算符查找的结果匹配左侧参数但不匹配右侧参数。对于 AND NOT 运算符，还可以使用连字符 (-)；两者是等同的。

例如，以下语句互相等效，都返回包含术语 **fabric** 但不包含任何以 **ski** 开头的术语的行。

```
SELECT *
   FROM MarketingInformation
   WHERE CONTAINS ( MarketingInformation.Description, 'fabric AND NOT
ski*' );
SELECT *
   FROM MarketingInformation
   WHERE CONTAINS ( MarketingInformation.Description, 'fabric -ski*' );
```

```
SELECT *
   FROM MarketingInformation
   WHERE CONTAINS ( MarketingInformation.Description, 'fabric & -ski*' );
```

组合不同的布尔运算符

可在查询字符串中组合布尔运算符。例如，以下语句相互等效，都在 MarketingInformation.Description 列中搜索包含 **fabric** 和 **skin**，但不包含 **cotton** 的项目：

```
SELECT *
   FROM MarketingInformation
   WHERE CONTAINS ( MarketingInformation.Description, 'skin fabric -
cotton' );
SELECT *
   FROM MarketingInformation
   WHERE CONTAINS ( MarketingInformation.Description, 'fabric -cotton AND
skin' );
```

以下语句相互等效，都在 MarketingInformation.Description 列中搜索包含 **fabric**，或同时包含 **cotton** 和 **skin** 的项目：

```
SELECT *
   FROM MarketingInformation
   WHERE CONTAINS ( MarketingInformation.Description, 'fabric | cotton AND
skin' );
SELECT *
   FROM MarketingInformation
   WHERE CONTAINS ( MarketingInformation.Description, 'cotton skin OR
fabric' );
```

对术语和短语分组

可以使用括号对术语和表达式分组。例如，以下语句在 MarketingInformation.Description 列中搜索包含 **cotton** 或 **fabric**，并包含以 **ski** 开头的术语的项目。

```
SELECT ID, Description FROM MarketingInformation
  WHERE CONTAINS( MarketingInformation.Description, '( cotton OR fabric ) AND
shi*' );
```

	Description
902	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Tee Shirt</title></head><body lang=EN-US> This simple, sleek, and lightweight technical shirt is designed for high-intensity workouts in hot and humid weather. The recycled polyester fabric is gentle on the earth and soft against your skin. </body></html>

	Description
903	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Tee Shirt</title></head><body lang=EN-US> A sporty, casual shirt made of recycled water bottles. It will serve you equally well on trails or around town. The fabric has a wicking finish to pull perspiration away from your skin. </body></html>
906	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Visor</title></head><body lang=EN-US> Lightweight 100% organically grown cotton construction. Shields against sun and precipitation. Metallic ions in the fibers inhibit bacterial growth, and help neutralize odor. </body></html>

在多列间执行搜索

可以通过单一查询在多个列之间执行全文搜索，前提是这些列是同一文本索引的一部分。

```
SELECT *
   FROM Products
      WHERE CONTAINS ( t.c1, t.c2, 'term1|term2' );

SELECT *
   FROM t
   WHERE CONTAINS( t.c1, 'term1' )
      OR CONTAINS( t.c2, 'term2' );
```

如果 *t1.c1* 包含 *term1*，或 *t1.c2* 包含 *term2*，则第一个查询匹配。

如果 *t1.c1* 或 *t1.c2* 包含 *term1* 和 *term2* 中的任一个，则第二个查询匹配。以这种方式使用 CONTAINS 子句也会返回匹配分数。请参见“对全文搜索结果进行计分”一节第 316 页。

模糊搜索

模糊搜索可用于搜索拼写错误的单词或单词的变体。为此，可在带双引号的字符串前使用 FUZZY 运算符，来查找该字符串的近似匹配。例如，`CONTAINS ( Products.Description, 'FUZZY "cotton"' )` 将返回 **cotton** 及其拼写错误的形式，如 **coton** 或 **cotten**。

注意
只能对使用 NGRAM 术语断开器构建的文本索引执行模糊搜索。有关 NGRAM 术语断开器以及如何将其应用于模糊搜索的详细信息，请参见“文本配置对象设置”一节第 285 页。

使用 FUZZY 运算符相当于手动将字符串分解成长度为 *n* 的子字符串并使用 OR 运算符将它们分隔开。例如，假定有一个配置有 NGRAM 术语断开器且最大术语长度为 3 的文本索引。则指定

'FUZZY "500 main street"' 相当于指定 '500 OR mai OR ain OR str OR tre OR ree OR eet'。

FUZZY 运算符在返回分数的全文搜索中很有用。这是因为可能会返回许多近似匹配项，但通常只有分数最高的匹配项才有意义。

从索引中删除的术语

文本索引根据为用来创建文本索引的文本配置对象而定义的设置进行构建。当以下一个或多个条件成立时，术语不会在文本索引中出现：

- 术语包括在非索引字表中
- 术语的长度小于最小术语长度（仅限 GENERIC）
- 术语长度比最大术语长度长

此规则也适用于查询字符串。删除的术语可以匹配短语末尾或开头处的零个或多个术语。例如，假设术语 ['the'] 在非索引字表中：

- 如果该术语出现在 AND、OR 或 NEAR 的任意一侧，则同时删除该运算符和该术语。例如，搜索 'the AND apple'、'the OR apple' 或 'the NEAR apple' 相当于搜索 'apple'。
- 如果该术语出现在 AND NOT 的右侧，则 AND NOT 和该术语均被删除。例如，搜索 'apple AND NOT the' 相当于搜索 'apple'。

  如果该术语出现在 AND NOT 的左侧，则删除整个表达式，并且不会返回任何行。例如，['orange and the AND NOT apple' = 'orange']
- 如果该术语出现在短语中，则允许该短语与出现在该删除的术语位置处的任何术语匹配。例如，搜索 'feed the dog' 匹配 'feed the dog'、'feed my dog'、'feed any dog' 等。

如果想要搜索的术语都不在文本索引中，则不会返回任何行。例如，假设 'the' 和 'a' 都在非索引字表中。则搜索 'a OR the' 不会返回任何行。

另请参见

- “文本配置对象”一节第 285 页

查看搜索

要在全文搜索中使用视图，必须首先在基表中的所需列上构建文本索引。例如，假定您在 Employees.Address 列上创建一个名为 EmployeeAddressTxtIdx 的文本索引。随后在 Employees 表上创建一个名为 MyEmployeesView 的视图。通过使用一个与下面类似的语句，您可以在基础表上用文本索引来查询视图。

```
SELECT COUNT(*) FROM MyEmployeesView WHERE CONTAINS( EmployeeAddressTxtIdx,
'Avenue' );
```

在基础性基表上使用文本索引搜索视图有如下限制：

- 视图不能含有 TOP、FIRST、DISTINCT、GROUP BY、ORDER BY、UNION、INTERSECT 和 EXCEPT 子句或窗口函数。
- CONTAINS 查询能查询视图内的基表，但不能另一视图内的视图内的基表。

对全文搜索结果进行计分

在查询的 FROM 子句中使用 CONTAINS 子句时，各个匹配都会有与之相关联的分数。分数表明了匹配的近似程度，因此可以使用分数信息来排序数据。

计分过程基于两个主要标准：

- **术语在索引行中出现的次数** 术语在索引行中出现的次数越多，其分数就越高。
- **术语在文本索引中出现的次数** 术语在文本索引中出现的次数越多，其分数就越低。在 Sybase Central 中，可通过查看文本索引的 [**词汇**] 选项卡来了解术语在文本索引中的出现次数。选择 **term** 列可按字母顺序对术语进行排序。查看 **freq** 列可了解术语在文本索引中出现的次数。

根据全文搜索类型的不同，其它标准也可能影响计分过程。例如，在邻近搜索中，搜索术语的接近程度会影响计分过程。

如何使用分数

缺省情况下，CONTAINS 子句的结果集具有相关名 **contains**，其中包含一个名为 **score** 的列。可以引用 SELECT 列表、ORDER BY 子句或查询的其它部分中的 `"contains".score`。但是，由于 contains 是 SQL 保留字，因此一定要将其放在双引号之中。也可以指定另一相关名（例如 [`CONTAINS ( expression ) AS ct`]）。在全文搜索的文档示例中，分数列引用为 `ct.score`。

以下语句在 MarketingInformation.Description 中搜索以 **stretch** 开头或以 **comfort** 开头的术语：

```
SELECT ID, ct.score, Description
    FROM MarketingInformation CONTAINS ( MarketingInformation.Description,
'stretch* | comfort*' ) AS ct
    ORDER BY ct.score DESC;
```


ID	score	Description
910	5.570408968026068	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Shorts</title></head><body lang=EN-US> These quick-drying cotton shorts provide all day comfort on or off the trails. Now with a more comfortable and stretchy fabric and an adjustable drawstring waist. </body></html>
907	3.658418186470189	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Visor</title></head><body lang=EN-US> A polycarbonate visor with an abrasion-resistant coating on the outside. Great for jogging in the spring, summer, and early fall. The elastic headband has plenty of stretch to give you a snug yet comfortable fit every time you wear it. </body></html>

ID	score	Description
905	1.6750365447462499	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Baseball Cap</title></head><body lang=EN-US> A lightweight wool cap with mesh side vents for breathable comfort during aerobic activities. Moisture-absorbing headband liner. </body></html>

项目 910 的分数最高，因为它包含前缀术语 **comfort** 的两个实例，而其它项目仅包含一个实例。同样，项目 910 包含前缀术语 **stretch.** 的一个实例。

示例 2：搜索多列

以下示例说明了如何在多列之间执行全文搜索并对结果进行计分：

1. 在 Products 表上创建快速文本索引，如下所示：

```
CREATE TEXT INDEX scoringExampleMult
   ON Products ( Description, Name );
```

2. 对 Description 和 Name 列执行全文搜索，查找术语 **cap** 或 **visor**，如下所示。CONTAINS 子句的结果会被赋予相关名 ct，并在 SELECT 列表中引用，因此包括在结果集中。此外，ORDER BY 子句中也引用 ct.score 列，以按照分数对结果进行降序排序。

```
SELECT Products.Description, Products.Name, ct.score
    FROM Products CONTAINS ( Products.Description, Products.Name, 'cap OR
visor' ) ct
    ORDER BY ct.score DESC;
```

Description	Name	score
Cloth Visor	Visor	3.5635154905713042
Plastic Visor	Visor	3.4507856451176244
Wool cap	Baseball Cap	3.2340501745357333
Cotton Cap	Baseball Cap	3.090467108972918

计算多列搜索的分数，就好像将列值连接一起并作为单个值建立索引一样。但是，请注意，短语和 NEAR 运算符的匹配范围从来都不会跨越列边界，并且在多个列中出现的搜索术语会使分数增加，使其大于在单个连接值中的分数。

3. 为使本文档中的其它示例能够正确执行，必须删除在 Products 表上创建的文本索引。为此，请执行以下语句：

```
DROP TEXT INDEX scoringExampleMult ON Products;
```

教程：对 GENERIC 文本索引执行全文搜索

可通过以下过程对使用 GENERIC 术语断开器的文本索引执行全文搜索。

另请参见： “教程：执行模糊全文搜索”一节第 325 页.

♦ 对 GENERIC 文本索引执行全文搜索

1. 创建文本配置对象。

   以下示例创建一个名为 myTxtConfig 的文本配置对象。请记住，必须包括 FROM 子句，才能指定用作模板的文本配置对象。

```
CREATE TEXT CONFIGURATION myTxtConfig FROM default_char;
```

2. 自定义文本配置对象。

   添加包括 because、about、therefore 和 only 等单词的非索引字表。然后，将最大术语长度设置为 30。必须通过单独的 ALTER TEXT CONFIGURATION 语句来执行此操作，如下所示：

```
ALTER TEXT CONFIGURATION myTxtConfig
   STOPLIST 'because about therefore only';
ALTER TEXT CONFIGURATION myTxtConfig
   MAXIMUM TERM LENGTH 30;
```

3. 创建 MarketingInformation 表的副本。
   a. 在 Sybase Central 中，展开 [**表**] 文件夹。
   b. 右击 **MarketingInformation** 并选择 [**复制**]。
   c. 右击 [**表**] 文件夹，然后选择 [**粘贴**]。
   d. 在 [**名称**] 字段中键入 **MarketingInformation1**。单击 [**确定**]。
4. 在 Interactive SQL 中，执行以下命令以向新表中填充数据：

```
INSERT INTO MarketingInformation1
   SELECT * FROM MarketingInformation;
```

5. 在 demo 数据库中 MarketingInformation1 表的 Description 列上创建引用 myTxtConfig 文本配置对象的文本索引。将刷新间隔设置为 24 小时。

```
CREATE TEXT INDEX myTxtIndex ON MarketingInformation1 ( Description )
   CONFIGURATION myTxtConfig
   AUTO REFRESH EVERY 24 HOURS;
```

6. 执行以下语句来刷新文本索引：

```
REFRESH TEXT INDEX myTxtIndex ON MarketingInformation1;
```

7. 执行以下语句来测试文本索引。

   a. 此语句在文本索引中搜索术语 **cotton** 或 **cap**。结果按分数降序排序。**Cap** 比 **cotton** 的分数高，因为 **cap** 在文本索引中的出现频率较低。

```
SELECT ID, Description, ct.*
   FROM MarketingInformation1
     CONTAINS ( Description, 'cotton | cap' ) ct
   ORDER BY score DESC;
```

ID	Description	Score
905	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Baseball Cap</title></head><body lang=EN-US> A lightweight wool cap with mesh side vents for breathable comfort during aerobic activities. Moisture-absorbing headband liner. </body></html>	2.2742084275032632
904	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Baseball Cap</title></head><body lang=EN-US> This fashionable hat is ideal for glacier travel, sea-kayaking, and hiking. With concealed draw cord for windy days. </body></html>	1.6980426550094467
908	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Sweatshirt</title></head><body lang=EN-US> Lightweight 100% organically grown cotton hooded sweatshirt with taped neck seams. Comes pre-washed for softness and to lessen shrinkage. </body></html>	0.9461597363521859

ID	Description	Score
910	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Shorts</ title></head><body lang=EN-US> These quick-drying cotton shorts provide all day comfort on or off the trails. Now with a more comfortable and stretchy fabric and an adjustable drawstring waist. </body></html>	0.9244136988525732
906	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Visor</ title></head><body lang=EN-US> Lightweight 100% organically grown cotton construction. Shields against sun and precipitation. Metallic ions in the fibers inhibit bacterial growth, and help neutralize odor. </body></html>	0.9134171046194403
909	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Sweatshirt</ title></head><body lang=EN-US> Top-notch construction includes durable topstitched seams for strength with low-bulk, resilient rib-knit collar, cuffs and bottom. An 80% cotton/20% polyester blend makes it easy to keep them clean. </body></html>	0.8856420222728282

b. 查询 2：

```
SELECT ID, Description
   FROM MarketingInformation1
   WHERE CONTAINS( Description, 'cotton -visor' );
```

ID	Description
908	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Sweatshirt</title></head><body lang=EN-US> Lightweight 100% organically grown cotton hooded sweatshirt with taped neck seams. Comes pre-washed for softness and to lessen shrinkage. </body></html>
909	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Sweatshirt</title></head><body lang=EN-US> Top-notch construction includes durable topstitched seams for strength with low-bulk, resilient rib-knit collar, cuffs and bottom. An 80% cotton/20% polyester blend makes it easy to keep them clean. </body></html>
910	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Shorts</title></head><body lang=EN-US> These quick-drying cotton shorts provide all day comfort on or off the trails. Now with a more comfortable and stretchy fabric and an adjustable drawstring waist. </body></html>

c. 以下语句测试各行中是否存在术语 **cotton**。如果行包含该术语，则结果列中显示 1；否则返回 0。

```
SELECT ID, Description, IF CONTAINS ( Description, 'cotton' )
      THEN 1
      ELSE 0
      ENDIF AS Results
   FROM MarketingInformation1;
```

ID	Description	Results
901	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Tee Shirt</title></head><body lang=EN-US> We've improved the design of this perennial favorite. A sleek and technical shirt built for the trail, track, or sidewalk. UPF rating of 50+. </body></html>	0

ID	Description	Results
902	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Tee Shirt</title></head><body lang=EN-US> This simple, sleek, and lightweight technical shirt is designed for high-intensity workouts in hot and humid weather. The recycled polyester fabric is gentle on the earth and soft against your skin. </body></html>	0
903	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Tee Shirt</title></head><body lang=EN-US> A sporty, casual shirt made of recycled water bottles. It will serve you equally well on trails or around town. The fabric has a wicking finish to pull perspiration away from your skin. </body></html>	0
904	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Baseball Cap</title></head><body lang=EN-US> This fashionable hat is ideal for glacier travel, sea-kayaking, and hiking. With concealed draw cord for windy days. </body></html>	0
905	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Baseball Cap</title></head><body lang=EN-US> A lightweight wool cap with mesh side vents for breathable comfort during aerobic activities. Moisture-absorbing headband liner. </body></html>	0

ID	Description	Results
906	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Visor</title></head><body lang=EN-US> Lightweight 100% organically grown cotton construction. Shields against sun and precipitation. Metallic ions in the fibers inhibit bacterial growth, and help neutralize odor. </body></html>	1
907	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Visor</title></head><body lang=EN-US> A polycarbonate visor with an abrasion-resistant coating on the outside. Great for jogging in the spring, summer, and early fall. The elastic headband has plenty of stretch to give you a snug yet comfortable fit every time you wear it. </body></html>	0
908	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Sweatshirt</title></head><body lang=EN-US> Lightweight 100% organically grown cotton hooded sweatshirt with taped neck seams. Comes pre-washed for softness and to lessen shrinkage. </body></html>	1
909	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Sweatshirt</title></head><body lang=EN-US> Top-notch construction includes durable topstitched seams for strength with low-bulk, resilient rib-knit collar, cuffs and bottom. An 80% cotton/20% polyester blend makes it easy to keep them clean. </body></html>	1

ID	Description	Results
910	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Shorts</title></head><body lang=EN-US> These quick-drying cotton shorts provide all day comfort on or off the trails. Now with a more comfortable and stretchy fabric and an adjustable drawstring waist. </body></html>	1

另请参见

- “全文搜索”一节第 284 页
- “文本配置对象”一节第 285 页
- “CREATE TEXT CONFIGURATION 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “ALTER TEXT CONFIGURATION 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “文本索引”一节第 298 页
- “CREATE TEXT INDEX 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “ALTER TEXT INDEX 语句”一节 《SQL Anywhere 服务器 - SQL 参考》

教程：执行模糊全文搜索

可通过以下过程对使用 NGRAM 术语断开器的文本索引执行模糊全文搜索。

另请参见： “教程：对 GENERIC 文本索引执行全文搜索”一节第 319 页.

♦ 对 NGRAM 术语索引执行模糊全文搜索

1. 执行以下语句以创建名为 myFuzzyTextConfig 的文本配置对象。

```
CREATE TEXT CONFIGURATION myFuzzyTextConfig FROM default_char;
```

2. 执行以下语句，将术语断开器更改为 NGRAM 并将最大术语长度设置为 3。然后使用 n 元语法词执行模糊搜索。使用单独的 ALTER TEXT CONFIGURATION 语句来实现这些更改：

```
ALTER TEXT CONFIGURATION myFuzzyTextConfig
   TERM BREAKER NGRAM;
ALTER TEXT CONFIGURATION myFuzzyTextConfig
   MAXIMUM TERM LENGTH 3;
```

3. 创建 MarketingInformation 表的副本。
   a. 在 Sybase Central 中，展开 [**表**] 文件夹。
   b. 右击 **MarketingInformation** 并选择 [**复制**]。
   c. 右击 [**表**] 文件夹，然后选择 [**粘贴**]。
   d. 在 [**名称**] 字段中键入 **MarketingInformation2**。单击 [**确定**]。

4. 执行以下命令以向 MarketingInformation2 表中添加数据：

```
INSERT INTO MarketingInformation2
   SELECT * FROM MarketingInformation;
```

5. 执行以下命令可在 MarketingInformation2.Description 列上创建引用 myFuzzyTextConfig 文本配置对象的文本索引：

```
CREATE TEXT INDEX myFuzzyTextIdx ON MarketingInformation2 ( Description )
   CONFIGURATION myFuzzyTextConfig;
```

6. 执行以下语句来测试文本索引。

   以下模糊查询将检查是否存在与 **coten** 相类似的术语。

```
SELECT MarketingInformation2.Description, ct.*
    FROM MarketingInformation2 CONTAINS
( MarketingInformation2.Description, 'FUZZY "coten"' ) ct
    ORDER BY ct.score DESC;
```

Description	Score
<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Sweatshirt</title></head><body lang=EN-US> Lightweight 100% organically grown cotton hooded sweatshirt with taped neck seams. Comes pre-washed for softness and to lessen shrinkage. </body></html>	0.9461597363521859
<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Shorts</title></head><body lang=EN-US> These quick-drying cotton shorts provide all day comfort on or off the trails. Now with a more comfortable and stretchy fabric and an adjustable drawstring waist. </body></html>	0.9244136988525732
<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Visor</title></head><body lang=EN-US> Lightweight 100% organically grown cotton construction. Shields against sun and precipitation. Metallic ions in the fibers inhibit bacterial growth, and help neutralize odor. </body></html>	0.9134171046194403

Description	Score
<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Sweatshirt</title></head><body lang=EN-US> Top-notch construction includes durable topstitched seams for strength with low-bulk, resilient rib-knit collar, cuffs and bottom. An 80% cotton/20% polyester blend makes it easy to keep them clean. </body></html>	0.8856420222728282
<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Baseball Cap</title></head><body lang=EN-US> This fashionable hat is ideal for glacier travel, sea-kayaking, and hiking. With concealed draw cord for windy days. </body></html>	0
<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Baseball Cap</title></head><body lang=EN-US> A lightweight wool cap with mesh side vents for breathable comfort during aerobic activities. Moisture-absorbing headband liner. </body></html>	0
<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Tee Shirt</title></head><body lang=EN-US> We've improved the design of this perennial favorite. A sleek and technical shirt built for the trail, track, or sidewalk. UPF rating of 50+. </body></html>	0

Description	Score
<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Tee Shirt</title></head><body lang=EN-US> A sporty, casual shirt made of recycled water bottles. It will serve you equally well on trails or around town. The fabric has a wicking finish to pull perspiration away from your skin. </body></html>	0
<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Tee Shirt</title></head><body lang=EN-US> This simple, sleek, and lightweight technical shirt is designed for high-intensity workouts in hot and humid weather. The recycled polyester fabric is gentle on the earth and soft against your skin. </body></html>	0
<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Visor</title></head><body lang=EN-US> A polycarbonate visor with an abrasion-resistant coating on the outside. Great for jogging in the spring, summer, and early fall. The elastic headband has plenty of stretch to give you a snug yet comfortable fit every time you wear it. </body></html>	0

注意
最后六行具有包含匹配 n 元语法词的术语。然而，不会为其指定分数，因为表中的所有行都包含这些术语。

另请参见

- “全文搜索”一节第 284 页
- “文本配置对象”一节第 285 页
- “CREATE TEXT CONFIGURATION 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “ALTER TEXT CONFIGURATION 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “文本索引”一节第 298 页
- “CREATE TEXT INDEX 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “ALTER TEXT INDEX 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “对全文搜索结果进行计分”一节第 316 页

教程：对 NGRAM 文本索引执行全文搜索

可通过以下过程对使用 NGRAM 术语断开器的文本索引执行全文搜索。此过程也可用于创建中文、日文或朝鲜文数据的全文搜索。

在具有多字节字符集的数据库中，某些标点符号和空格字符（如全角逗号和全角空格）可能会处理为字母数字字符。

另请参见： “教程：执行模糊全文搜索”一节第 325 页.

♦ 对 NGRAM 文本索引执行全文搜索

1. 执行以下语句以创建名为 myNcharNGRAMTextConfig 的 NCHAR 文本配置对象：

   ```
   CREATE TEXT CONFIGURATION myNcharNGRAMTextConfig FROM default_nchar;
   ```

2. 执行以下语句，将 TERM BREAKER 算法更改为 NGRAM，并将 MAXIMUM TERM LENGTH (N) 设置为 2。

   ```
   ALTER TEXT CONFIGURATION myNcharNGRAMTextConfig
      TERM BREAKER NGRAM;

   ALTER TEXT CONFIGURATION myNcharNGRAMTextConfig
      MAXIMUM TERM LENGTH 2;
   ```

   对于中文、日文和朝鲜文数据，N 的建议值为 2 或 3。如果搜索限制为一个或两个字符，则将 N 值设置为 1。将 N 值设置为 1 可能会降低较长查询的执行速度。

3. 创建 MarketingInformation 表的副本。
   a. 在 Sybase Central 中，展开 [**表**] 文件夹。
   b. 右击 **MarketingInformation** 并选择 [**复制**]。
   c. 右击 [**表**] 文件夹，然后选择 [**粘贴**]。
   d. 在 [**名称**] 字段中键入 **MarketingInformationNgram**。单击 [**确定**]。

4. 执行以下语句向 MarketingInformationNgram 表中添加数据：

   ```
   INSERT INTO MarketingInformationNgram
      SELECT *
         FROM MarketingInformation;
   COMMIT;
   ```

5. 执行以下语句可在 MarketingInformationNgram.Description 列上创建使用 myNcharNGRAMTextConfig 文本配置对象的 IMMEDIATE REFRESH 文本索引：

```
CREATE TEXT INDEX ncharNGRAMTextIndex
   ON MarketingInformationNgram( Description )
      CONFIGURATION myNcharNGRAMTextConfig;
```

6. 执行以下语句来测试文本索引。

   a. 此语句在 2 元语法词文本索引中搜索包含 **sw** 的术语。结果按分数降序排序。

```
SELECT M.Description, ct.*
   FROM MarketingInformationNgram AS M
CONTAINS( M.Description, 'sw' ) ct
   ORDER BY ct.score DESC;
```

Description	Score
<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Sweatshirt</ title></head><body lang=EN-US> Lightweight 100% organically grown cotton hooded Sweatshirt with taped neck seams. Comes pre-washed for softness and to lessen shrinkage. </body></html>	2.262071918398649
<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Sweatshirt</ title></head><body lang=EN-US> Top-notch construction includes durable topstitched seams for strength with low-bulk, resilient rib-knit collar, cuffs and bottom. An 80% cotton/ 20% polyester blend makes it easy to keep them clean. </body></html>	1.5556043490424176

   b. 以下语句搜索包含 **ams** 的术语。结果按分数降序排序。

```
SELECT M.Description, ct.*
   FROM MarketingInformationNgram AS M
CONTAINS( M.Description, 'ams' ) ct
   ORDER BY ct.score DESC;
```

   对于 2 元语法词文本索引，以上语句在语义上等效于：

```
SELECT M.Description, ct.*
   FROM MarketingInformationNgram AS M
CONTAINS( M.Description, '"am ms"' ) ct
   ORDER BY ct.score DESC;
```

Description	Score
<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Sweatshirt</title></head><body lang=EN-US> Lightweight 100% organically grown cotton hooded sweatshirt with taped neck seams. Comes pre-washed for softness and to lessen shrinkage. </body></html>	1.6619019465461564
<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Sweatshirt</title></head><body lang=EN-US> Top-notch construction includes durable topstitched seams for strength with low-bulk, resilient rib-knit collar, cuffs and bottom. An 80% cotton/20% polyester blend makes it easy to keep them clean. </body></html>	1.5556043490424176

c. 以下语句搜索包含后跟任意字母数字字符的 **v** 的术语。因为 **ve** 在索引数据中出现频率较高，所以包含 2 元语法词 **ve** 的行所得到的分数要低于包含 **vi** 的行。结果按分数降序排序。

```
SELECT M.ID, M.Description, ct.*
   FROM MarketingInformationNgram AS M
CONTAINS( M.Description, 'v*' ) ct
   ORDER BY ct.score DESC;
```

ID	Description	Score
901	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Tee Shirt</title></head><body lang=EN-US> We've improved the design of this perennial favvorite. A sleek and technical shirt built for the trail, track, or sidewalk. UPF rating of 50+. </body></html>	3.3416789108071976

ID	Description	Score
907	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Visor</title></head><body lang=EN-US> A polycarbonate visor with an abrasion-resistant coating on the outside. Great for jogging in the spring, summer, and early fall. The elastic headband has plenty of stretch to give you a snug yet comfortable fit every time you wear it. </body></html>	2.1123084896159376
905	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Baseball Cap</title></head><body lang=EN-US> A lightweight wool cap with mesh side vents for breathable comfort during aerobic activities. Moisture-absorbing headband liner. </body></html>	1.6750365447462499
910	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Shorts</title></head><body lang=EN-US> These quick-drying cotton shorts provide all day comfort on or off the trails. Now with a more comfortable and stretchy fabric and an adjustable drawstring waist. </body></html>	0.9244136988525732
906	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Visor</title></head><body lang=EN-US> Lightweight 100% organically grown cotton construction. Shields against sun and precipitation. Metallic ions in the fibers inhibit bacterial growth, and help neutralize odor. </body></html>	0.9134171046194403

ID	Description	Score
904	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Baseball Cap</title></head><body lang=EN-US> This fashionable hat is ideal for glacier travel, sea-kayaking, and hiking. With concealed draw cord for windy days. </body></html>	0.7313071661212746
903	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Tee Shirt</title></head><body lang=EN-US> A sporty, casual shirt made of recycled water bottles. It will serve you equally well on trails or around town. The fabric has a wicking finish to pull perspiration away from your skin. </body></html>	0.6799436746197272

d. 以下语句在各行中搜索包含 v 的全部术语。第二个语句之后，变量包含字符串 [av OR ev OR iv OR ov OR rv OR ve OR vi OR vo]。结果按分数降序排序。如果某一 n 元语法词在所有索引行中出现，则为其指定的分数为零。

   如果单个字符出现在空白字符或非字母数字字符之前，那么这是用于定位该字符的唯一方法。

```
CREATE VARIABLE query NVARCHAR (100);
SELECT LIST (term, ' OR ' )
INTO query
   FROM sa_text_index_vocab( 'ncharNGRAMTextIndex',
'MarketingInformationNgram', 'dba' )
   WHERE term LIKE '%v%';
SELECT M.ID, M.Description, ct.*
   FROM MarketingInformationNgram AS M
   CONTAINS( M.Description, query ) ct
   ORDER BY ct.score DESC;
```

ID	Description	Score
901	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Tee Shirt</title></head><body lang=EN-US> We've improved the design of this perennial favorite. A sleek and technical shirt built for the trail, track, or sidewalk. UPF rating of 50+. </body></html>	6.654350268810443
907	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Visor</title></head><body lang=EN-US> A polycarbonate visor with an abrasion-resistant coating on the outside. Great for jogging in the spring, summer, and early fall. The elastic headband has plenty of stretch to give you a snug yet comfortable fit every time you wear it. </body></html>	4.265623837817126
903	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Tee Shirt</title></head><body lang=EN-US> A sporty, casual shirt made of recycled water bottles. It will serve you equally well on trails or around town. The fabric has a wicking finish to pull perspiration away from your skin. </body></html>	2.9386676702799504

ID	Description	Score
910	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Shorts</title></head><body lang=EN-US> These quick-drying cotton shorts provide all day comfort on or off the trails. Now with a more comfortable and stretchy fabric and an adjustable drawstring waist. </body></html>	2.5481193655722336
904	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Baseball Cap</title></head><body lang=EN-US> This fashionable hat is ideal for glacier travel, sea-kayaking, and hiking. With concealed draw cord for windy days. </body></html>	2.4293498211307214
905	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Baseball Cap</title></head><body lang=EN-US> A lightweight wool cap with mesh side vents for breathable comfort during aerobic activities. Moisture-absorbing headband liner. </body></html>	1.6750365447462499
906	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Visor</title></head><body lang=EN-US> Lightweight 100% organically grown cotton construction. Shields against sun and precipitation. Metallic ions in the fibers inhibit bacterial growth, and help neutralize odor. </body></html>	0.9134171046194403

ID	Description	Score
902	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Tee Shirt</title></head><body lang=EN-US> This simple, sleek, and lightweight technical shirt is designed for high-intensity workouts in hot and humid weather. The recycled polyester fabric is gentle on the earth and soft against your skin. </body></html>	0
908	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Sweatshirt</title></head><body lang=EN-US> Lightweight 100% organically grown cotton hooded sweatshirt with taped neck seams. Comes pre-washed for softness and to lessen shrinkage. </body></html>	0
909	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Sweatshirt</title></head><body lang=EN-US> Top-notch construction includes durable topstitched seams for strength with low-bulk, resilient rib-knit collar, cuffs and bottom. An 80% cotton/20% polyester blend makes it easy to keep them clean. </body></html>	0

e. 以下语句在 Description 列中搜索包含 **ea**、**ka** 和 **k** 的行。

```
SELECT M.ID, M.Description, ct.*
   FROM MarketingInformationNgram AS M
CONTAINS( M.Description, 'ea ka ki' ) ct
  ORDER BY ct.score DESC;
```

ID	Description	Score
904	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Baseball Cap</title></head><body lang=EN-US> This fashionable hat is ideal for glacier travel, sea-kayaking, and hiking. With concealed draw cord for windy days. </body></html>	3.4151032739119733

f. 以下语句在 Description 列中搜索包含 **ve** 和 **vi**，但不包含 **gg** 的行。

```
SELECT M.ID, M.Description, ct.*
   FROM MarketingInformationNgram AS M
CONTAINS( M.Description, 've & vi  -gg' ) ct
   ORDER BY ct.score DESC;
```

ID	Description	Score
905	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Baseball Cap</title></head><body lang=EN-US> A lightweight wool cap with mesh side vents for breathable comfort during aerobic activities. Moisture-absorbing headband liner. </body></html>	1.6750365447462499

对查询结果进行汇总、分组和排序

目录

使用集合函数汇总查询结果

集合函数可以显示指定列中值的汇总。还可以使用 GROUP BY 子句、HAVING 子句和 ORDER BY 子句对使用集合函数得到的查询结果进行分组和排序，以及使用 UNION 运算符来合并查询结果。

可以将集合函数应用于表中的所有行、WHERE 子句指定的表的子集或表中的一组行或多组行。SQL Anywhere 会从每一组应用了集合函数的行生成一个值。

以下是部分受支持的集合函数：

- **avg（表达式）** 对于返回的行，计算所提供的表达式的平均值。
- **count（表达式）** 在表达式不是 NULL 的情况下计算所提供的组中的行数。
- **count(*)** 计算每一组中行的数目。
- **list（字符串表达式）** 包含逗号分隔列表的字符串，由每一组行中 *string-expr* 的所有值组成。
- **max（表达式）** 对于返回的行，计算表达式的最大值。
- **min（表达式）** 对于返回的行，计算表达式的最小值。
- **stddev（表达式）** 对于返回的行，计算表达式的标准偏差。
- **sum（表达式）** 对于返回的行，计算表达式的总和。
- **variance（表达式）** 对于返回的行，计算表达式的方差。

有关集合函数的完整列表，请参见“集合函数”一节 《SQL Anywhere 服务器 - SQL 参考》。

您可以将可选关键字 DISTINCT 与 AVG、SUM、LIST 和 COUNT 结合使用，以在应用集合函数前消除重复值。

语法语句引用的表达式通常是列名。也可以是更普通的表达式。

例如，通过以下语句，您可以求出在将 1 美元与每一个价格相加之后所有产品的平均价格：

```
SELECT AVG ( UnitPrice + 1 )
FROM Products;
```

示例

以下查询根据 Employees 表中的年薪来计算工资单总额：

```
SELECT SUM( Salary )
FROM Employees;
```

要使用集合函数，必须提供后跟表达式的函数名，该表达式说明函数所的作用对象。该表达式（在此示例中是 Salary 列）是函数的参数，必须在括号内指定它。

在哪些地方可以使用集合函数

可以在选择列表中使用集合函数（如前面的示例中所示），也可以在包括 GROUP BY 子句的选择语句的 HAVING 子句中使用集合函数。请参见“HAVING 子句：选择数据组”一节第 349 页。

不能在 WHERE 子句或 JOIN 条件中使用集合函数。不过，选择列表中有集合函数的 SELECT 语句通常包括 WHERE 子句，该子句会对应用集合的行加以限制。

如果 SELECT 语句包括 WHERE 子句，而不包括 GROUP BY 子句，则集合函数会为 WHERE 子句指定的行的子集生成一个值。

只要是在不包括 GROUP BY 子句的 SELECT 语句中使用集合函数，集合函数就会生成一个值。无论集合函数是作用于表中的所有行，还是 where 子句定义的行的子集，都是如此。

您可以在同一选择列表中使用多个集合函数，并且可以在一个 SELECT 语句中生成多个标量集合。

集合函数和外部引用

SQL Anywhere 遵循 SQL/2003 标准，明确集合函数在子查询中出现时的用法。这些更改将影响为该软件的以前版本编写的语句的行为：以前正确的查询现在可能产生错误消息，结果集也可能发生变化。

当集合函数出现在子查询中并且集合函数引用的列是外部引用时，整个集合函数本身现在会被视为外部引用。这意味着集合函数现在将在外部块而非子查询中进行计算，并会成为子查询内的常量。

在子查询中使用外部引用集合函数时受到以下限制：

- 外部引用集合函数只能出现在位于 SELECT 列表或 HAVING 子句中的子查询中，并且这些子句必须位于紧接的外部块中。
- 外部引用集合函数只能包含一个外部列引用。
- 本地列引用和外部列引用不能在同一集合函数中混用。

通过重写集合函数来使其仅包括本地引用，可以规避与新标准有关的某些问题。例如，子查询 `(SELECT MAX(S.y + R.y) FROM S)` 同时包含本地列引用 (S.y) 和外部列引用 (R.y)，现在这是非法的。可以将其重写为 `(SELECT MAX(S.y) + R.y FROM S)`。在改写的内容中，集合函数只具有本地列引用。当外部引用集合函数出现在 SELECT 或 HAVING 以外的子句中时，可以进行同一类型的改写。

示例

以下查询在 Adaptive Server Anywhere 第 7 版中生成以下结果。

```
SELECT Name,
       ( SELECT SUM( p.Quantity )
            FROM SalesOrderItems )
   FROM Products p;
```

Name	SUM(p.Quantity)
Tee shirt	30,716
Tee shirt	59,238

在更高版本中，同一查询会生成错误消息 [`SQL Anywhere` **错误** `-149:对 'name' 的函数或列引用还必须出现在 GROUP BY 中`]。该语句不再有效的原因在于：外部引用集合函数 [`sum(p.Quantity)`] 现在是在外部块中进行计算。在更高版本中，该查询在语义上等效于以下语句（不同的是 Z 不作为结果集的一部分出现）：

```
SELECT Name,
       SUM( p.Quantity ) AS Z,
       ( SELECT Z
            FROM SalesOrderItems )
   FROM Products p;
```

由于现在是由外部块来计算集合函数，因此外部块被视作分组查询，而列名必须出现在 GROUP BY 子句中，才能在 SELECT 列表中出现。

集合函数和数据类型

一些集合函数仅对某些类型的数据有意义。例如，只能将 SUM 和 AVG 与数字类型的列一起使用。

不过，可以使用 MIN 求出字符列中的最小值，即距离字母表的开头最近的值：

```
SELECT MIN( Surname )
   FROM Contacts;
```

使用 COUNT(*)

COUNT(*) 返回指定表中的行数（不排除重复的行）。它会对每一行单独计数，包括含有 NULL 的行。该函数不需要使用表达式作为参数，因为按照定义它不使用与任何特定列有关的信息。

以下语句查找 Employees 表中的雇员总数：

```
SELECT COUNT(*)
   FROM Employees;
```

与其它集合函数一样，可以将 COUNT(*) 与选择列表中的其它集合函数、WHERE 子句等结合使用。例如：

```
SELECT COUNT(*), AVG( UnitPrice )
   FROM Products
   WHERE UnitPrice > 10;
```

COUNT(*)	AVG(Products.UnitPrice)
5	18.2

将集合函数与 DISTINCT 一起使用

可选择将 DISTINCT 关键字与 SUM、AVG 和 COUNT 一起使用。如果使用 DISTINCT，则在计算总和、平均值或计数之前会消除重复值。例如，要求出有联系人的不同城市的数目，请执行以下语句：

```
SELECT COUNT( DISTINCT City )
   FROM Contacts;
```

COUNT(DISTINCT Contacts.City)
16

可以在查询中使用多个含 DISTINCT 的集合函数。每个 DISTINCT 都是独立计算的。例如：

```
SELECT COUNT( DISTINCT GivenName ) "first names",
       COUNT( DISTINCT Surname ) "last names"
   FROM Contacts;
```

first names	last names
48	60

集合函数和 NULL

对于除 COUNT(*) 之外的函数，集合函数所作用的列中的所有 NULL 都会被忽略；而 COUNT(*) 函数则会包括所有 NULL。如果列中的所有值均为 NULL，则 COUNT(column_name) 返回 0。

如果没有任何行符合 WHERE 子句所指定的条件，则 COUNT 返回值 0。其它函数都返回 NULL。以下是一些示例：

```
SELECT COUNT( DISTINCT Name )
   FROM Products
   WHERE UnitPrice > 50;
```

COUNT(DISTINCT Name)
0

```
SELECT AVG( UnitPrice )
   FROM Products
   WHERE UnitPrice > 50;
```

AVG(Products.UnitPrice)
(NULL)

GROUP BY 子句：将查询结果划分为组

GROUP BY 子句将表的输出划分为若干个组。可以按一个或多个列名或者按计算列的结果将行分组。

子句顺序
GROUP BY 子句必须始终在 HAVING 子句之前。如果 WHERE 子句和 GROUP BY 子句都存在，则 WHERE 子句必须在 GROUP BY 子句之前。

可以在一个查询中同时使用 HAVING 子句和 WHERE 子句。只有在构造了组之后，HAVING 子句中的条件才会在逻辑上限制结果行。WHERE 子句中的条件在构造组之前进行逻辑计算，这样可以节约时间。

当查询涉及 GROUP BY 子句时，很难了解哪些查询有效哪些查询无效。本节介绍如何对含有 GROUP BY 的查询进行分析，以便可以更好地理解查询的结果和有效性。

具有 GROUP BY 的查询是如何执行的

本节在说明和示例中使用 GROUP BY 子句的 ROLLUP 从属子句。有关 ROLLUP 子句的详细信息，请参见“使用 ROLLUP 和 CUBE 作为 GROUPING SETS 的快捷方式”一节第 430 页。

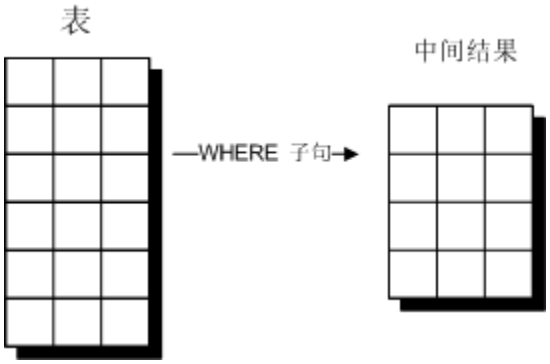
假设有一个以下形式的单表查询：

```
SELECT select-list
  FROM table
  WHERE where-search-condition
  GROUP BY [ group-by-expression | ROLLUP (group-by-expression) ]
  HAVING having-search-condition
```

该查询的执行方式如下：

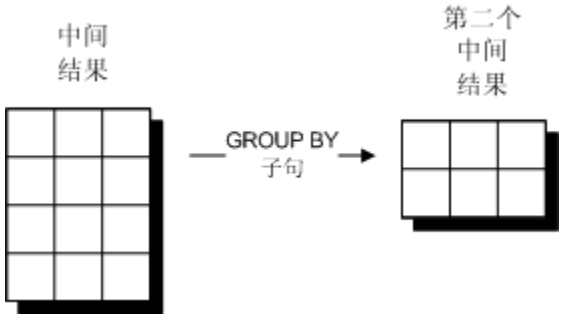
1. **应用 WHERE 子句**

   这就生成只包含该表的某些行的中间结果。



2. **将结果划分为若干个组**

此操作按 GROUP BY 子句的要求生成一个中间结果，结果中的每一行对应一组。生成的每一行都包含对应于一个组的 *group-by-expression*，并包含 *select-list* 和 *having-search-condition* 中已计算完的集合函数。



3. **应用任何 ROLLUP 操作**

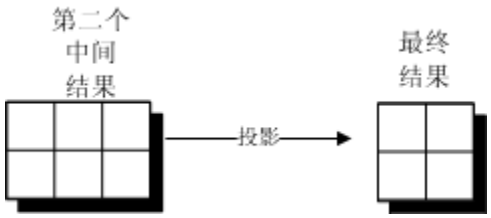
   将作为 ROLLUP 操作的一部分计算的分类汇总行添加到结果集中。

4. **应用 HAVING 子句**

   此时，此第二个中间结果中的不满足 HAVING 子句条件的所有行都将被删除。

5. **将结果投影到显示屏中**

   此操作只从第 3 步获取需要在查询的结果集中显示的那些列，也就是说，它只获取与 *select-list* 中的表达式相对应的那些列。



此过程对带有 GROUP BY 子句的查询有如下要求：

- 首先计算 WHERE 子句。因此，将只对满足 WHERE 子句条件的那些行计算集合函数。
- 最终结果集是根据第二个中间结果（保存已划分的行）建立的。第二个中间结果保留与 *group-by-expression* 相对应的那些行。因此，如果一个非集合函数的表达式出现在 *select-list* 中，则该表达式也必须出现在 *group-by-expression* 中。在投影步骤中无法执行函数计算。
- 可以在 *group-by-expression* 中包含表达式，但不能在 *select-list* 中包含表达式。它被投影到结果之外。

使用具有多个列的 GROUP BY

可以在 GROUP BY 子句中列出多个表达式—也就是说，可以通过表达式的任意组合对表进行分组。

以下查询列出了产品的平均价格，首先按名称分组，然后按尺寸分组：

```
SELECT Name, Size, AVG( UnitPrice )
   FROM Products
   GROUP BY Name, Size;
```

Name	Size	AVG(Products.UnitPrice)
Baseball Cap	One size fits all	9.5
Sweatshirt	Large	24
Tee Shirt	Large	14
Tee Shirt	One size fits all	14
...	...	...

GROUP BY 中的不位于选择列表中的列

Adaptive Server Enterprise 和 SQL Anywhere 均支持 Sybase 对 SQL/92 标准的一个扩充，即允许对不在选择列表中的 GROUP BY 子句使用表达式。例如，以下查询列出了每个城市中联系人的数目：

```
SELECT State, COUNT( ID )
   FROM Contacts
   GROUP BY State, City;
```

WHERE 子句和 GROUP BY

可以在含有 GROUP BY 的语句中使用 WHERE 子句。WHERE 子句将先于 GROUP BY 子句进行计算。在执行任何分组前消除不满足 WHERE 子句中的条件的行。下面是一个示例：

```
SELECT  Name, AVG( UnitPrice )
   FROM Products
   WHERE ID > 400
   GROUP BY Name;
```

用于生成查询结果的组中只包括 ID 值大于 400 的行。

示例

以下查询举例说明了在一个查询中同时使用 WHERE、GROUP BY 和 HAVING 子句的方法：

```
SELECT Name, SUM( Quantity )
   FROM Products
   WHERE Name LIKE '%shirt%'
   GROUP BY Name
   HAVING SUM( Quantity ) > 100;
```

Name	SUM(Products.Quantity)
Tee Shirt	157

在此示例中：

- WHERE 子句只包括名称中含有 *shirt* 一词（Tee Shirt、Sweatshirt）的行。
- GROUP BY 子句收集同名的行。
- SUM 集合函数将计算每一组中的产品总数。
- HAVING 子句从最终结果中排除其库存总计不超过 100 的那些组。

将 GROUP BY 与集合函数一起使用

GROUP BY 子句几乎总是出现在包括集合函数的语句中，在此情况下，集合会为每个组生成一个值。这些值被称作**矢量集合**。（**标量集合**是由不带 GROUP BY 子句的集合函数生成的单个值。）

示例

以下查询列出了每一种产品的平均价格：

```
SELECT Name, AVG( UnitPrice ) AS Price
   FROM Products
   GROUP BY Name;
```

Name	Price
Tee Shirt	12.333333333
Baseball Cap	9.5
Visor	7
Sweatshirt	24
...	...

由带有集合和 GROUP BY 的 SELECT 语句所生成的矢量集合在每个结果行中都以列的形式出现。相比之下，由带有集合但不带 GROUP BY 的查询生成的标量集合也以列的形式出现，但只有一行。例如：

```
SELECT AVG( UnitPrice )
   FROM Products;
```

AVG(Products.UnitPrice)
13.3

GROUP BY 和 SQL/2003 标准

针对 GROUP BY 的 SQL/2003 标准的要求如下：

- GROUP BY 子句中必须有在 SELECT 子句的表达式中使用的列。否则，使用该列的表达式将是集合函数。
- GROUP BY 表达式只能包含选择列表中的列名，而不能包含只用作矢量集合参数的那些列名。

带有矢量集合函数的标准 GROUP BY 会生成每组具有一个值的一行。

SQL Anywhere 支持在 HAVING 子句中使用集合函数这一扩充，即使这些集合函数在选择列表或 GROUP BY 子句中不存在。

有关 SQL Anywhere 是否符合其它标准的详细信息，请参见“SQL 方言”第 605 页。

HAVING 子句：选择数据组

HAVING 子句会对查询所返回的行加以限制。它为 GROUP BY 子句设置条件的方式与 WHERE 为 SELECT 子句设置条件的方式类似。

HAVING 子句搜索条件基本上等同于 WHERE 搜索条件，只是 WHERE 搜索条件不能包括集合。例如，允许以下用法：

```
HAVING AVG( UnitPrice ) > 20
```

不允许以下用法：

```
WHERE AVG( UnitPrice ) > 20
```

将 HAVING 与集合函数一起使用

以下语句是将 HAVING 子句与集合函数一起使用的简单示例。

要列出那些有多种尺寸或颜色的产品，您需要使用查询来按名称对 Products 表中的行分组，但要将只包含一种不同产品的组排除在外：

```
SELECT Name
   FROM Products
   GROUP BY Name
   HAVING COUNT( * ) > 1;
```

Name
Tee Shirt
Baseball Cap
Visor
Sweatshirt

有关哪些情况下可以在 HAVING 子句中使用集合函数的信息，请参见“在哪些地方可以使用集合函数”一节第 340 页。

不与集合函数一起使用 HAVING

HAVING 子句也可以不与集合函数一起使用。

以下查询对产品进行分组，然后将结果集限制为只包含名称以 B 开头的那些组。

```
SELECT Name
   FROM Products
   GROUP BY Name
   HAVING Name LIKE 'B%';
```

Name
Baseball Cap

在 HAVING 中有多个条件

可以在 HAVING 子句中包括多个条件。可以用 AND、OR 或 NOT 运算符将这些条件组合起来，如以下示例所示。

要列出那些有多种尺寸或颜色，并且一个款式的价格超过 10 美元的产品，您需要使用查询来按名称将 Products 表中的行分组，但要将只包含一种不同产品的组以及最高单价低于 10 美元的组排除在外。

```
SELECT Name
   FROM Products
   GROUP BY Name
   HAVING COUNT( * ) > 1
   AND MAX( UnitPrice ) > 10;
```

Name
Tee Shirt
Sweatshirt

ORDER BY 子句：查询结果排序

ORDER BY 子句用于按一列或多列对查询结果进行排序。每一次排序都可以是升序 (ASC) 或降序 (DESC)。如果两种排序顺序都未予指定，则采用 ASC 进行排序。

简单示例

以下查询将返回按名称排序的结果：

```
SELECT ID, Name
   FROM Products
   ORDER BY Name;
```

ID	Name
400	Baseball Cap
401	Baseball Cap
700	Shorts
600	Sweatshirt
...	...

按多列排序

如果您在 ORDER BY 子句中命名了多个列，排序将形成嵌套形式。

以下语句先按名称对 Products 表中的衬衫进行升序排序，然后在每一名称内按 Quantity（降序）进行排序：

```
SELECT ID, Name, Quantity
   FROM Products
   WHERE Name like '%shirt%'
   ORDER BY Name, Quantity DESC;
```

ID	Name	Quantity
600	Sweatshirt	39
601	Sweatshirt	32
302	Tee Shirt	75
301	Tee Shirt	54
...	...	...

使用列位置

可以在选择列表中不使用列名，而改用列位置号。列名称和选择列表编号可以混合使用。以下两个语句与前面语句生成的结果是相同的。

```
SELECT ID, Name, Quantity
   FROM Products
   WHERE Name like '%shirt%'
   ORDER BY 2, 3 DESC;
SELECT ID, Name, Quantity
   FROM Products
   WHERE Name like '%shirt%'
   ORDER BY 2, Quantity DESC
```

大多数 SQL 版本都要求选择列表中有 ORDER BY，但 SQL Anywhere 没有此限制。以下查询按 Quantity 对结果进行排序，尽管该列并不出现在选择列表中：

```
SELECT ID, Name
   FROM Products
   WHERE Name like '%shirt%'
   ORDER BY 2, Quantity DESC;
```

ORDER BY 和 NULL

使用 ORDER BY 时如果按升序进行排序，NULL 将排在所有其它值之前。

ORDER BY 和大小写

ORDER BY 子句对混合大小写数据的影响取决于创建数据库时指定的数据库归类和大小写区分属性。

显式限制查询返回的行数

可以使用 FIRST 或 TOP 关键字限制查询结果集中包括的行数。这些关键字用于包括 ORDER BY 子句的查询。

示例

以下查询返回在按姓氏对雇员进行排序时首先出现的雇员的信息：

```
SELECT FIRST *
   FROM Employees
   ORDER BY Surname;
```

以下查询返回按姓氏排序时出现的前五个雇员：

```
SELECT TOP 5 *
   FROM Employees
   ORDER BY Surname;
```

使用 TOP 时，还可以使用 START AT 来提供一个偏移值。以下语句列出了按姓氏以降序进行排序时出现的第五个和第六个雇员：

```
SELECT TOP 2 START AT 5 *
   FROM Employees
   ORDER BY Surname DESC;
```


FIRST 和 TOP 应只与 ORDER BY 子句联合使用，以确保获得一致的结果。如果不将 FIRST 或 TOP 与 ORDER BY 联合使用，则会触发语法警告，并可能产生无法预知的结果。

注意
START AT 值必须大于 0。当 TOP 为常量时，其值必须大于 0；当 TOP 为变量时，其值必须大于或等于 0。

ORDER BY 和 GROUP BY

您可以使用 ORDER BY 子句以特定方式对 GROUP BY 的结果排序。

示例

以下查询将求出每一产品的平均价格并按平均价格对结果排序：

```
SELECT Name, AVG( UnitPrice )
   FROM Products
   GROUP BY Name
   ORDER BY AVG( UnitPrice );
```

Name	AVG(Products.UnitPrice)
Visor	7
Baseball Cap	9.5
Tee Shirt	12.333333333
Shorts	15
...	...

使用 UNION、INTERSECT 和 EXCEPT 对查询结果执行集合运算

本节中介绍的运算符用于对两个或多个查询的结果执行集合运算。尽管其中的许多运算也可以通过 WHERE 子句或 HAVING 子句中的运算来执行，但有一些运算如果不使用这些基于集合的运算符来执行，难度会非常大。例如：

- 当数据未进行规范化时，您可能想要将看似不同的信息汇编到单个结果集中，尽管这些表并不相关。
- 集合运算符对 NULL 的处理方式与 WHERE 子句或 HAVING 子句中对 NULL 的处理方式不同。在 WHERE 子句或 HAVING 子句中，具有完全相同的非空条目的两个包含空值行不被视为完全相同，因为这两个 NULL 值没有定义为完全相同。集合运算符将两个这样的行视为相同。

另请参见

- “集合运算符和 NULL”一节第 356 页
- “EXCEPT 子句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “INTERSECT 子句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “UNION 子句”一节 《SQL Anywhere 服务器 - SQL 参考》

使用 UNION 语句组合集合

UNION 运算符可以将两个或多个查询的结果组合到一个结果集中。

缺省情况下，UNION 运算符可以从结果集中删除重复行。如果使用 ALL 选项，则不删除重复行。最终结果集中的列与第一个结果集中的列具有相同的名称。可以使用任意数量的 UNION 运算符。

缺省情况下，按从左到右的顺序计算包含多个 UNION 运算符的语句。可以使用括号来指定计算顺序。

例如，以下两个表达式不是等效的，因为它们从结果集中删除重复行的方式不同：

```
x UNION ALL ( y UNION z )
(x UNION ALL y) UNION z
```

在第一个表达式中，y 和 z 之间的 UNION 中删除了重复行，而得到的集合与 x 之间的 UNION 则不删除重复行。在第二个表达式中，x 与 y 之间的 UNION 包括重复行，但随后在与 z 的 UNION 中则会删除这些重复的行。

使用 EXCEPT 和 INTERSECT

EXCEPT 语句列出两个结果集的差集。以下总结构会列出所有出现在 query-1 结果集中但未出现在 query-2 结果集中的行。

query-1
EXCEPT
query-2

INTERSECT 语句用于列出在两个结果集中都出现的行。以下总结构会列出所有同时出现在 query-1 和 query-2 结果集中的行。

query-1
INTERSECT
query-2

与 UNION 语句一样，EXCEPT 和 INTERSECT 都采用 ALL 修饰符，该修饰符可防止从结果集中消除重复行。

另请参见

- “EXCEPT 子句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “INTERSECT 子句”一节 《SQL Anywhere 服务器 - SQL 参考》

集合运算的规则

以下规则适用于 UNION、EXCEPT 和 INTERSECT 语句：

- **选择列表中项的数目相同** 查询中的所有选择列表都必须具有相同数目的表达式（如列名、算术表达式和集合函数）。以下语句是无效的，因为第一个选择列表比第二个选择列表长：

```
SELECT store_id, city, state
   FROM stores
UNION
   SELECT store_id, city
     FROM stores_east;
```

- **数据类型必须匹配** SELECT 列表中的相应表达式必须属于同一数据类型，或者在两种数据类型之间必须能够进行隐式数据转换，或者应提供显式转换。

  例如，除非提供显式转换，否则在 CHAR 数据类型的列和 INT 数据类型的列之间无法进行 UNION、INTERSECT 或 EXCEPT 运算。不过，在 MONEY 数据类型的列和 INT 数据类型的列之间可以进行集合运算。

- **列排序** 您必须以相同的顺序将相应的表达式放置于集合运算的各个查询中，因为集合运算将按 SELECT 子句的各个查询中给定的顺序一一比较这些表达式。

- **多个集合运算** 您可以将若干个集合运算排列在一起，如以下示例所示：

```
SELECT City AS Cities
   FROM Contacts
   UNION
      SELECT City
         FROM Customers
   UNION
      SELECT City
         FROM Employees;
```

  对于 UNION 语句，查询的顺序并不重要。对于 INTERSECT，当存在两个或多个查询时顺序是很重要的。对于 EXCEPT，顺序在任何情况下都很重要。

- **列标题** 由 UNION 运算生成的表中的列名是从该语句中的第一个单独查询中获取的。如果要为结果集定义新的列标题，可以在第一个查询的选择列表中执行该操作，如以下示例所示：

```
SELECT City AS Cities
   FROM Contacts
   UNION
      SELECT City
         FROM Customers;
```

在以下查询中，列标题仍为 City，即 UNION 语句的第一个查询中定义的列标题。

```
SELECT City
   FROM Contacts
   UNION
      SELECT City AS Cities
         FROM Customers;
```

也可以使用 WITH 子句来定义列名。例如：

```
WITH V( Cities )
AS ( SELECT City
     FROM Contacts
     UNION
        SELECT City
         FROM Customers )
SELECT * FROM V;
```

- **对结果进行排序** 可以使用 SELECT 语句的 WITH 子句对选择列表中的列名进行排序。例如：

```
WITH V( CityName )
AS ( SELECT City AS Cities
     FROM Contacts
     UNION
        SELECT City
           FROM Customers )
SELECT * FROM V
   ORDER BY CityName;
```

您也可以在查询列表的末尾使用单个 ORDER BY 子句，但必须使用整数而不是列名，如以下示例所示：

```
SELECT City AS Cities
   FROM Contacts
   UNION
      SELECT City
        FROM Customers
   ORDER BY 1;
```

集合运算符和 NULL

集合运算符 UNION、EXCEPT 和 INTERSECT 对 NULL 的处理方式与搜索条件中对 NULL 的处理方式不同。此差异是使用集合运算符的主要原因之一。

比较行时，集合运算符将 NULL 值视为彼此相等。而相反，在搜索条件中将 NULL 与 NULL 进行比较时，结果是未知的（不为真）。

这种差异的一个结果是，`query-1 EXCEPT ALL query-2` 结果集中的行数*始终*是各个查询结果集中的行数之差。

以两个表 T1 和 T2 为例，其中每个表都具有以下列：


```
col1 INT,
col2 CHAR(1)
```

表和数据的设置如下：

```
CREATE TABLE T1 (col1 INT, col2 CHAR(1));
CREATE TABLE T2 (col1 INT, col2 CHAR(1));
INSERT INTO T1 (col1, col2) VALUES(1, 'a');
INSERT INTO T1 (col1, col2) VALUES(2, 'b');
INSERT INTO T1 (col1) VALUES(3);
INSERT INTO T1 (col1) VALUES(3);
INSERT INTO T1 (col1) VALUES(4);
INSERT INTO T1 (col1) VALUES(4);
INSERT INTO T2 (col1, col2) VALUES(1, 'a');
INSERT INTO T2 (col1, col2) VALUES(2, 'x');
INSERT INTO T2 (col1) VALUES(3);
```

表中的数据如下：

- 表 T1。

col1	col2
1	a
2	b
3	(NULL)
3	(NULL)
4	(NULL)
4	(NULL)

- 表 T2

col1	col2
1	a
2	x
3	(NULL)

一个询问同时出现在 T1 和 T2 中的行的查询如下：

```
SELECT T1.col1, T1.col2
   FROM T1 JOIN T2
   ON T1.col1 = T2.col1
   AND T1.col2 = T2.col2;
```

T1.col1	T1.col2
1	a

(3, NULL) 这一行不出现在结果集中，因为 NULL 和 NULL 之间的比较不为真。相比之下，使用 INTERSECT 运算符处理该问题时，则会包括含有 NULL 的行：

```
SELECT col1, col2
   FROM T1
   INTERSECT
      SELECT col1, col2
         FROM T2;
```

col1	col2
1	a
3	(NULL)

以下查询使用搜索条件列出 T1 中那些不出现在 T2 中的行：

```
SELECT col1, col2
   FROM T1
   WHERE col1 NOT IN (
       SELECT col1
         FROM T2
         WHERE T1.col2 = T2.col2 )
   OR col2 NOT IN (
       SELECT col2
         FROM T2
         WHERE T1.col1 = T2.col1 );
```

col1	col2
2	b
3	(NULL)
4	(NULL)
3	(NULL)
4	(NULL)

通过比较，并未将 T1 中包含 NULL 的行排除在外。相比之下，使用 EXCEPT ALL 处理该问题会排除在这两个表中都出现的包含 NULL 的行。在此情况下，即认为 T2 中的 (3, NULL) 行与 T1 中的 (3, NULL) 行相同。

```
SELECT col1, col2
   FROM T1
   EXCEPT ALL
      SELECT col1, col2
         FROM T2;
```


col1	col2
2	b
3	(NULL)
4	(NULL)
4	(NULL)

尽管如此，EXCEPT 运算符还是有更多限制。它会将 T1 中的两个 (3, NULL) 行都去除，并将其中一个 (4, NULL) 行作为重复行排除在外。

```
SELECT col1, col2
   FROM T1
   EXCEPT
      SELECT col1, col2
         FROM T2;
```

col1	col2
2	b
4	(NULL)

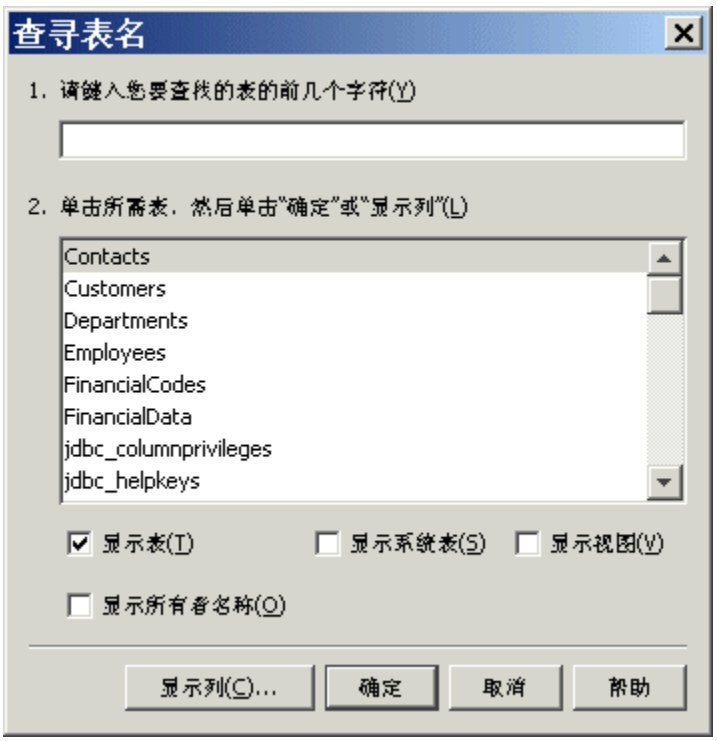
连接：从多个表检索数据

目录

在创建数据库时，您通过将特定于不同对象的信息放置在多个不同的表中，而不是放置在具有许多冗余条目的一个大表中来规范化数据。因此，从多个表检索相关数据时，使用 SQL JOIN 操作符执行连接操作。连接操作通过使用来自两个或多个表（或多个视图）的信息，重新创建更大的表。使用不同的连接，您可以构建多种此类虚拟表，每一个表都适合于执行某种特定任务。

显示一组表

在 Interactive SQL 中，按 F7 键显示所连接数据库中的表的列表。



选择一个表，然后单击 [**显示列**] 查看该表的列。按 Esc 键返回到表列表；再次按 Esc 键返回到 [**SQL 语句**] 窗格。按 Enter 键，将所选择的表或列名称复制到 [**SQL 语句**] 窗格中光标当前所在的位置。

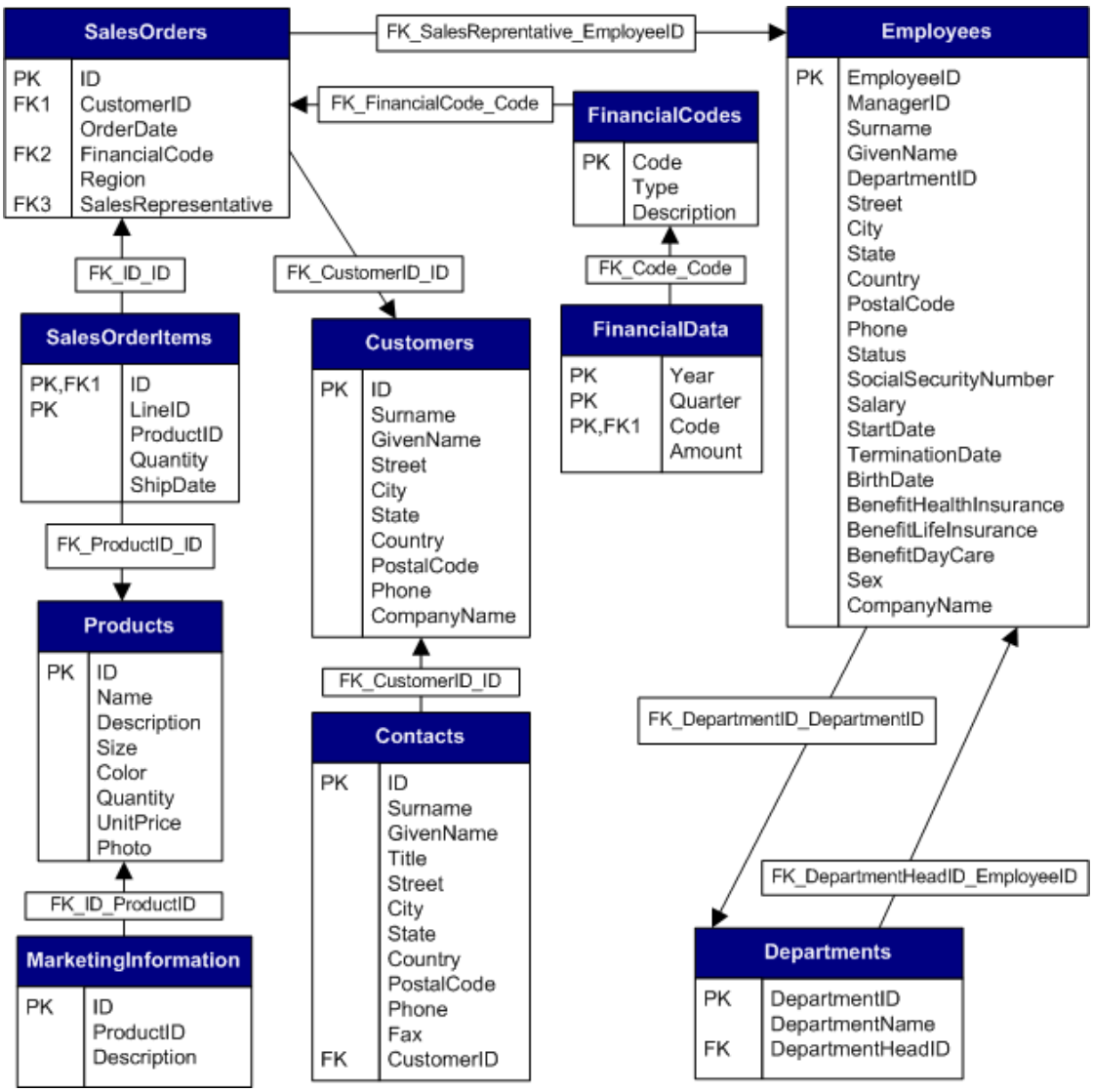
按 Esc 键可以离开该列表。

有关 SQL Anywhere 示例数据库中表的详细信息，请参见“教程：使用示例数据库” 《SQL Anywhere 服务器 - 数据库管理》。

示例数据库模式

下图显示 SQL Anywhere 示例数据库及其中用于关联这些表的外键的名称。某些高级连接需要这些外键角色名。

有关角色名的详细信息，请参见“在有多个外键关系时的键连接”一节第 392 页。



连接的工作原理

连接操作通过比较指定的列中的值来组合多个表中的行。本节概要介绍 SQL Anywhere 连接语法。

关系数据库将有关不同类型的对象的信息存储在不同的表中。例如，与雇员有关的信息在一个表中出现，而与部门有关的信息在另一个表中出现。Employees 表包含诸如雇员姓名和地址等有关信息。Departments 表包含有关一个部门的信息，例如部门名称和部门的领导。

大多数问题只能通过使用来自不同表的信息组合来回答。例如，要回答问题 "谁管理销售部门？"，可使用 Departments 表确定正确的雇员，然后在 Employees 表中查找雇员姓名。

连接是一种通过建立一个包含来自多个表的信息的新虚拟表来回答此类问题的方法。例如，您可以通过将 Employees 表中包含的信息与 Departments 表中包含的信息进行组合来创建部门领导的列表。您使用 FROM 子句指定哪些表包含所需信息。

若要使连接有效，您必须组合每个表的正确的列。为了列出部门领导，组合后的表中的每一行都应包含部门的名称以及管理该部门的雇员的姓名。通过指定连接操作的具体类型或使用 ON 子句，您可以控制在组合表中匹配列的方式。

另请参见

- “FROM 子句”一节 《SQL Anywhere 服务器 - SQL 参考》

FROM 子句

使用 FROM 子句指定要连接哪些基表、临时表、视图或派生表。FROM 子句可用于 SELECT 或 UPDATE 语句中。FROM 子句的简写语法如下：

FROM *table-expression*, ...

其中：

table-expression :
table-name
| *view-name*
| *derived-table-name*
| *lateral-derived-table-name*
| *join-expression*
| **(** *table-expression*, ... **)**
| *openstring-expression*
| *apply-expression*

table-name or *view-name*:
[*owner*.] *table-or-view-name* [[**AS**] *correlation-name*]

derived-table-name :
(*select-statement* **)** [**AS**] *correlation-name* [**(** *column-name*, ... **)**]

join-expression :
table-expression join-operator table-expression [**ON** *join-condition*]

join-operator:
[**KEY** | **NATURAL**] [*join-type*] **JOIN**
| **CROSS JOIN**

join-type:
INNER
| **FULL** [**OUTER**]
| **LEFT** [**OUTER**]
| **RIGHT** [**OUTER**]

apply-expression :
table-expression { **CROSS** | **OUTER** } **APPLY** *table-expression*

join-condition :

请参见“搜索条件”一节 《SQL Anywhere 服务器 - SQL 参考》。

注意

您不能将 ON 子句与 CROSS JOIN 一起使用。

有关语法的详细信息，请参见“FROM 子句”一节 《SQL Anywhere 服务器 - SQL 参考》。

连接条件

表可以使用**连接条件**进行连接。连接条件只不过是搜索条件。它基于列中各值间的关系从连接的各表中选择行的子集。例如，以下查询从 Products 和 SalesOrderItems 表中检索数据。

```
SELECT *
FROM Products JOIN SalesOrderItems
   ON Products.ID = SalesOrderItems.ProductID;
```

此查询中的连接条件为

```
Products.ID = SalesOrderItems.ProductID
```

此连接条件意味着，只有那些在两个表中具有同一产品 ID 的行才能组合在结果集中。

连接条件可以是显式连接条件或生成的连接条件。**显式连接条件**是放在 ON 子句或 WHERE 子句中的连接条件。以下查询使用 ON 子句。该查询生成两个表的矢量积（各行的所有组合），但不包括 ID 号不匹配的那些行。结果是一个客户列表，列表中具有这些客户的订单的详细信息。

```
SELECT *
FROM Customers
JOIN SalesOrders
ON SalesOrders.CustomerID = Customers.ID;
```

生成的连接条件是您指定 KEY JOIN 或 NATURAL JOIN 时自动创建的连接条件。在键连接的情况下，生成的连接条件基于各表间的外键关系。在自然连接的情况下，生成的连接条件基于具有相同名称的列。

提示
键连接语法和自然连接语法都是快捷方式：您只使用关键字 JOIN（无需 KEY 或 NATURAL），然后在 ON 子句中显式声明相同的连接条件，也可以获得相同的结果。

当您将 ON 子句与键连接或自然连接一起使用时，所使用的连接条件是显式指定的连接条件与生成的连接条件的**联合**。这意味着，这些连接条件用关键字 AND 组合在一起。

连接的表

SQL Anywhere 支持以下几类连接表。

- **交叉连接** 两个表的这种连接类型生成来自这两个表的各行的所有可能组合。结果集的规模是第一个表中的行数乘以第二个表中的行数。交叉连接也称作矢量积或笛卡儿积。您不能将 ON 子句与交叉连接一起使用。
- **键连接** 这种类型的连接条件使用各表间的外键关系。如果使用 JOIN 关键字时未指定连接类型（如 INNER、OUTER 等），并且没有 ON 子句，则缺省是键连接。
- **自然连接** 此连接是基于具有相同名称的列自动生成的。
- **使用 ON 子句连接** 此类型的连接由 ON 子句中的连接条件的显式说明产生。在与键连接或自然连接一起使用时，该连接条件既包含生成的连接条件，也包含显式连接条件。在与 JOIN 关键字一起使用（但不与 KEY 或 NATURAL 关键字一起使用）时，没有生成的连接条件。请参见“显式连接条件（ON 子句）”一节第 369 页。

内连接和外连接

可以通过指定 INNER、LEFT OUTER、RIGHT OUTER 或 FULL OUTER 对键连接、自然连接和使用 ON 子句的连接加以限定。缺省值是 INNER。在使用关键字 LEFT、RIGHT 或 FULL 时，关键字 OUTER 是可选的。

在内连接中，结果中的每一行都满足连接条件。

在左外连接或右外连接中，为一个表保留所有行，为另一个表不满足连接条件的行返回空值。例如，在右外连接中，保留右侧表的所有行，左侧的用空值补充。

在完全外连接中，保留这两个表的所有行；并且为不满足连接条件的行提供空值。

连接两个表

为了理解简单内连接的计算方式，请参见以下查询。它回答的问题是：哪些产品尺码的订购数量与库存数量相同？

```
SELECT DISTINCT Name, Size,
    SalesOrderItems.Quantity
FROM Products JOIN SalesOrderItems
ON Products.ID = SalesOrderItems.ProductID
  AND Products.Quantity = SalesOrderItems.Quantity;
```

name	Size	Quantity
Baseball Cap	One size fits all	12
Visor	One size fits all	36

该查询解释如下。请注意，这是对如何处理此查询的概念说明，用来阐释涉及连接的查询的语义。它不表示 SQL Anywhere 实际计算结果集的方式。

- 创建 Products 表和 SalesOrderItems 表的矢量积。矢量积包含来自这两个表的各行的所有组合。
- 排除产品 ID 不相同的所有行（由于使用了连接条件 `Products.ID = SalesOrderItems.ProductID`）。
- 排除数量不相同的所有行（由于使用了连接条件 `Products.Quantity = SalesOrderItems.Quantity`）。
- 创建具有以下三列的结果表：Products.Name、Products.Size 和 SalesOrderItems.Quantity。
- 排除所有重复行（由于 DISTINCT 关键字）。

有关如何计算外连接的说明，请参见“外连接”一节第 373 页。

连接两个以上的表

使用 SQL Anywhere，对您可以连接的表的数目没有固定限制。

在连接两个以上的表时，可以选择使用括号。如果您没有使用括号，SQL Anywhere 会按从左到右的顺序计算语句的值。因此，`A JOIN B JOIN C` 等效于 `( A JOIN B ) JOIN C`。同样，下面的两个 SELECT 语句是等效的：

```
SELECT *
FROM A JOIN B JOIN C JOIN D;

SELECT *
FROM ( ( A JOIN B ) JOIN C ) JOIN D;
```

只要连接两个以上的表，连接就会涉及表表达式。在 `A JOIN B JOIN C` 示例中，将表表达式 `A JOIN B` 连接到表 C。这意味着，从理论上讲，首先连接表 A 和表 B，然后结果被连接到 C。

如果表表达式中包含外连接，则连接的顺序是十分重要的。例如，`A JOIN B LEFT OUTER JOIN C` 被解释为 `(A JOIN B) LEFT OUTER JOIN C`。这意味着表表达式 `A JOIN B` 被连接到表 C。表表达式 `A JOIN B` 是保留的，表 C 是提供空值的。

有关外连接的详细信息，请参见“外连接”一节第 373 页。

有关 SQL Anywhere 如何执行表表达式的键连接的详细信息，请参见“表的表达式的键连接”一节第 394 页。

有关 SQL Anywhere 如何执行表表达式的自然连接的详细信息，请参见“表的表达式的自然连接”一节第 389 页。

连接兼容数据类型

在连接两个表时，您所比较的列必须具有相同的或兼容的数据类型。

有关连接中数据类型转换的详细信息，请参见“数据类型之间的比较”一节 《SQL Anywhere 服务器 - SQL 参考》。

在 delete、update 和 insert 语句中使用连接

可以在 DELETE、UPDATE、INSERT 和 SELECT 语句中使用连接。如果将 ansi_update_constraints 选项设置为 Off，则可以更新包含连接的某些游标。对于在 SQL Anywhere 版本 7 之前的版本中创建的数据库，Off 是缺省值。对于在版本 7 或更高版本中创建的数据库，缺省值为 Cursors。请参见“ansi_update_constraints 选项 [兼容性]”一节 《SQL Anywhere 服务器 - 数据库管理》。

非 ANSI 连接

SQL Anywhere 支持用于连接的 ISO/ANSI 标准。它还支持以下非标准连接：

- “Transact-SQL 外连接（*= 或 =*）”一节第 377 页
- “连接中的重复相关名（星形连接）”一节第 380 页
- “键连接”一节第 391 页
- “自然连接”一节第 387 页

您可以使用 REWRITE 函数查看非 ANSI 连接的等效 ANSI 连接。请参见“REWRITE 函数 [Miscellaneous]”一节 《SQL Anywhere 服务器 - SQL 参考》。

显式连接条件（ON 子句）

您可以单独使用显式连接条件而不用键连接或自然连接，也可以与键连接或自然连接一起使用。通过在紧随该连接之后插入 ON 子句来指定连接条件。该连接条件始终针对它前面的连接。ON 子句对连接中的行应用限制，其方式在很大程度上与 WHERE 子句对查询中的行应用限制相同。

与 CROSS JOIN 相比，可使用 ON 子句构建更为有用的连接。例如，您可以对 SalesOrders 表和 Employees 表的连接应用 ON 子句以便只检索特定的行，从而在检索结果的每一行中，SalesOrders 表中的 SalesRepresentative 与 Employees 表中的 SalesRepresentative 相同。这样，每行都包含有关订单和负责该订单的销售代表的信息。

例如，在下面的查询中，第一个 ON 子句用于将 SalesOrders 连接到 Customers。第二个 ON 子句用于将表表达式 (SalesOrders JOIN Customers) 连接到基表 SalesOrderItems。

```
SELECT *
FROM SalesOrders JOIN Customers
    ON SalesOrders.CustomerID = Customers.ID
  JOIN SalesOrderItems
    ON SalesOrderItems.ID = SalesOrders.ID;
```

在 ON 子句中引用表

在 ON 子句中引用的表必须是 ON 子句修饰的连接的一部分。例如，以下语句无效：

```
FROM ( A KEY JOIN B ) JOIN ( C JOIN D ON A.x = C.x )
```

问题在于：连接条件 `A.x = C.x` 引用表 A，而表 A 不是该连接条件修饰的连接（在此示例中为 `C JOIN D`）的一部分。

但是，从 ANSI/ISO 标准 SQL99 和 Adaptive Server Anywhere 7.0 开始，此规则有一个例外：如果您在表表达式之间使用逗号，则连接的 ON 条件可以引用 FROM 子句中在语法上位于该条件前面的表。因此，以下语句是有效的：

```
FROM (A KEY JOIN B) , (C JOIN D ON A.x = C.x)
```

有关逗号的详细信息，请参见“逗号”一节第 372 页。

示例

以下示例连接 SalesOrders 表与 Employees 表。结果中的每一行都反映 SalesOrders 表中 SalesRepresentative 列的值与 Employees 表中 EmployeeID 列的值匹配的行。

```
SELECT Employees.Surname, SalesOrders.ID, SalesOrders.OrderDate
FROM SalesOrders
JOIN Employees
ON SalesOrders.SalesRepresentative = Employees.EmployeeID;
```

Surname	ID	OrderDate
Chin	2008	4/2/2001

Surname	ID	OrderDate
Chin	2020	3/4/2001
Chin	2032	7/5/2001
Chin	2044	7/15/2000
Chin	2056	4/15/2001
...	...	...

下面是关于此示例的一些说明：

- 此查询的结果中仅包含 648 行（每行对应 SalesOrders 表中的一行）。在矢量积的 48,600 行中，只有其中的 648 行在两个表中具有相同的雇员编号。
- 结果的排序没有任何意义。您可以添加一个 ORDER BY 子句对查询强制特定的顺序。
- ON 子句包含没有在最终结果集中显示的列。

生成的连接和 ON 子句

如果使用 JOIN 关键字并且没有指定任何连接类型，则键连接是缺省设置—除非您使用 ON 子句。如果您将 ON 子句与未指定类型的 JOIN 一起使用，则缺省不是键连接，也没有其它生成的连接条件。

例如，以下是键连接，因为在使用 JOIN 关键字并且没有 ON 子句时键连接是缺省设置：

```
SELECT *
FROM A JOIN B;
```

以下是表 A 和表 B 之间的连接，并且具有连接条件 `A.x = B.y`。它不是键连接。

```
SELECT *
FROM A JOIN B ON A.x = B.y;
```

如果您指定 KEY JOIN 或 NATURAL JOIN 并且使用 ON 子句，则最终的连接条件是生成的连接条件与显式连接条件的联合。例如，以下语句有两个连接条件：一个由键连接生成的连接条件和一个在 ON 子句中显式指定的连接条件。

```
SELECT *
FROM A KEY JOIN B ON A.x = B.y;
```

如果由键连接生成的连接条件是 `A.w = B.z`，则以下语句是等效的：

```
SELECT *
FROM A JOIN B
  ON A.x = B.y
  AND A.w = B.z;
```

有关键连接的详细信息，请参见“键连接”一节第 391 页。

显式连接条件的类型

大多数连接条件都是基于等同性的，因此称作**等值连接**。例如，

```
SELECT *
FROM Departments JOIN Employees
   ON Departments.DepartmentID = Employees.DepartmentID;
```

但是，您并非一定要在连接条件中使用等号 (=)。您可以使用任何搜索条件，例如包含 LIKE、SOUNDEX、BETWEEN、>（大于）和 !=（不等于）的搜索条件。

示例

以下示例回答的问题是：哪些产品的订购数量大于库存数量？

```
SELECT DISTINCT Products.Name
FROM Products JOIN SalesOrderItems
ON Products.ID = SalesOrderItems.ProductID
   AND SalesOrderItems.Quantity > Products.Quantity;
```

有关搜索条件的详细信息，请参见“搜索条件”一节 《SQL Anywhere 服务器 - SQL 参考》。

使用 WHERE 子句用于连接条件

除了使用外连接的情况之外，您可以在 WHERE 子句中指定连接条件来代替 ON 子句。但是，您应该知道，如果查询包含外连接，则 WHERE 子句和 ON 子句两者之间可能存在语义差别。

ON 子句是 FROM 子句的一部分，因此在 WHERE 子句之前进行处理。除了外连接，结果通常不会不同。在外连接中，使用 WHERE 子句可能会将连接转换为内连接。

在决定是将连接条件放置于 ON 子句还是放置于 WHERE 子句中时，请牢记以下规则：

- 在指定外连接时，将连接条件放置到 WHERE 子句中可能会将外连接转换为内连接。

  有关 WHERE 子句和外连接的详细信息，请参见“外连接和连接条件”一节第 374 页。
- ON 子句中的条件只能引用由关联的 JOIN 连接的表表达式中的表。而 WHERE 子句中的条件可以引用任何表，即使这些表不是连接的一部分。
- 您不能将 ON 子句与关键字 CROSS JOIN 一起使用，但您始终可以使用 WHERE 子句。
- 在连接条件位于 ON 子句中时，键连接不是缺省值。但是，如果连接条件置于 WHERE 子句中，则键连接可以是缺省值。

  有关在什么条件下键连接是缺省值的详细信息，请参见“在键连接是缺省值时”一节第 391 页。

在本文档的一些示例中，将连接条件放置于 ON 子句中。在使用外连接的示例中，这是必需的。在其它一些例子中，这样做是为了更明显地表明它们是连接条件，而不是一般的搜索条件。

交叉连接

两个表的交叉连接生成来自这两个表的各行的所有可能组合。交叉连接也称作矢量积或笛卡儿积。

第一个表的每一行针对第二个表的每一行都出现一次。因此，结果集的行数等于第一个表的行数与第二个表的行数的乘积减去因 WHERE 子句中的限制而省略的行数。

您不能将 ON 子句与交叉连接一起使用。但是，您可以将限制放置于 WHERE 子句中。

内部修饰符和外部修饰符不适用于交叉连接

除了 WHERE 子句中存在的附加限制，两个表中的所有行都始终出现在交叉连接的结果集中。因此，INNER、LEFT OUTER 和 RIGHT OUTER 关键字不适用于交叉连接。

例如，以下语句连接两个表。

```
SELECT *
FROM A CROSS JOIN B;
```

来自此查询的结果集包括 A 中的所有列和 B 中的所有列。A 的任一行和 B 的任一行的每一组合在结果集中均有一行。如果 A 有 *n* 行，B 有 *m* 行，则查询返回 *n*x*m* 行。

逗号

逗号的工作原理类似于连接运算符，但并不完全相同。逗号所创建的矢量积与关键字 CROSS JOIN 所创建的矢量积完全相同。但是，连接关键字创建表表达式，而逗号则创建表表达式列表。

在以下两个表的简单内连接中，逗号和关键字 CROSS JOIN 是等效的：

```
SELECT *
FROM A CROSS JOIN B CROSS JOIN C
WHERE A.x = B.y;
```

和

```
SELECT *
FROM A, B, C
WHERE A.x = B.y;
```

通常，您可以使用逗号来代替关键字 CROSS JOIN。逗号语法大体上等效于交叉连接语法，只在表表达式中使用逗号时生成的连接条件情况下除外。

有关逗号如何用于生成的连接条件的信息，请参见“表的表达式的键连接”一节第 394 页。

在星形连接的语法中，逗号有特殊用途。有关详细信息，请参见“连接中的重复相关名（星形连接）”一节第 380 页。

内连接和外连接

关键字 INNER、LEFT OUTER、RIGHT OUTER 和 FULL OUTER 可用于修饰键连接、自然连接和使用 ON 子句的连接。缺省值是 INNER。这些修饰符不适用于交叉连接。

内连接

缺省情况下，连接是**内连接**。这意味着，只有满足连接条件的行才能包括在结果集中。

示例

例如，以下查询的结果集的每一行都包含满足键连接条件的、来自 Customers 的一行和来自 SalesOrders 的一行的信息。如果某特定客户没有下任何订单，则该客户不满足键连接条件，因而结果集中不包含对应于该客户的行。

```
SELECT GivenName, Surname, OrderDate
FROM Customers KEY INNER JOIN SalesOrders
ORDER BY OrderDate;
```

GivenName	Surname	OrderDate
Hardy	Mums	2000-01-02
Aram	Najarian	2000-01-03
Tommie	Wooten	2000-01-03
Alfredo	Margolis	2000-01-06
...	...	...

因为内连接和键连接是缺省值，所以，您可以使用以下 FROM 子句获得与上述相同的结果：

```
SELECT GivenName, Surname, OrderDate
FROM Customers JOIN SalesOrders
ORDER BY OrderDate;
```

外连接

通常，您创建的连接只返回满足连接条件的行；这些连接称为内连接，它们是查询时使用的缺省连接。但是，您有时可能想保留一个表中的所有行。若要实现此目的，您可以使用**外连接**。

两个表的左或右**外连接**将保留一个表的所有行，并为另一个表中不满足连接条件的行提供空值。**左外连接**保留左侧表中的每一行，**右外连接**保留右侧表中的每一行。在**完全外连接**中，保留两个表的所有行。

左外连接或右外连接的左右两侧的表表达式称作**保留的**和**提供空值的**。在左外连接中，左侧表表达式是保留的，右侧表表达式是提供空值的。

有关使用 Transact-SQL 语法创建外连接的信息，请参见“Transact-SQL 外连接（*= 或 =*）”一节 第 377 页。

示例

以下语句包括了所有客户。如果某个特定的客户没有下订单，则结果中对应于订单信息的每一列都包含 NULL 值。

```
SELECT Surname, OrderDate, City
FROM Customers LEFT OUTER JOIN SalesOrders
   ON Customers.ID = SalesOrders.CustomerID
WHERE Customers.State = 'NY'
ORDER BY OrderDate;
```

Surname	OrderDate	City
Thompson	(NULL)	Bancroft
Reiser	2000-01-22	Rockwood
Clarke	2000-01-27	Rockwood
Mentary	2000-01-30	Rockland
...	...	...

此语句中的外连接可解释如下。请注意，这只是概念上的说明，并不表示 SQL Anywhere 实际就是这样计算结果集的。

- 为客户下的每一份销售订单返回一行。由于为每一个销售订单都返回一行，因此，如果客户下了两个或两个以上的销售订单，就会返回多行。此结果与内连接的结果相同。使用 ON 条件来匹配客户和销售订单行。在此步骤中不使用 WHERE 子句。
- 为没有下任何销售订单的每一位客户包含一行。这确保包括 Customers 表中的每一行。对于所有这些行，都用空值填充来自 SalesOrders 表的列。因为使用了关键字 OUTER，所以加上这些行，如果是内连接就不会有这些行。在此步骤中既不使用 ON 条件也不使用 WHERE 子句。
- 使用 WHERE 子句，排除客户不居住在纽约的每一行。

外连接和连接条件

使用外连接时通常会出现连接条件放置错误。在大多数情况下，如果您在 WHERE 子句中对提供空值的表施加限制，则连接等效于内连接。

原因在于：在搜索条件的任何输入都为 NULL 时，大多数搜索条件无法计算为 TRUE（真）。对提供空值的表的 WHERE 子句限制将限制值与 NULL 进行比较，导致从结果集中删除相应的行。保留的表中的行不再保留，因此连接是内连接。

例外的是有些比较在输入都为 NULL 时的值为真。这些包括 IS NULL、IS UNKNOWN、IS FALSE、IS NOT TRUE 和涉及 ISNULL 或 COALESCE 的表达式。

示例

例如，以下语句计算左外连接。

```
SELECT *
FROM Customers KEY LEFT OUTER JOIN SalesOrders
   ON SalesOrders.OrderDate < '2000-01-03';
```

相比之下，以下语句创建内连接。

```
SELECT Surname, OrderDate
FROM Customers KEY LEFT OUTER JOIN SalesOrders
   WHERE SalesOrders.OrderDate < '2000-01-03';
```

这两个语句中的第一个语句可按如下说明来理解：首先，将 Customers 表左外连接到 SalesOrders 表。结果集中包括 Customers 表的所有行。对于在 2000 年 1 月 3 日之前没有任何订单的那些客户，用空值填充销售订单字段。

在第二条语句中，首先左外连接 Customers 和 SalesOrders。结果集中包括 Customers 表的所有行。对于没有任何订单的那些客户，用空值填充销售订单字段。接下来，通过只选择在其中客户自 2000 年 1 月 3 日以后已下了订单的那些行，应用 WHERE 条件。对于没有下任何订单的那些客户，这些值是空值。将任何值与空值进行比较将计算为 UNKNOWN（未知）。因此，这些行被删除，该语句简化为内连接。

有关搜索条件的详细信息，请参见“搜索条件”一节 《SQL Anywhere 服务器 - SQL 参考》。

理解复杂外连接

在查询包括使用外连接的表表达式时，连接的顺序是十分重要的。例如，`A JOIN B LEFT OUTER JOIN C` 被解释为 `(A JOIN B) LEFT OUTER JOIN C`。这意味着表表达式 `(A JOIN B)` 被连接到表 C。表表达式 `(A JOIN B)` 是保留的，表 C 是提供空值的。

考虑以下语句，在该语句中，A、B 和 C 是表：

```
SELECT *
FROM A LEFT OUTER JOIN B RIGHT OUTER JOIN C;
```

为了理解此语句，首先要记住，SQL Anywhere 是按从左到右的顺序对语句求值的，并且按以下所示添加括号。由此得到

```
SELECT *
FROM (A LEFT OUTER JOIN B) RIGHT OUTER JOIN C;
```

接下来，您可能想要将右外连接转换为左外连接，使这两个连接的类型相同。为此，只需调换右外连接中的表的位置，得到：

```
SELECT *
FROM C LEFT OUTER JOIN (A LEFT OUTER JOIN B);
```

对于嵌套的外连接，表 A 是保留的，表 B 是提供空值的。在第一个外连接中，表 C 是保留的。

您可按如下所示解释该连接：

- 将 A 连接到 B，保留 A 中的所有行。

- 接下来，将 C 与 A 和 B 的连接的结果相连接，保留 C 中的所有行。

该连接没有 ON 子句，因此缺省为键连接。SQL Anywhere 为此类型的连接生成连接条件的方法在“不包含逗号的表的表达式的键连接”一节第 395 页中予以说明。

此外，外连接的连接条件必须只包括以前已在 FROM 子句中引用的表。此限制依据 ANSI/ISO 标准，并且强制实施以避免出现多义性。例如，以下两个语句在语法上是不正确的，因为在引用表 C 之前已在连接条件中引用表 C。

```
SELECT *
FROM (A LEFT OUTER JOIN B ON B.x = C.x) JOIN C;
```

和

```
SELECT *
FROM A LEFT OUTER JOIN B ON A.x = C.x, C;
```

视图和派生表的外连接

还可为视图和派生表指定外连接。

语句

```
SELECT *
FROM V LEFT OUTER JOIN A ON (V.x = A.x);
```

可解释如下：

- 计算视图 V。
- 通过保留来自 V 的所有行并且使用连接条件 `V.x = A.x`，将计算出的视图 V 中的所有行与 A 相连接。

示例

以下示例定义一个称为 V 的视图，该视图返回收入超过 $60000 的女雇员的雇员 ID 和部门名称。

```
CREATE VIEW V AS
SELECT Employees.EmployeeID, DepartmentName
  FROM Employees JOIN Departments
    ON Employees.DepartmentID = Departments.DepartmentID
  WHERE Sex = 'F' and Salary > 60000;
```

接着使用该视图添加这些女雇员的工作部门和销售区域的列表。视图 V 会保留下来，SalesOrders 是提供空值的。

```
SELECT DISTINCT V.EmployeeID, Region, V.DepartmentName
  FROM V LEFT OUTER JOIN SalesOrders
    ON V.EmployeeID = SalesOrders.SalesRepresentative;
```

EmployeeID	Region	DepartmentName
243	(NULL)	R & D

EmployeeID	Region	DepartmentName
316	(NULL)	R & D
529	(NULL)	R & D
902	Eastern	Sales
...	...	...

Transact-SQL 外连接（*= 或 =*）

注意
不建议使用 Transact-SQL 外连接运算符 *= 和 =*，在以后的版本中将不再提供支持。

与 ANSI/ISO SQL 标准相符，SQL Anywhere 支持 LEFT OUTER、RIGHT OUTER 和 FULL OUTER 关键字。为与 Adaptive Server Enterprise 版本 12 以前的版本兼容，只要 tsql_outer_joins 数据库选项设置为 On，SQL Anywhere 也支持与这些关键字相对应的 Transact-SQL 运算符 *= 和 =*。请参见“tsql_outer_joins 选项 [兼容性]”一节《SQL Anywhere 服务器 - 数据库管理》。

有一些与 Transact-SQL 语义有关的限制和可能存在的问题。有关 Transact-SQL 外连接的详细论述，请参见白皮书 Semantics and Compatibility of Transact-SQL Outer Joins，您可从以下网址获得该白皮书 http://www.sybase.com/detail?id=1017447。

在 Transact-SQL 方言中，您通过在 FROM 子句中提供以逗号分隔的表列表并在 WHERE 子句中使用特殊运算符 *= 或 =* 来创建外连接。在 Adaptive Server Enterprise 版本 12 以前的版本中，连接条件必须出现在 WHERE 子句中（不支持 ON 子句）。

小心
创建外连接时，不要将 *= 语法与 ON 子句语法混合使用。这同样适用于在查询中引用的视图。

示例

以下左外连接列出所有客户并查找这些客户的订单日期（如果有）：

```
SELECT GivenName, Surname, OrderDate
FROM Customers, SalesOrders
WHERE Customers.ID *= SalesOrders.CustomerID
ORDER BY OrderDate;
```

该语句等效于以下语句，后者使用了 ANSI/ISO 语法：

```
SELECT GivenName, Surname, OrderDate
FROM Customers LEFT OUTER JOIN SalesOrders
ON Customers.ID = SalesOrders.CustomerID
ORDER BY OrderDate;
```

Transact-SQL 外连接限制

注意
不建议使用 Transact-SQL 外连接运算符 *= 和 =*，在以后的版本中将不再提供支持。

对于 Transact-SQL 外连接有若干限制：

- 如果您指定了某一外连接并对来自该外连接的提供空值的表的某一列指定了限定，则结果可能不是您预期的结果。查询中的限定不从结果集中排除行，而是影响结果集的这些行中出现的值。对于不满足限定的行，在提供空值的表中显示 NULL 值。
- 您不能在一个查询中混合使用 ANSI/ISO SQL 语法和 Transact-SQL 外连接语法。如果视图是使用用于某一外连接的一个方言定义的，则必须使用同一方言来用于对该视图的任何外连接查询。
- 提供空值的表不能同时涉及 Transact-SQL 外连接和常规连接，或同时涉及两个外连接。例如，不允许以下 WHERE 子句，因为表 S 违反上述限制。

  ```
  WHERE R.x *= S.x
  AND S.y = T.y
  ```

  如果您无法重写查询以避免在外连接和常规连接子句中都使用同一个表，则必须将语句分成两个单独的查询，或者只使用 ANSI/ISO SQL 语法。
- 如果连接条件涉及外连接的提供空值的表，则您不能使用包含这样的连接条件的子查询。例如，不允许使用以下 WHERE 子句：

  ```
  WHERE R.x *= S.y
  AND EXISTS ( SELECT *
               FROM T
               WHERE T.x = S.x )
  ```

将视图与 Transact-SQL 外连接一起使用

如果您用外连接定义一个视图，然后查询该视图并且对来自该外连接的提供空值的表的列加以限定，则结果可能不是您预期的结果。该查询返回来自提供空值的表的所有行。对于不满足限定的行，在这些行的相应列中显示 NULL 值。

以下规则确定您通过包含外连接的视图对列可以执行哪些类型的更新：

- 对外连接视图不允许使用 INSERT 和 DELETE 语句。
- 对外连接视图允许使用 UPDATE 语句。如果该视图定义了 WITH CHECK 选项，若任何受影响的列出现在 WHERE 子句的表达式中，且该表达式包括来自多个表的列，则更新会失败。

空值是如何影响 Transact-SQL 连接的

在 Transact-SQL 外连接中，所连接的表或视图中的 NULL 值永远不会彼此匹配。一个 NULL 值与另一个 NULL 值比较的结果是 FALSE（假）。

专用连接

本节介绍如何创建某些专用连接，例如自连接、星形连接和使用派生表的连接。

自连接

在**自连接**中，表通过引用使用不同相关名的同一个表，与其自身连接。

示例 1

以下自连接生成雇员对列表。每一个雇员姓名与各雇员姓名彼此组合在一起显示。

```
SELECT a.GivenName, a.Surname,
       b.GivenName, b.Surname
FROM Employees AS a CROSS JOIN Employees AS b;
```

GivenName	Surname	GivenName	Surname
Fran	Whitney	Fran	Whitney
Fran	Whitney	Matthew	Cobb
Fran	Whitney	Philip	Chin
Fran	Whitney	Julie	Jordan
...	...	...	...

因为 Employees 表有 75 行，所以此连接包含 75 x 75 = 5625 行。它还包括将每一雇员与他们自己一起列出的行。例如，它包括以下行

GivenName	Surname	GivenName	Surname
Fran	Whitney	Fran	Whitney

如果您想要排除包含同一姓名两次的行，则添加一个连接条件，指示雇员 ID 应互不相同。

```
SELECT a.GivenName, a.Surname,
       b.GivenName, b.Surname
FROM Employees AS a CROSS JOIN Employees AS b
WHERE a.EmployeeID != b.EmployeeID;
```

如果没有这些重复行，该连接包含 75 x 74 = 5550 行。

该新连接包含将每个雇员与其他每个雇员配对的行，但因为每个姓名对可以按两种可能的顺序出现，所以每一对出现两次。例如，以上连接的结果包含以下两行。

GivenName	Surname	GivenName	Surname
Matthew	Cobb	Fran	Whitney

GivenName	Surname	GivenName	Surname
Fran	Whitney	Matthew	Cobb

如果姓名的顺序不重要，则可以生成包含 (75 x 74)/2 = 2775 个唯一对的列表。

```
SELECT a.GivenName, a.Surname,
      b.GivenName, b.Surname
FROM Employees AS a CROSS JOIN Employees AS b
WHERE a.EmployeeID < b.EmployeeID;
```

此语句通过只选择雇员 a 的 EmployeeID 小于雇员 b 的 EmployeeID 的那些行，消除了重复行。

示例 2

下面的自连接使用相关名 report 和 manager 来区分 Employees 表的两个实例，并且创建雇员及其经理的列表。

```
SELECT report.GivenName, report.Surname,
   manager.GivenName, manager.Surname
FROM Employees AS report JOIN Employees AS manager
   ON (report.ManagerID = manager.EmployeeID)
ORDER BY report.Surname, report.GivenName;
```

此语句生成以下部分显示的结果。雇员姓名出现在左侧的两列中，其经理的姓名在右侧显示。

GivenName	Surname	GivenName	Surname
Alex	Ahmed	Scott	Evans
Joseph	Barker	Jose	Martinez
Irene	Barletta	Scott	Evans
Jeannette	Bertrand	Jose	Martinez
...	...	...	...

连接中的重复相关名（星形连接）

使用重复表名称的原因是为了创建**星形连接**。在星形连接中，一个表或视图与若干其它表或视图相连接。

若要创建星形连接，应在 FROM 子句中多次使用同一个表名称、视图名称或相关名。这是对 ANSI/ISO SQL 标准的扩展。能够使用重复名并不添加任何附加功能，但这可以使某些查询写起来更加容易。

为使语法有意义，重复名必须位于不同的连接中。如果在同一连接中使用两次表名称或视图名称，则忽略第二个实例。例如，`FROM A,A` 和 `FROM A CROSS JOIN A` 均解释为 `FROM A`。

在 SQL Anywhere 中，以下示例是有效的，其中 A、B 和 C 是表。在此示例中，表 A 的同一实例同时连接到 B 和 C。请注意，在星形连接中需要使用逗号分隔这些连接。在星形连接中使用逗号是星形连接语法所特有的。

```
SELECT *
FROM A LEFT OUTER JOIN B ON A.x = B.x,
     A LEFT OUTER JOIN C ON A.y = C.y;
```

下一个示例是等效的。

```
SELECT *
FROM A LEFT OUTER JOIN B ON A.x = B.x,
     C RIGHT OUTER JOIN A ON A.y = C.y;
```

这两个示例都等效于以下标准 ANSI/ISO 语法。（括号是可选的。）

```
SELECT *
FROM (A LEFT OUTER JOIN B ON A.x = B.x)
LEFT OUTER JOIN C ON A.y = C.y;
```

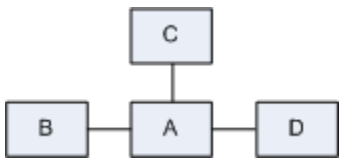
在下一个示例中，将表 A 连接到三个表：B、C 和 D。

```
SELECT *
FROM A JOIN B ON A.x = B.x,
     A JOIN C ON A.y = C.y,
     A JOIN D ON A.w = D.w;
```

这等效于以下标准 ANSI/ISO 语法。（括号是可选的。）

```
SELECT *
FROM ((A JOIN B ON A.x = B.x)
JOIN C ON A.y = C.y)
JOIN D ON A.w = D.w;
```

遇到复杂连接时，画图会很有帮助。上面的示例可通过下图描述，该图示阐释表 B、C 和 D 通过表 A 连接。



注意

只有 extended_join_syntax 选项为 On（缺省值）时，才能使用重复表名称。

有关详细信息，请参见“extended_join_syntax 选项 [数据库]”一节 《SQL Anywhere 服务器 - 数据库管理》。

示例 1

创建给 Rollin Overbey 下订单的客户名称的列表。请注意，FROM 子句中的表 Employees 的任何列均未出现在结果中。所连接的任何列—例如 Customers.ID 或 Employees.EmployeeID—也未出现在结果中。不过，只有在 FROM 子句中使用 Employees 表才能实现该连接。

```
SELECT Customers.GivenName, Customers.Surname,
   SalesOrders.OrderDate
```

```
FROM   SalesOrders KEY JOIN Customers,
       SalesOrders KEY JOIN Employees
WHERE Employees.GivenName = 'Rollin'
  AND Employees.Surname = 'Overbey'
ORDER BY SalesOrders.OrderDate;
```

GivenName	Surname	OrderDate
Tommie	Wooten	2000-01-03
Michael	Agliori	2000-01-08
Salton	Pepper	2000-01-17
Tommie	Wooten	2000-01-23
...	...	...

以下是采用标准 ANSI/ISO 语法的等效语句：

```
SELECT Customers.GivenName, Customers.Surname,
  SalesOrders.OrderDate
FROM SalesOrders JOIN Customers
  ON SalesOrders.CustomerID =
   Customers.ID
JOIN Employees
  ON SalesOrders.SalesRepresentative =
   Employees.EmployeeID
WHERE Employees.GivenName = 'Rollin'
  AND Employees.Surname = 'Overbey'
ORDER BY SalesOrders.OrderDate;
```

示例 2

本示例回答的问题是：每位客户订购每种产品的数量是多少？谁是获得订单的销售人员的经理？

为回答这些问题，应首先列出您需要检索的信息。在此例子中，需要检索的信息是产品、数量、客户名称和经理姓名。接下来，列出保存这些信息的表。它们是 Products、SalesOrderItems、Customers 和 Employees。查看 SQL Anywhere 示例数据库的结构时（请参见“示例数据库模式”一节 第 363 页），您将注意到，这些表全部通过 SalesOrders 表进行关联。您可以为 SalesOrders 表创建星形连接，以检索其它表中的信息。

此外，您还需要创建自连接，以便获取经理姓名，因为 Employees 表包含经理的 ID 号和所有雇员的姓名，但没有仅列出经理姓名的列。有关详细信息，请参见“自连接”一节第 379 页。

以下语句围绕 SalesOrders 表创建星形连接。这些连接全都是外连接，因此结果集将包括所有客户。有些客户没有下订单，因此对应这些客户的其它值为空。结果集中的列是客户、产品、订购的数量以及销售人员的经理的姓名。

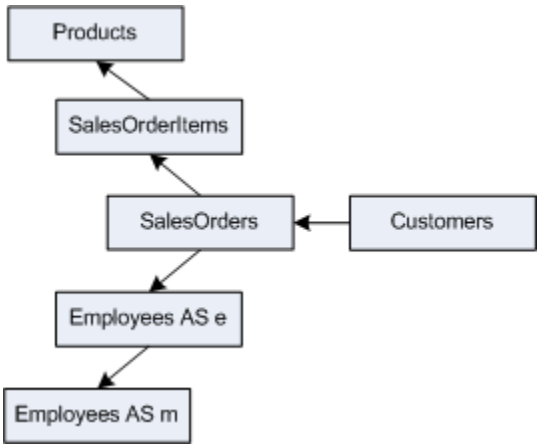
```
SELECT Customers.GivenName, Products.Name,
  SUM(SalesOrderItems.Quantity), m.GivenName
FROM SalesOrders
    KEY RIGHT OUTER JOIN Customers,
   SalesOrders
    KEY LEFT OUTER JOIN SalesOrderItems
    KEY LEFT OUTER JOIN Products,
```



```
      SalesOrders
    KEY LEFT OUTER JOIN Employees AS e
    LEFT OUTER JOIN Employees AS m
        ON (e.ManagerID = m.EmployeeID)
WHERE Customers.State = 'CA'
GROUP BY Customers.GivenName, Products.Name, m.GivenName
ORDER BY SUM(SalesOrderItems.Quantity) DESC,
   Customers.GivenName;
```

GivenName	Name	SUM(SalesOrderItems.Quantity)	GivenName
Sheng	Baseball Cap	240	Moira
Laura	Tee Shirt	192	Moira
Moe	Tee Shirt	192	Moira
Leilani	Sweatshirt	132	Moira
...	...	...	...

以下是此星形连接中各表的图示。箭头指示这些外连接的方向（左或右）。正如您所看到的，整个客户列表是贯穿所有连接来维护的。



以下标准 ANSI/ISO 语法等效于示例 2 中的星形连接。

```
SELECT Customers.GivenName, Products.Name,
  SUM(SalesOrderItems.Quantity), m.GivenName
FROM SalesOrders LEFT OUTER JOIN SalesOrderItems
    ON SalesOrders.ID = SalesOrderItems.ID
  LEFT OUTER JOIN Products
    ON SalesOrderItems.ProductID = Products.ID
  LEFT OUTER JOIN Employees as e
    ON SalesOrders.SalesRepresentative = e.EmployeeID
  LEFT OUTER JOIN Employees as m
    ON e.ManagerID = m.EmployeeID
  RIGHT OUTER JOIN Customers
    ON SalesOrders.CustomerID = Customers.ID
WHERE Customers.State = 'CA'
GROUP BY Customers.GivenName, Products.Name, m.GivenName
```

```
ORDER BY SUM(SalesOrderItems.Quantity) DESC,
   Customers.GivenName;
```

涉及派生表的连接

派生表使您能够在 FROM 子句中嵌套查询。使用派生表，您可以对组进行分组或构造与组之间的连接，而无需创建单独的视图或表并与组连接

在以下示例中，内部 SELECT 语句（用括号括起来）创建一个派生表，该派生表是按客户 ID 值进行分组的。外部 SELECT 语句为此表指派相关名 sales_order_counts，并使用连接条件将它与 Customers 表相连接。

```
SELECT Surname, GivenName, number_of_orders
FROM Customers JOIN
   (  SELECT CustomerID, COUNT(*)
      FROM SalesOrders
      GROUP BY CustomerID  )
   AS sales_order_counts ( CustomerID, number_of_orders )
   ON ( Customers.ID = sales_order_counts.CustomerID )
WHERE number_of_orders > 3;
```

结果得到所下订单超过三个的那些客户的姓名的表，并包括每位客户所下订单的数目。

有关派生表的键连接的说明，请参见“视图和派生表的键连接”一节第 398 页。

有关派生表的自然连接的说明，请参见“视图和派生表的自然连接”一节第 390 页。

有关派生表的外连接的说明，请参见“视图和派生表的外连接”一节第 376 页。

从 apply 表达式生成的连接

apply 表达式是一种指定右侧取决于左侧的连接的简单方式。例如，使用一个 apply 表达式来计算过程或派生表，在表表达式中每行计算一次。apply 表达式位于 SELECT 语句的 FROM 子句中，不允许使用 ON 子句。

APPLY 组合多个源的行，类似于 JOIN，只不过不能为 APPLY 指定 ON 条件。APPLY 和 JOIN 之间的主要差别在于 APPLY 的右侧可以根据左侧的当前行发生改变。针对左侧各行，重新计算右侧，生成的行与左侧的行连接。在左侧的行在右侧返回多个行的情况下，右侧返回多少行，左侧就在结果中重复多少次。

您可以指定两种类型的 APPLY：CROSS APPLY 和 OUTER APPLY。CROSS APPLY 仅在左侧返回可在右侧产生结果的行。OUTER APPLY 返回所有 CROSS APPLY 返回的行，还在左侧返回不能在右侧返回行的所有行（通过为右侧提供 NULL）。

apply 表达式的语法如下：

table-expression { **CROSS** | **OUTER** } **APPLY** *table-expression*

示例

下面的示例创建一个过程 EmployeesWithHighSalary，将部门 ID 作为输入，并返回在该部门中薪水大于 $80,000 的所有雇员姓名。


```
CREATE PROCEDURE EmployeesWithHighSalary( IN dept INTEGER )
  RESULT ( Name LONG VARCHAR )
   BEGIN
    SELECT E.GivenName || ' ' || E.Surname
        FROM Employees E
        WHERE E.DepartmentID = dept AND E.Salary > 80000;
   END;
```

以下查询使用 OUTER APPLY 来连接 Departments 表和 EmployeesWithHighSalary 过程的结果，并返回在各部门中薪水大于 $80,000 的所有雇员姓名。该查询返回的行在右侧具有 NULL，表示在相应的部门中没有薪水超过 $80,000 的雇员。

```
SELECT D.DepartmentName, HS.Name
    FROM Departments D
    OUTER APPLY EmployeesWithHighSalary( D.DepartmentID ) AS HS;
```

DepartmentName	Name
R & D	Kim Lull
R & D	David Scott
R & D	John Sheffield
Sales	Moira Kelly
Finance	Mary Anne Shea
Marketing	NULL
Shipping	NULL

下一查询使用 CROSS APPLY 来连接 Departments 表和 EmployeesWithHighSalary 过程的结果。注意，不包括右侧具有 NULL 的行。

```
SELECT D.DepartmentName, HS.Name
    FROM Departments D
    CROSS APPLY EmployeesWithHighSalary( D.DepartmentID ) AS HS;
```

DepartmentName	名称
R & D	Kim Lull
R & D	David Scott
R & D	John Sheffield
Sales	Moira Kelly
Finance	Mary Anne Shea

下一查询返回与上一查询相同的结果，但使用派生表作为 CROSS APPLY 的右侧。

```
SELECT D.DepartmentName, HS.Name
   FROM Departments D
    CROSS APPLY (
        SELECT E.GivenName || ' ' || E.Surname
            FROM Employees E
            WHERE E.DepartmentID = D.DepartmentID AND E.Salary > 80000
    ) HS( Name );
```

另请参见

- “FROM 子句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “交叉连接”一节第 372 页
- “内连接和外连接”一节第 373 页

自然连接

在您指定自然连接时，SQL Anywhere 基于名称相同的列生成连接条件。因此，要使用基表的自然连接，必须至少有一对具有相同名称的列，每个表一列。如果没有公用列名称，将发出错误消息。

如果表 A 和表 B 有一个共同的列名称，并且此列名为 x，则

```
SELECT *
FROM A NATURAL JOIN B;
```

等效于以下语句：

```
SELECT *
FROM A JOIN B
 ON A.x = B.x;
```

如果表 A 和表 B 有两个共同的列名称，并且它们分别名为 a 和 b，则 `A NATURAL JOIN B` 等效于以下语句：

```
A JOIN B
 ON A.a = B.a
 AND A.b = B.b;
```

示例 1

例如，您可以使用自然连接来连接 Employees 表和 Departments 表，因为这两个表有一个共同的列名称：DepartmentID 列。

```
SELECT GivenName, Surname, DepartmentName
FROM Employees NATURAL JOIN Departments
ORDER BY DepartmentName, Surname, GivenName;
```

GivenName	Surname	DepartmentName
Janet	Bigelow	Finance
Kristen	Coe	Finance
James	Coleman	Finance
Jo Ann	Davidson	Finance
...	...	...

以下语句是等效的。该语句显式指定在以上示例中生成的连接条件。

```
SELECT GivenName, Surname, DepartmentName
FROM Employees JOIN Departments
  ON (Employees.DepartmentID = Departments.DepartmentID)
ORDER BY DepartmentName, Surname, GivenName;
```

示例 2

在 Interactive SQL 中，执行以下查询：

```
SELECT Surname, DepartmentName
FROM Employees NATURAL JOIN Departments;
```

Surname	DepartmentName
Whitney	R & D
Cobb	R & D
Breault	R & D
Shishov	R & D
Driscoll	R & D
...	...

SQL Anywhere 查看这两个表，并确定这两个表仅有一个共同的列名称 DepartmentID。在内部将生成以下 ON 子句以用来执行该连接：

```
FROM Employees JOIN Departments
   ON Employees.DepartmentID = Departments.DepartmentID
```

NATURAL JOIN 正是输入 ON 子句的快捷方式；这两个查询是等同的。

使用 NATURAL JOIN 时出错

NATURAL JOIN 运算符可能会使得您不希望等同的列相等，从而引起问题。例如，以下查询生成不想要的结果：

```
SELECT *
FROM SalesOrders NATURAL JOIN Customers;
```

此查询的结果没有任何行。SQL Anywhere 内部生成以下 ON 子句：

```
FROM SalesOrders JOIN Customers
   ON SalesOrders.ID = Customers.ID
```

SalesOrders 表中的 ID 列是订单的 ID 号。Customers 表中的 ID 列是客户的 ID 号。没有一个 ID 编号匹配。当然，即使有匹配的编号，也没有任何意义。

自然连接与 ON 子句一起使用

如果您指定了 NATURAL JOIN 并将连接条件置于 ON 子句中，则结果将是这两个连接条件的联合。

例如，以下两个查询是等效的。在第一个查询中，SQL Anywhere 生成连接条件 `Employees.DepartmentID = Departments.DepartmentID`。该查询还包含一个显式连接条件。


```
SELECT GivenName, Surname, DepartmentName
FROM Employees NATURAL JOIN Departments
  ON Employees.ManagerID = Departments.DepartmentHeadID;
```

下一个查询是等效的。在该查询中，上一个查询中生成的自然连接条件是在 ON 子句中指定。

```
SELECT GivenName, Surname, DepartmentName
FROM Employees JOIN Departments
  ON Employees.ManagerID = Departments.DepartmentHeadID
   AND Employees.DepartmentID = Departments.DepartmentID;
```

表的表达式的自然连接

当自然连接至少有一侧有多个表表达式时，SQL Anywhere 通过比较连接运算符两侧的列集合，查找同名的列，由此来生成连接条件。

例如，在下列语句中

```
SELECT *
FROM (A JOIN B) NATURAL JOIN (C JOIN D);
```

有两个表表达式。表表达式 `A JOIN B` 中的列名称与表表达式 `C JOIN D` 中的列名称进行比较，然后为每个明确的匹配列名称对生成一个连接条件。**明确的匹配列对**意味着列名称在两个表表达式中均出现，但不在同一个表表达式中出现两次。

如果有不明确的列名称对，将发出错误消息。但是，列名称可以在同一个表表达式中出现两次，条件是它还不匹配另一个表表达式中的列名称。

列表的自然连接

在表的表达式列表位于自然连接的至少一侧时，为列表中的每一表的表达式生成单独的连接条件。

考虑以下表：

- 表 A 包含名为 a、b 和 c 的列
- 表 B 包含名为 a 和 d 的列
- 表 C 包含名为 d 和 c 的列

在本例中，`(A,B) NATURAL JOIN C` 连接可使 SQL Anywhere 生成两个连接条件：

```
ON A.c = C.c
 AND B.d = C.d
```

如果 A-C 之间或 B-C 之间没有公用列名称，将发出错误消息。

如果表 C 包含列 a、d 和 c，则连接 `(A,B) NATURAL JOIN C` 无效。原因在于：列 a 出现在所有三个表中，因此连接是不明确的。

示例

以下示例回答的问题是：提供每一笔销售的相关信息，例如销售的产品及销售人员？

```
SELECT *
FROM ( Employees KEY JOIN SalesOrders )
  NATURAL JOIN ( SalesOrderItems KEY JOIN Products );
```

这等效于

```
SELECT *
FROM ( Employees KEY JOIN SalesOrders )
  JOIN ( SalesOrderItems KEY JOIN Products )
    ON SalesOrders.ID = SalesOrderItems.ID;
```

视图和派生表的自然连接

对 ANSI/ISO SQL 标准的扩展是：您可以在自然连接的两侧指定视图或派生表。在以下语句中，

```
SELECT *
FROM View1 NATURAL JOIN View2;
```

将 View1 中的列与 View2 中的列进行比较。例如，如果在这两个视图中都发现名为 EmployeeID 的列，并且再没有其它同名的列，则生成的连接条件是 `(View1.EmployeeID = View2.EmployeeID)`。

示例

以下示例说明在自然连接中使用的视图可以包括表达式，而不只是包括列，并且采用与自然连接相同的方式处理它们。首先，创建包含名为 x 的列的视图 V，如下所示：

```
CREATE VIEW V(x) AS
SELECT R.y + 1
FROM R;
```

接下来，创建该视图与派生表的自然连接。该派生表具有相关名 T 及名为 x 的列。

```
SELECT *
FROM V NATURAL JOIN (SELECT P.y FROM P) as T(x);
```

此连接等效于以下语句：

```
SELECT *
FROM V JOIN (SELECT P.y FROM P) as T(x) ON (V.x = T.x);
```


键连接

许多常见的连接建立在两个由外键关联的表之间。最常见的连接限制外键值与主键值相等。KEY JOIN 运算符基于外键关系连接两个表。换句话说，SQL Anywhere 生成一个 ON 子句，该子句使一个表的主键列与另一个表的外键列相等。若要使用键连接，表和表之间必须有外键关系，否则将发出错误消息。

键连接可视为 ON 子句的快捷方式；这两个查询是相同的。但是，也可以将 ON 子句与 KEY JOIN 一起使用。当您指定了 JOIN 但未指定 CROSS、NATURAL、KEY 或使用 ON 子句时，缺省使用键连接。如果您查看 SQL Anywhere 示例数据库的图示，表之间的连线表示外键。您可以在图示中两个表通过连线连接的任何地方使用 KEY JOIN 运算符。有关 SQL Anywhere 示例数据库的详细信息，请参见“教程：使用示例数据库” 《SQL Anywhere 服务器 - 数据库管理》。

在键连接是缺省值时

在 SQL Anywhere 中，在满足以下所有条件时键连接是缺省值：

- 使用关键字 JOIN。
- 未指定关键字 CROSS、NATURAL 或 KEY。
- 没有 ON 子句。

示例

例如，以下查询基于数据库中的外键关系连接表 Products 和表 SalesOrderItems。

```
SELECT *
FROM Products KEY JOIN SalesOrderItems;
```

下一个查询是等效的。它省去了 KEY 一词，但缺省情况下，没有 ON 子句的 JOIN 是 KEY JOIN。

```
SELECT *
FROM Products JOIN SalesOrderItems;
```

下面这个查询也是等效的，因为在 ON 子句中指定的连接条件正好与 SQL Anywhere 基于 SQL Anywhere 示例数据库中表的外键关系为这些表生成的连接条件相同。

```
SELECT *
FROM Products JOIN SalesOrderItems
ON SalesOrderItems.ProductID = Products.ID;
```

键连接与 ON 子句一起使用

如果您指定了 KEY JOIN 并将连接条件置于 ON 子句中，则结果将是这两个连接条件的联合。例如，

```
SELECT *
FROM A KEY JOIN B
ON A.x = B.y;
```

如果由 A 和 B 的键连接生成的连接条件是 `A.w = B.z`，则该查询等效于

```
SELECT *
FROM A JOIN B
ON A.x = B.y AND A.w = B.z;
```

在有多个外键关系时的键连接

SQL Anywhere 尝试基于外键关系生成连接条件时，有时会找到有多个关系。在这些情况下，SQL Anywhere 通过将外键的角色名与该外键引用的主键表的相关名相匹配，确定要使用的外键关系。

以下几节介绍 SQL Anywhere 如何为键连接生成连接条件。在“描述键连接的操作的规则”一节 第 400 页中对这些信息进行了总结。

相关名和角色名

相关名是在查询的 FROM 子句中使用的表或视图的名称—它的初始名称或者在 FROM 子句中定义的别名。

角色名是外键的名称。对于给定的外（子）表，角色名必须是唯一的。

如果您没有为外键指定角色名，则按如下所示指派角色名：

- 如果没有任何外键的名称与主表名称相同，则将该主表名称指派为角色名。
- 如果主表名称已由其它外键使用，则角色名将由该主表名称后接用零填充的三位数字组合而成（对于外表是唯一的）。

如果您不知道外键的角色名，则可以通过在 Sybase Central 的左窗格中展开数据库容器来查看。在左窗格中选择表，然后在右窗格中单击 [**约束**] 选项卡。表的外键列表会显示在右窗格中。

有关包括 SQL Anywhere 示例数据库中所有外键的角色名的图示，请参见“示例数据库模式”一节 第 363 页。

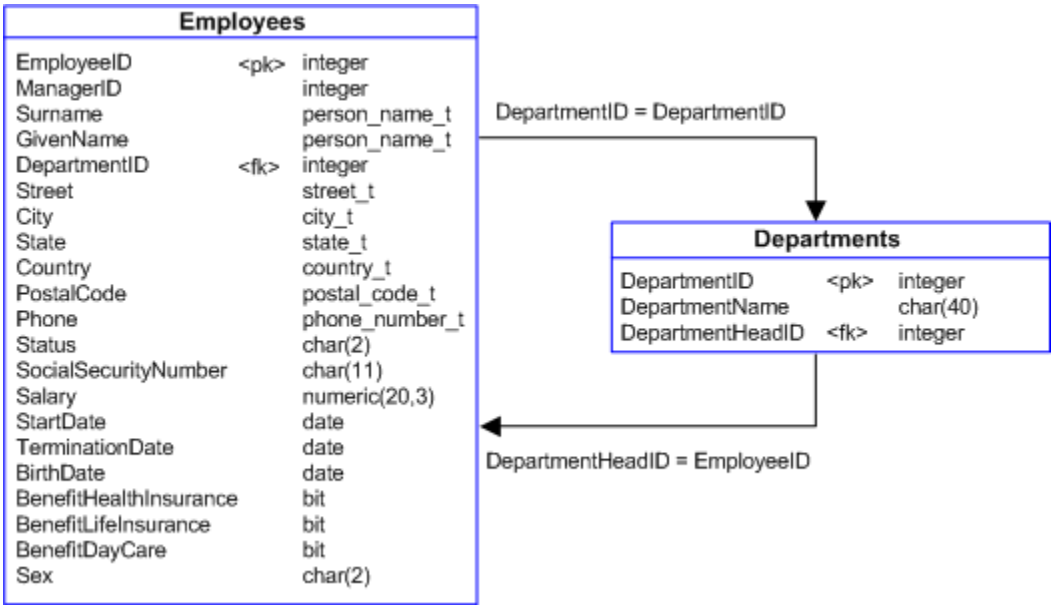
生成连接条件

SQL Anywhere 查找其角色名与主键表的相关名相同的外键：

- 如果恰好具有一个与连接中的表同名的外键，则 SQL Anywhere 使用它来生成连接条件。
- 如果具有多个与表同名的外键，则连接是不明确的，并且将发出错误消息。
- 如果没有与表同名的外键，则 SQL Anywhere 将查找任何外键关系，即使这些名称不匹配。如果有多个外键关系，则连接是不明确的，并且将发出错误消息。

示例 1

在 SQL Anywhere 示例数据库中，在表 Employees 和表 Departments 之间定义了两个外键关系：Employees 表中的外键 FK_DepartmentID_DepartmentID 引用 Departments 表；Departments 表中的外键 FK_DepartmentHeadID_EmployeeID 引用 Employees 表。



以下查询是不明确的，因为该查询具有两个外键关系并且这两个外键关系都不具有与主键表名称相同的角色名。因此，尝试执行此查询将引起语法错误 SQLE_AMBIGUOUS_JOIN (-147)。

```
SELECT Employees.Surname, Departments.DepartmentName
FROM Employees KEY JOIN Departments;
```

示例 2

此查询为 Departments 表指定相关名 FK_DepartmentID_DepartmentID，修改了示例 1 中的查询。现在，外键 FK_DepartmentID_DepartmentID 与其引用的表具有相同的名称，因此用它来定义连接条件。结果中包括所有雇员的姓氏以及他们就职的部门。

```
SELECT Employees.Surname,
    FK_DepartmentID_DepartmentID.DepartmentName
FROM Employees KEY JOIN Departments
    AS FK_DepartmentID_DepartmentID;
```

下面的查询是等效的。在此示例中，不必为 Departments 表创建别名。在此查询的 ON 子句中指定与上述语句生成的相同的连接条件：

```
SELECT Employees.Surname, Departments.DepartmentName
FROM Employees JOIN Departments
   ON Departments.DepartmentID = Employees.DepartmentID;
```

示例 3

如果想要列出所有担任部门领导的雇员，则应使用外键 FK_DepartmentHeadID_EmployeeID，并且应按如下所示重写示例 1。此查询通过为主键表 Employees 指定相关名 FK_DepartmentHeadID_EmployeeID，强制使用外键 FK_DepartmentHeadID_EmployeeID。

```
SELECT FK_DepartmentHeadID_EmployeeID.Surname, Departments.DepartmentName
FROM Employees AS FK_DepartmentHeadID_EmployeeID
    KEY JOIN Departments;
```

下一个查询是等效的。在此查询的 ON 子句中指定与上述语句生成的相同的连接条件：

```
SELECT Employees.Surname, Departments.DepartmentName
FROM Employees JOIN Departments
   ON Departments.DepartmentHeadID = Employees.EmployeeID;
```

示例 4

如果外键角色名与主键表名相同，则不需要相关名。例如，可以为 Employees 表定义外键 Departments：

```
ALTER TABLE Employees
   ADD FOREIGN KEY Departments (DepartmentID)
   REFERENCES Departments (DepartmentID);
```

现在，此外键关系是在这两个表之间指定 KEY JOIN 时的缺省连接条件。如果定义外键 Departments，则下面的查询等效于示例 3。

```
SELECT Employees.Surname, Departments.DepartmentName
FROM Employees KEY JOIN Departments;
```

注意

如果您在 Interactive SQL 中尝试此示例，则应使用下面的语句撤消对 SQL Anywhere 示例数据库的更改：

```
ALTER TABLE Employees DROP FOREIGN KEY Departments;
```

表的表达式的键连接

SQL Anywhere 通过检查语句中每一对表的外键关系，为表表达式的键连接生成连接条件。

以下示例连接四对表。

```
SELECT *
FROM (A NATURAL JOIN B) KEY JOIN (C NATURAL JOIN D);
```

这四个表对是 A-C、A-D、B-C 和 B-D。SQL Anywhere 考虑每一对中的关系，然后将表表达式视为一个整体为其创建生成的连接条件。SQL Anywhere 的具体处理方法取决于表表达式是否使用逗号。因此，以下两个示例中生成的连接条件是不同的。`A JOIN B` 是不包含逗号的表表达式，`(A,B)` 是表表达式列表。

```
SELECT *
FROM (A JOIN B) KEY JOIN C;
```

在语义上不同于

```
SELECT *
FROM (A,B) KEY JOIN C;
```

这两种类型的连接行为将在以下各节中说明：

- “不包含逗号的表的表达式的键连接”一节第 395 页
- “表的表达式列表的键连接”一节第 396 页

不包含逗号的表的表达式的键连接

当要连接的两个表表达式都不包含逗号时，SQL Anywhere 将检查语句中表对中的外键关系，并且生成单个连接条件。

例如，以下连接具有两个表对：A-C 和 B-C。

```
(A NATURAL JOIN B) KEY JOIN C
```

SQL Anywhere 通过查看表对 A-C 和 B-C 中的外键关系为 C 与 (A NATURAL JOIN B) 的连接生成一个连接条件。它根据存在多个外键关系时确定键连接的规则为这两对表生成一个连接条件：

- 首先，它在 A-C 和 B-C 中查找一个外键，该单个外键的角色名与其所引用的某一主键表的相关名相同。如果正好只有一个外键满足此条件，则使用此外键。如果有多个外键的角色名与主键表的相关名相同，则认为该连接是不明确的，并且发出错误消息。
- 如果没有任何外键与主键表的相关名同名，则 SQL Anywhere 将查找这些表间的任何外键关系。如果只有一个外键关系，则使用此外键关系。如果有多个外键关系，则认为该连接是不明确的，并会发出错误消息。
- 如果没有任何外键关系，则发出错误消息。

有关详细信息，请参见“在有多个外键关系时的键连接”一节第 392 页。

示例

以下查询查找职位为销售代表的所有雇员及其所在部门。

```
SELECT Employees.Surname,
        FK_DepartmentID_DepartmentID.DepartmentName
FROM ( Employees KEY JOIN Departments
        AS FK_DepartmentID_DepartmentID )
    KEY JOIN SalesOrders;
```

该查询解释如下。

- SQL Anywhere 考虑表表达式 (Employees KEY JOIN Departments as FK_DepartmentID_DepartmentID) 并基于外键 FK_DepartmentID_DepartmentID 生成连接条件 Employees.DepartmentID = FK_DepartmentID_DepartmentID.DepartmentID。
- 接着 SQL Anywhere 考虑 Employees/SalesOrders 和 Departments/SalesOrders 两对表。请注意，表 SalesOrders 和表 Employees 之间以及表 SalesOrders 和表 Departments 之间只能存在一个外键，否则该连接是不明确的。在表 SalesOrders 和表 Employees 之间碰巧只有一个外键关系 (FK_SalesRepresentative_EmployeeID)，在表 SalesOrders 和表 Departments 之间没有外键关系。因此，生成的连接条件是 SalesOrders.EmployeeID = Employees.SalesRepresentative。

以下查询因此等效于前一查询：

```
SELECT Employees.Surname, Departments.DepartmentName
FROM ( Employees JOIN Departments
    ON ( Employees.DepartmentID = Departments.DepartmentID ) )
JOIN SalesOrders
    ON ( Employees.EmployeeID = SalesOrders.SalesRepresentative );
```

表的表达式列表的键连接

为了给两个表表达式列表的键连接生成连接条件，SQL Anywhere 检查该语句中的表对，并为每对表生成一个连接条件。最终的连接条件是用于每一对的连接条件的联合。每对表之间必须是一个外键关系。

以下示例连接两个表对：A-C 和 B-C。

```
SELECT *
FROM ( A,B ) KEY JOIN C;
```

SQL Anywhere 通过为两个对 A-C 和 B-C 的每一对生成连接条件，生成一个连接 C 与 (A,B) 的连接条件。它根据存在多个外键关系时的键连接规则生成连接条件：

- 对于每一对，SQL Anywhere 查找其角色名与主键表相关名相同的外键。如果正好只有一个外键满足此条件，则使用此外键。如果有多个外键关系，则认为该连接是不明确的，并且发出错误消息。
- 对于每一对，如果没有与表的相关名同名的外键，则 SQL Anywhere 将查找这些表间的任何外键关系。如果只有一个外键关系，则使用此外键关系。如果有多个外键关系，则认为该连接是不明确的，并且发出错误消息。
- 对于每一对，如果没有任何外键关系，则发出错误消息。
- 如果 SQL Anywhere 能够为每一对确定恰好一个连接条件，则它使用 AND 组合这些连接条件。

另请参见“在有多个外键关系时的键连接”一节第 392 页。

示例

以下查询返回向特定区域售出至少一个订单的所有销售人员的姓名。

```
SELECT DISTINCT Employees.Surname,
        FK_DepartmentID_DepartmentID.DepartmentName,
        SalesOrders.Region
FROM ( SalesOrders, Departments
        AS FK_DepartmentID_DepartmentID )
    KEY JOIN Employees;
```

Surname	DepartmentName	Region
Chin	Sales	Eastern
Chin	Sales	Western
Chin	Sales	Central
...	...	...

此查询处理两对表：SalesOrders 和 Employees 以及 Departments AS FK_DepartmentID_DepartmentID 和 Employees。

对于 SalesOrders 和 Employees 这一对，没有任何外键具有与其中一个表同名的角色名。但是，存在一个与这两个表关联的外键 (FK_SalesRepresentative_EmployeeID)。该外键是关联这两个表的唯

一外键，所以使用它，结果得到生成的连接条件 (Employees.EmployeeID = SalesOrders.SalesRepresentative)。

对于 Departments AS FK_DepartmentID_DepartmentID 和 Employees 这一对，存在一个外键，其角色名与主键表的相关名同名。它是 FK_DepartmentID_DepartmentID，并且它与在该查询中为 Departments 表指派的相关名匹配。由于再没有其它外键的角色名与主键表的相关名同名，因此使用 FK_DepartmentID_DepartmentID 为该表对形成连接条件。生成的连接条件是 (Employees.DepartmentID = FK_DepartmentID_DepartmentID.DepartmentID)。请注意，还有一个外键与这两个表关联，但因为它的名称不同于这两个表中的任何一个表的名称，所以不考虑它。

最终的连接条件是为每个表对生成的连接条件的联合。因此，以下查询是等效的：

```
SELECT DISTINCT Employees.Surname,
   Departments.DepartmentName,
   SalesOrders.Region
FROM ( SalesOrders, Departments )
    JOIN Employees
    ON Employees.EmployeeID = SalesOrders.SalesRepresentative
    AND Employees.DepartmentID = Departments.DepartmentID;
```

表的表达式列表和不包含逗号的表的表达式的键连接

当表表达式列表通过键连接与不包含逗号的表表达式连接时，SQL Anywhere 为表表达式列表中的每个表生成一个连接条件。

例如，以下语句是一个表表达式列表与一个不包含逗号的表表达式的键连接。本示例为表 A 和表表达式 C NATURAL JOIN D 以及表 B 和表表达式 C NATURAL JOIN D 生成一个连接条件。

```
SELECT *
FROM (A,B) KEY JOIN (C NATURAL JOIN D);
```

(A,B) 是表表达式列表，C NATURAL JOIN D 是表表达式。因此，SQL Anywhere 必须生成两个连接条件：它为 A-C 和 A-D 对生成一个连接条件，为 B-C 和 B-D 对生成第二个连接条件。它根据存在多个外键关系时的键连接规则生成连接条件：

- 对于每组表对，SQL Anywhere 查找其角色名与其中一个主键表的相关名同名的外键。如果正好只有一个外键满足此条件，则使用此外键。如果有多个，则认为该连接是不明确的，并且发出错误消息。
- 对于每组表对，如果不存在其角色名与主键表的相关名同名的外键，则 SQL Anywhere 查找这些表间的任何外键关系。如果恰好有一个外键关系，则连接条件使用此关系。如果有多个，则认为该连接是不明确的，并且发出错误消息。
- 对于每组对，如果没有任何外键关系，则发出错误消息。
- 如果 SQL Anywhere 能够为每一组表对确定恰好一个连接条件，则它将使用关键字 AND 来组合这些连接条件。

示例 1

考虑以下 5 个表的连接：

```
((A,B) JOIN (C NATURAL JOIN D) ON A.x = D.y) KEY JOIN E
```

在此示例中，SQL Anywhere 通过在 (A,B) 和 E 之间或在 C NATURAL JOIN D 和 E 之间生成条件的方式来为与 E 的键连接生成连接条件。“不包含逗号的表的表达式的键连接”一节第 395 页中说明了这一情况。

如果 SQL Anywhere 在 (A,B) 和 E 之间生成连接条件，则它需要创建两个连接条件，为 A-E 创建一个，为 B-E 创建另一个。它必须查找每个表对中有效的外键关系。“表的表达式列表的键连接”一节第 396 页中说明了这一情况。

如果 SQL Anywhere 在 C NATURAL JOIN D 和 E 之间创建连接条件，则它将只创建一个连接条件，因此必须只在 C-E 和 D-E 对中找到一个外键关系。“不包含逗号的表的表达式的键连接”一节第 395 页中说明了这一情况。

示例 2

以下是表表达式和表表达式列表的键连接的示例。该示例提供既是销售代表也是经理的雇员的姓名和部门。

```
SELECT DISTINCT Employees.Surname,
        FK_DepartmentID_DepartmentID.DepartmentName
FROM ( SalesOrders, Departments
        AS FK_DepartmentID_DepartmentID )
    KEY JOIN ( Employees JOIN Departments AS d
        ON Employees.EmployeeID = d.DepartmentHeadID );
```

SQL Anywhere 生成两个连接条件：

- 表对 SalesOrders/Employees 和 SalesOrders/d 之间正好有一个外键关系：SalesOrders.SalesRepresentative = Employees.EmployeeID.
- 表对 FK_DepartmentID_DepartmentID/Employees 和 FK_DepartmentID_DepartmentID/d 之间正好有一个外键关系：FK_DepartmentID_DepartmentID.DepartmentID = Employees.DepartmentID。

此示例等效于下面的语句。在以下版本中，不必创建相关名 Departments AS FK_DepartmentID_DepartmentID，因为只有在说明应使用两个外键中的哪一个来连接 Employees 和 Departments 时才需要相关名。

```
SELECT DISTINCT Employees.Surname,
   Departments.DepartmentName
FROM ( SalesOrders, Departments )
    JOIN ( Employees JOIN Departments AS d
       ON Employees.EmployeeID = d.DepartmentHeadID )
    ON SalesOrders.SalesRepresentative = Employees.EmployeeID
       AND Departments.DepartmentID = Employees.DepartmentID;
```

视图和派生表的键连接

当键连接中包括视图或派生表时，SQL Anywhere 遵循与处理表相同的基本过程，但有以下差异：

- 对于每一个键连接，SQL Anywhere 都会考虑在查询和视图的 FROM 子句中的表对，并为所有表对的集合生成一个连接条件，而不管视图的 FROM 子句中是包含逗号还是连接关键字。
- SQL Anywhere 基于其角色名与视图或派生表的相关名同名的外键连接表。

- 在键连接中包括视图或派生表时，视图或派生表定义中不能包含 UNION、INTERSECT、EXCEPT、ORDER BY、DISTINCT、GROUP BY、集合函数、窗口函数、TOP、FIRST、START AT 或 FOR XML。如果它包含以上任何一项，则返回错误。此外，不能将派生表定义为递归表表达式。

  派生表的工作原理与视图大体相同。唯一差别就是：表的定义包括在语句中，而不是引用预定义的视图。

  有关递归表表达式的信息，请参见“递归公用表表达式”一节第 411 页和“RecursiveTable 算法 (RT)”一节第 563 页。

示例 1

例如，在以下语句中，View1 是一个视图。

```
SELECT *
FROM View1 KEY JOIN B;
```

View1 的定义可以是以下任何一个，结果得到相同的连接条件与 B 连接。（结果集不同，但连接条件相同。）

```
SELECT *
FROM C CROSS JOIN D;
```

或者

```
SELECT *
FROM C,D;
```

或者

```
SELECT *
FROM C JOIN D ON (C.x = D.y);
```

在每种情况下，在为 View1 和 B 的键连接生成连接条件时，SQL Anywhere 会考虑表对 C-B 和 D-B，并生成一个连接条件。它基于“表的表达式的键连接”一节第 394 页中描述的适用于多个外键关系的规则生成连接条件，只不过它会查找与视图（而不是在视图中引用的表）的相关名同名的外键。

使用以上任一视图定义时，您都可以按如下所述解释 `View1 KEY JOIN B` 的处理过程：

SQL Anywhere 通过考虑表对 C-B 和 D-B 来生成单个连接条件。它根据以下有多个外键关系时确定键连接的规则来生成该连接条件：

- 首先，它查找 C-B 和 D-B 是否存在其角色名与视图的相关名同名的一个外键。如果正好只有一个外键满足此条件，则使用此外键。如果有多个角色名与视图的相关名同名的外键，则认为该连接是不明确的，并且发出错误消息。
- 如果没有任何外键与视图的相关名同名，则 SQL Anywhere 将查找这些表间的任何外键关系。如果只有一个外键关系，则使用此外键关系。如果有多个外键关系，则认为该连接是不明确的，并会发出错误消息。
- 如果没有任何外键关系，则发出错误消息。

假设此生成的连接条件是 `B.y = D.z`。现在可以展开原始连接。例如，以下两个语句是等效的：

```
SELECT *
FROM View1 KEY JOIN B;

SELECT *
FROM View1 JOIN B ON B.y = View1.z;
```

请参见“在有多个外键关系时的键连接”一节第 392 页。

示例 2

以下视图包含与每一部门的经理有关的所有雇员信息。

```
CREATE VIEW V AS
SELECT Departments.DepartmentName, Employees.*
FROM Employees JOIN Departments
  ON Employees.EmployeeID = Departments.DepartmentHeadID;
```

以下查询将该视图连接到一个表表达式。

```
SELECT *
FROM V KEY JOIN ( SalesOrders,
   Departments FK_DepartmentID_DepartmentID );
```

以下查询等同于上一查询：

```
SELECT *
FROM V JOIN ( SalesOrders,
   Departments FK_DepartmentID_DepartmentID )
ON ( V.EmployeeID = SalesOrders.SalesRepresentative
AND V.DepartmentID =
    FK_DepartmentID_DepartmentID.DepartmentID );
```

描述键连接的操作的规则

以下规则总结前面提供的信息。

规则 1：两个表的键连接

此规则适用于 `A KEY JOIN B`，其中 A 和 B 是基表或临时表。

1. 从 A 中查找所有引用 B 的外键。

   如果存在其角色名是表 B 的相关名的外键，则将它标记为首选外键。
2. 从 B 中查找所有引用 A 的外键。

   如果存在其角色名是表 A 的相关名的外键，则将它标记为首选外键。
3. 如果有多个首选键，则连接是不明确的。发出语法错误 `SQLE_AMBIGUOUS_JOIN (-147)`。
4. 如果有单个首选键，则选择此外键为此 KEY JOIN 表达式定义生成的连接条件。
5. 如果没有首选键，则使用 A 和 B 间的其它外键：
   - 如果在 A 和 B 间有多个外键，则连接是不明确的。发出语法错误 `SQLE_AMBIGUOUS_JOIN (-147)`。

- 如果有单个外键，则选择此外键为此 KEY JOIN 表达式定义生成的连接条件。
- 如果没有任何外键，则该连接是无效的，将生成错误。

规则 2：不包含逗号的表表达式的键连接

此规则适用于 `A KEY JOIN B`，其中 A 和 B 是不包含逗号的表表达式。

1. 针对来自表达式 A 和表达式 B 的每对表，列出所有外键，并且标记这些表间的所有首选外键。用于确定首选外键的规则在前面的规则 1 中提供。
2. 如果有多个首选键，则连接是不明确的。发出语法错误 `SQLE_AMBIGUOUS_JOIN (-147)`。
3. 如果有单个首选键，则选择此外键为此 KEY JOIN 表达式定义生成的连接条件。
4. 如果没有首选键，则使用表对间的其它外键：
   - 如果有多个外键，则该连接是不明确的。发出语法错误 `SQLE_AMBIGUOUS_JOIN (-147)`。
   - 如果有单个外键，则选择此外键为此 KEY JOIN 表达式定义生成的连接条件。
   - 如果没有任何外键，则该连接是无效的，将生成错误。

规则 3：表的表达式列表的键连接

此规则适用于 `(A1, A2, ...) KEY JOIN ( B1, B2, ...)`，其中 A1、B1 等是不包含逗号的表表达式。

1. 对于每一对表表达式 Ai 和 Bj，通过应用规则 1 或 2 为表表达式 `(Ai KEY JOIN Bj)` 找到唯一生成的连接条件。如果按规则 1 或规则 2 确定任何一对表表达式的 KEY JOIN 是不明确的，则生成语法错误。
2. 用于此 KEY JOIN 表达式的生成的连接条件是在第 1 步中找到的连接条件的联合。

规则 4：表的表达式列表和不包含逗号的表的表达式的键连接

此规则适用于 `(A1, A2, ...) KEY JOIN ( B1, B2, ...)`，其中 A1、B1 等是包含逗号的表表达式。

1. 对于每一对表表达式 Ai 和 Bj，通过应用规则 1、2 或 3 为表表达式 `(Ai KEY JOIN Bj)` 找到唯一生成的连接条件。如果按规则 1、2 或 3 确定任何一对表表达式的 KEY JOIN 是不明确的，则生成语法错误。
2. 用于此 KEY JOIN 表达式的生成的连接条件是在第 1 步中找到的连接条件的联合。

公用表表达式

目录

通过使用 SELECT 语句的 WITH 前缀，您可以定义公用表表达式。公用表表达式是临时视图，它们只在某一条 SELECT 语句的范围内是已知的。使用它们，您编写查询会更容易，还可以编写无法以其它方式表达的查询。

如果查询涉及到多个集合函数，或者用于在存储过程中定义一个引用程序变量的视图，那么，公用表表达式就会很有用，或者可能是必需的。公用表表达式还提供了一种便捷的方式，用以临时存储值集。

使用公用表表达式

公用表表达式是使用 WITH 子句定义的，该子句在 SELECT 语句中的 SELECT 关键字之前。子句的内容定义了之后可能会在语句中的其它位置引用的一个或多个临时视图。此子句的语法模拟 CREATE VIEW 语句的语法。您可以使用公用表表达式来表达上一个查询，如下所示。

```
WITH CountEmployees( DepartmentID, n ) AS
  ( SELECT DepartmentID, COUNT( * ) AS n
    FROM Employees GROUP BY DepartmentID )
SELECT DepartmentID, n
FROM CountEmployees
WHERE n = ( SELECT MAX( n )
            FROM CountEmployees );
```

将查询改为搜索雇员最少的部门说明这种查询可以返回多个行。

```
WITH CountEmployees( DepartmentID, n ) AS
  ( SELECT DepartmentID, COUNT( * ) AS n
    FROM Employees GROUP BY DepartmentID )
SELECT DepartmentID, n
FROM CountEmployees
WHERE n = ( SELECT MIN( n )
            FROM CountEmployees );
```

在 SQL Anywhere 示例数据库中，有两个部门的雇员数最少，均为 9。

另请参见

- “指定多个相关名”一节第 405 页
- “使用多个表表达式”一节第 406 页
- “允许使用公用表表达式的位置”一节第 407 页

指定多个相关名

就像在使用表时一样，您可以为公用表表达式的多个实例指定不同的相关名。这样做使您可以将公用表表达式与它本身连接。例如，下面的查询生成了具有相同数量雇员的部门对，虽然 SQL Anywhere 示例数据库中只有两个具有相同数量雇员的部门。

```
WITH CountEmployees( DepartmentID, n ) AS
    ( SELECT DepartmentID, COUNT( * ) AS n
      FROM Employees GROUP BY DepartmentID )
SELECT a.DepartmentID, a.n, b.DepartmentID, b.n
FROM CountEmployees AS a JOIN CountEmployees AS b
ON a.n = b.n AND a.DepartmentID < b.DepartmentID;
```

另请参见

- “使用公用表表达式”一节第 404 页
- “使用多个表表达式”一节第 406 页
- “允许使用公用表表达式的位置”一节第 407 页

使用多个表表达式

一个 WITH 子句可以定义多个公用表表达式。这些定义必须用逗号分隔开。以下示例列出了工资额最少的部门和雇员数量最多的部门。

```
WITH
  CountEmployees( DepartmentID, n ) AS
    ( SELECT DepartmentID, COUNT( * ) AS n
      FROM Employees GROUP BY DepartmentID ),
  DeptPayroll( DepartmentID, amt ) AS
     ( SELECT DepartmentID, SUM( Salary ) AS amt
       FROM Employees GROUP BY DepartmentID )
SELECT count.DepartmentID, count.n, pay.amt
FROM CountEmployees AS count JOIN DeptPayroll AS pay
ON count.DepartmentID = pay.DepartmentID
WHERE count.n = ( SELECT MAX( n ) FROM CountEmployees )
   OR pay.amt = ( SELECT MIN( amt ) FROM DeptPayroll );
```

另请参见

- “使用公用表表达式”一节第 404 页
- “指定多个相关名”一节第 405 页
- “允许使用公用表表达式的位置”一节第 407 页

允许使用公用表表达式的位置

虽然可以在整个查询主体或任何子查询中引用公用表表达式定义，但只允许在三个位置使用它们。

- **顶级 SELECT 语句** 允许在顶级 SELECT 语句中使用公用表表达式，但不允许在子查询中使用。

```
WITH DeptPayroll( DepartmentID, amt ) AS
  ( SELECT DepartmentID, SUM( Salary ) AS amt
    FROM Employees GROUP BY DepartmentID )
SELECT DepartmentID, amt
FROM DeptPayroll
WHERE amt = ( SELECT MAX( amt )
              FROM DeptPayroll );
```

- **视图定义中的顶级 SELECT 语句** 允许在定义视图的顶级 SELECT 语句中使用公用表表达式，但不允许在定义中的子查询中使用。

```
CREATE VIEW LargestDept ( DepartmentID, Size, pay ) AS
   WITH
     CountEmployees( DepartmentID, n ) AS
       ( SELECT DepartmentID, COUNT( * ) AS n
         FROM Employees GROUP BY DepartmentID ),
     DeptPayroll( DepartmentID, amt ) AS
       ( SELECT DepartmentID, SUM( Salary ) AS amt
         FROM Employees GROUP BY DepartmentID )
  SELECT count.DepartmentID, count.n, pay.amt
  FROM CountEmployees count JOIN DeptPayroll pay
  ON count.DepartmentID = pay.DepartmentID
  WHERE count.n = ( SELECT MAX( n ) FROM CountEmployees )
     OR pay.amt = ( SELECT MAX( amt ) FROM DeptPayroll );
```

- **INSERT 语句中的顶级 SELECT 语句** 允许在 INSERT 语句中的顶级 SELECT 语句中使用公用表表达式，但不允许在 INSERT 语句中的子查询中使用。

```
CREATE TABLE LargestPayrolls ( DepartmentID INTEGER, Payroll NUMERIC,
CurrentDate DATE );
INSERT INTO LargestPayrolls( DepartmentID, Payroll, CurrentDate )
  WITH DeptPayroll( DepartmentID, amt ) AS
    ( SELECT DepartmentID, SUM( Salary ) AS amt
      FROM Employees
      GROUP BY DepartmentID )
  SELECT DepartmentID, amt, CURRENT TIMESTAMP
  FROM DeptPayroll
  WHERE amt = ( SELECT MAX( amt )
                FROM DeptPayroll );
```

另请参见

- “使用公用表表达式”一节第 404 页
- “指定多个相关名”一节第 405 页
- “使用多个表表达式”一节第 406 页

公用表表达式的典型应用

一般而言，只要表的表达式必须在一个查询中出现多次，公用表表达式就很有用。以下典型情况适合于公用表表达式。

- 涉及多个集合函数的查询。
- 过程中必须包含对程序变量的引用的视图。
- 使用临时视图存储一组值的查询。

此列表并未包括所有情况；您可能会遇到许多其它要使用公用表表达式的情况。

多个集合函数

只要必须在一个查询中显示多个级别的集合，公用表表达式就很有用。这是上一部分使用的示例中的情况。任务是检索雇员数量最多的部门的部门 ID。为此，要使用 count 集合函数来计算每个部门的雇员数量，并使用 MAX 函数选择最大的部门。

在编写查询来确定哪个部门的工资额最多时，会出现类似的情况。SUM 集合函数用于计算每个部门的工资额，而 MAX 函数用于确定哪个部门最大。查询中同时出现这两个函数表明公用表表达式可能有用。

```
WITH DeptPayroll( DepartmentID, amt ) AS
     ( SELECT DepartmentID, SUM( Salary ) AS amt
       FROM Employees GROUP BY DepartmentID )
SELECT DepartmentID, amt
FROM DeptPayroll
WHERE amt = ( SELECT MAX( amt )
              FROM DeptPayroll )
```

有关集合函数的详细信息，请参见“窗口集合函数”一节第 442 页。

引用程序变量的视图

有时，对于创建包含到程序变量的引用的视图，使用公用表表达式可能会很方便。例如，您可以定义过程中标识特定客户的变量。您要查询该客户的购买历史记录，并且如果您要多次访问类似信息或者可能会使用多个集合函数，则您需要创建一个包含有关该特定客户信息的视图。

您无法创建引用程序变量的视图，因为无法将视图范围限制为您的过程的范围。一旦创建了视图，就可以在其它环境中使用该视图。但是，您可以在您的过程中的查询内使用公用表表达式。因为公用表表达式的范围限制到语句，所以变量引用不会造成任何多义性，因此可以使用变量引用。

以下语句用于在 SQL Anywhere 示例数据库中选择各位不同销售代表的销售总额。

```
SELECT GivenName || ' ' || Surname AS sales_rep_name,
       SalesRepresentative AS sales_rep_id,
       SUM( p.UnitPrice * i.Quantity ) AS total_sales
FROM Employees LEFT OUTER JOIN SalesOrders AS o
               INNER JOIN SalesOrderItems AS I
               INNER JOIN Products AS p
```



```
WHERE OrderDate BETWEEN '2000-01-01' AND '2001-12-31'
GROUP BY SalesRepresentative, GivenName, Surname;
```

上面的查询是出现在以下过程中的公用表表达式的基础。销售代表的 ID 号和所讨论的年份是即将使用的参数。如同以下过程所演示的那样，可在 WITH 子句中引用过程参数和任何已声明的局部变量。

```
CREATE PROCEDURE sales_rep_total (
  IN rep  INTEGER,
  IN yyyy INTEGER )
BEGIN
  DECLARE StartDate DATE;
  DECLARE EndDate   DATE;
  SET StartDate = YMD( yyyy,  1,  1 );
  SET EndDate = YMD( yyyy, 12, 31 );
  WITH total_sales_by_rep ( sales_rep_name,
                            sales_rep_id,
                            month,
                            order_year,
                            total_sales ) AS
  ( SELECT GivenName || ' ' || Surname AS sales_rep_name,
           SalesRepresentative AS sales_rep_id,
           month( OrderDate),
           year( OrderDate ),
           SUM( p.UnitPrice * i.Quantity ) AS total_sales
    FROM Employees LEFT OUTER JOIN SalesOrders o
                       INNER JOIN SalesOrderItems I
                       INNER JOIN Products p
    WHERE OrderDate BETWEEN StartDate AND EndDate
           AND SalesRepresentative = rep
    GROUP BY year( OrderDate ), month( OrderDate ),
             GivenName, Surname, SalesRepresentative )
  SELECT sales_rep_name,
         monthname( YMD(yyyy, month, 1) ) AS month_name,
         order_year,
         total_sales
  FROM total_sales_by_rep
  WHERE total_sales =
    ( SELECT MAX( total_sales) FROM total_sales_by_rep )
  ORDER BY order_year ASC, month ASC;
END;
```

以下语句调用上述过程。

```
CALL sales_rep_total( 129, 2000 );
```

存储值的视图

这对于在某个 SELECT 语句或某个过程中存储一组特定的值可能会非常有用。例如，假定一家公司要按三分之一年度而不是按季度分析它的销售人员的结果。由于没有代表三分之一的内置日期部分（虽然有代表季度的内置日期部分），所以有必要将这些日期存储在过程中。

```
WITH thirds ( q_name, q_start, q_end ) AS
( SELECT 'T1', '2000-01-01', '2000-04-30' UNION
  SELECT 'T2', '2000-05-01', '2000-08-31' UNION
  SELECT 'T3', '2000-09-01', '2000-12-31' )
SELECT q_name,
       SalesRepresentative,
       count(*) AS num_orders,
```

```
       SUM( p.UnitPrice * i.Quantity ) AS total_sales
FROM thirds LEFT OUTER JOIN SalesOrders AS o
    ON OrderDate BETWEEN q_start and q_end
                  KEY JOIN SalesOrderItems AS I
                  KEY JOIN Products AS p
 GROUP BY q_name, SalesRepresentative
 ORDER BY q_name, SalesRepresentative;
```

使用此方法时应该小心，因为值可能需要定期维护。例如，如果要针对任何其它年度使用上面的语句，则必须对它进行修改。

您还可以在过程中应用此方法。以下示例声明了一个过程，该过程将所讨论的年份作为参数。

```
CREATE PROCEDURE sales_by_third ( IN y INTEGER )
BEGIN
  WITH thirds ( q_name, q_start, q_end ) AS
  ( SELECT 'T1', YMD( y, 01, 01), YMD( y, 04, 30 ) UNION
    SELECT 'T2', YMD( y, 05, 01), YMD( y, 08, 31 ) UNION
    SELECT 'T3', YMD( y, 09, 01), YMD( y, 12, 31 ) )
  SELECT q_name,
         SalesRepresentative,
         count(*) AS num_orders,
         SUM( p.UnitPrice * i.Quantity ) AS total_sales
  FROM thirds LEFT OUTER JOIN SalesOrders AS o
    ON OrderDate BETWEEN q_start and q_end
           KEY JOIN SalesOrderItems AS I
           KEY JOIN Products AS p
  GROUP BY q_name, SalesRepresentative
  ORDER BY q_name, SalesRepresentative;
END;
```

以下语句调用上述过程。

```
CALL sales_by_third (2000);
```


递归公用表表达式

公用表表达式可以是递归的。当 RECURSIVE 关键字紧跟在 WITH 后面出现时，公用表表达式是递归的。一个 WITH 子句可以包含多个递归表达式，并且可以同时包含递归公用表表达式和非递归公用表表达式。

递归提供了更加容易的方法来遍历呈现树状或类似树状的数据结构的表。在不使用递归表达式的情况下使用一个语句遍历这种结构的唯一方法是针对每个可能的级别让表与它自身连接一次。例如，如果报告层次最多包含七个级别，您必须将 Employees 表与它自身连接七次。如果公司进行重组并且引入了一个新的管理级别，您必须重写查询。

递归公用表表达式提供了一种方便的方法来编写将关系返回到任意深度的查询。例如，假定有一个呈现公司内的报告关系的表，您可以很容易地编写一个查询，让它返回向某特定人员报告的所有雇员。

如，以确定哪个部门的雇员数量最多的问题为例。SQL Anywhere 示例数据库中的 Employees 表列出了一家虚构公司内的所有雇员，并指定了每名雇员的工作部门。以下查询列出部门 ID 代码以及各部门中的雇员总数。

```
SELECT DepartmentID, COUNT( * ) AS n
FROM Employees
GROUP BY DepartmentID;
```

此查询可用于提取雇员最多的部门，如下所示：

```
SELECT DepartmentID, n
FROM ( SELECT DepartmentID, COUNT( * ) AS n
       FROM Employees GROUP BY DepartmentID ) AS a
WHERE a.n =
  ( SELECT MAX( n )
    FROM ( SELECT DepartmentID, COUNT( * ) AS n
           FROM Employees GROUP BY DepartmentID ) AS b );
```

虽然此语句提供了正确的结果，但它有几个缺点。第一个缺点是重复的子查询会降低此语句的效率。第二个缺点是此语句没有在子查询之间提供清楚的链接。

解决这些问题的一个方法是创建一个视图，然后使用该视图重新表达查询。这种方法可避免上述问题。

```
CREATE VIEW CountEmployees( DepartmentID, n ) AS
   SELECT DepartmentID, COUNT( * ) AS n
   FROM Employees GROUP BY DepartmentID;

SELECT DepartmentID, n
   FROM CountEmployees
   WHERE n = ( SELECT MAX( n )
               FROM CountEmployees );
```

这种方法的缺点是需要一些开销，因为数据库服务器必须在创建视图时更新系统表。如果经常使用该视图，则使用这种方法是合理的。但是，如果只在特定 SELECT 语句中使用一次该视图，则首选方法是改为使用公用表表达式。有关公用表表达式的详细信息，请参见“使用公用表表达式”一节第 404 页。

递归公用表表达式包含一个**初始子查询**（也就是种子）以及一个在每次迭代期间给结果集追加其它行的**递归子查询**。连接这两个部分只能使用运算符 UNION ALL。初始子查询是普通的非递归查

询，首先得到处理。递归部分包含对上一次迭代期间添加的行的引用。只要迭代不生成新行，递归就会自动停止。对于在上一次迭代之前选择的行，无法引用。

递归子查询的选择列表必须在编号和数据类型方面都与初始子查询的选择列表匹配。如果无法执行数据类型的自动转换，可显式转换一个子查询的结果，以便它们与其它子查询的结果匹配。

选择分层数据

根据您编写查询的方式，您最好限制递归级别数。如果限制级别数，您可以只返回顶级管理人员（举例来说），但是，如果命令链比您的预期要长，则可能会排除一些雇员。如果不对级别数加以限制，则可确保不会排除任何雇员，但是，如果执行需要任何循环，则可能会引入无限递归。例如，如果某雇员直接或间接地向他自己报告。公司的管理层次中也可能会发生这种情况，例如，公司的某名雇员同时也是董事会成员。

以下查询说明了如何按管理级别列出雇员。级别 0 表示没有经理的雇员。级别 1 表示直接向某一位级别 0 的经理报告的雇员，级别 2 表示直接向级别 1 的经理报告的雇员，依此类推。

```
WITH RECURSIVE
  manager ( EmployeeID, ManagerID,
            GivenName, Surname, mgmt_level ) AS
( ( SELECT EmployeeID, ManagerID,       -- initial subquery
           GivenName, Surname, 0
    FROM Employees AS e
    WHERE ManagerID = EmployeeID )
  UNION ALL
  ( SELECT e.EmployeeID, e.ManagerID,   -- recursive subquery
           e.GivenName, e.Surname, m.mgmt_level + 1
    FROM Employees AS e JOIN manager AS m
     ON   e.ManagerID =  m.EmployeeID
      AND e.ManagerID <> e.EmployeeID
      AND m.mgmt_level < 20 ) )
SELECT * FROM manager
ORDER BY mgmt_level, Surname, GivenName;
```

递归查询中将管理级别限制为低于 20 的条件是一项重要的预防措施。它可以防止在表数据包含循环时出现无限递归。

max_recursive_iterations 选项

max_recursive_iterations 选项旨在捕获失控的递归查询。此选项的缺省值是 100。超过此递归级别数的递归查询会结束，但是会导致出现错误。

虽然此选项似乎可以降低停止条件的重要性，但是通常并不是这样。在每次迭代期间选择的行数可能会以指数方式增加，在达到最大值之前就会严重影响数据库性能。递归查询中的停止条件提供了为每种情况设置适当限制的方法。

递归公用表表达式的限制

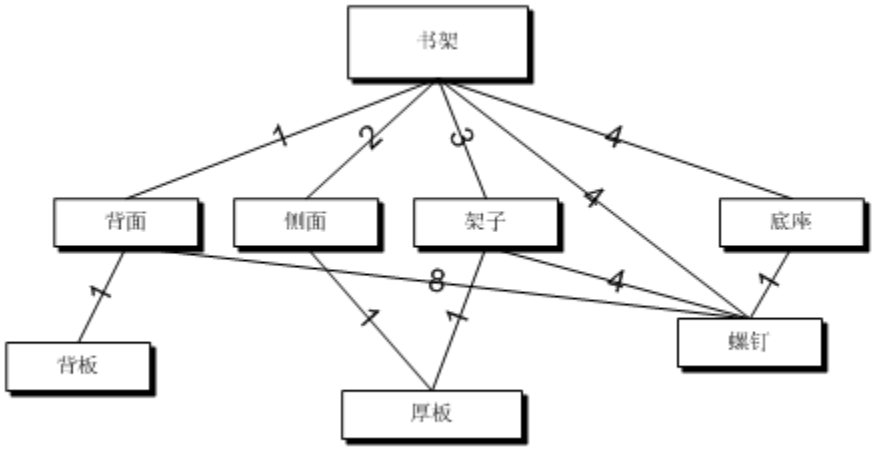
以下限制适用于递归公用表表达式。

- 递归公用表表达式的定义中不能出现对其它递归公用表表达式的引用。因此，递归公用表表达式不能相互递归。但是，非递归公用表表达式可以包含对递归公用表表达式的引用，递归公用表表达式可以包含对非递归公用表表达式的引用。
- 允许在初始子查询和递归子查询之间使用的唯一集合运算符是 UNION ALL。禁止使用其它集合运算符。
- 在递归子查询的定义中，对递归表表达式的自我引用只能出现在递归子查询的 FROM 子句中。
- 当自我引用出现在递归子查询的 FROM 子句中时，对递归表的引用不能出现在外连接的空值提供方。
- 递归子查询不能包含 DISTINCT、GROUP BY 或 ORDER BY 子句。
- 递归子查询不能使用任何集合函数。
- 为了防止递归查询失控，如果递归级别数超过 max_recursive_iterations 选项的当前设置，则会生成错误。此选项的缺省值是 100。

部件激增问题

部件激增问题是递归的典型应用。在此问题中，组合特定对象所必需的组件由图形表示。目标是使用数据库表表示此图形，然后计算必需元素部件的总数。

例如，下图显示了一个简单书架的各个组件。书架由三层架板、一块背板和四条腿组成，四条腿用四个螺钉固定。每块架板用四个螺钉固定。背板用八个螺钉固定。



下表中的信息表示书架图形的边。第一列命名组件，第二列命名该组件的其中一个子组件，第三列指定需要多少个子组件。

组件	子组件	数量
书架	背面	1
书架	侧面	2
书架	架子	3
书架	底部	4
书架	螺钉	4
背面	背板	1
背面	螺钉	8
侧面	厚板	1
架子	厚板	1

组件	子组件	数量
架子	螺钉	4

执行以下语句创建书架表并插入组件和子组件数据。

```
CREATE TABLE bookcase (
      component       VARCHAR(9),
      subcomponent    VARCHAR(9),
      quantity        INTEGER,
    PRIMARY KEY ( component, subcomponent )
);
INSERT INTO bookcase
  SELECT 'bookcase', 'back',      1 UNION
  SELECT 'bookcase', 'side',      2 UNION
  SELECT 'bookcase', 'shelf',     3 UNION
  SELECT 'bookcase', 'foot',      4 UNION
  SELECT 'bookcase', 'screw',     4 UNION
  SELECT 'back',     'backboard', 1 UNION
  SELECT 'back',     'screw',     8 UNION
  SELECT 'side',     'plank',     1 UNION
  SELECT 'shelf',    'plank',     1 UNION
  SELECT 'shelf',    'screw',     4;
```

执行以下语句生成由组件、子组件以及装配书架所需的数量组成的列表。

```
SELECT * FROM bookcase
ORDER BY component, subcomponent;
```

执行以下语句生成由子组件以及装配书架所需的数量组成的列表。

```
WITH RECURSIVE parts ( component, subcomponent, quantity ) AS
( SELECT component, subcomponent, quantity
  FROM bookcase WHERE component = 'bookcase'
    UNION ALL
  SELECT b.component, b.subcomponent, p.quantity * b.quantity
  FROM parts p JOIN bookcase b ON p.subcomponent = b.component )
SELECT subcomponent, SUM( quantity ) AS quantity
FROM parts
WHERE subcomponent NOT IN ( SELECT component FROM bookcase )
GROUP BY subcomponent
ORDER BY subcomponent;
```

下面显示了此查询的结果。

subcomponent	quantity
backboard	1
foot	4
plank	5
screw	24

或者，您也可以重写此查询以执行其它级别的递归，从而消除在主 SELECT 语句中使用子查询的需要。以下查询的结果与上一个查询的那些结果相同。

```
WITH RECURSIVE parts ( component, subcomponent, quantity ) AS
( SELECT component, subcomponent, quantity
  FROM bookcase WHERE component = 'bookcase'
    UNION ALL
  SELECT p.subcomponent, b.subcomponent,
    IF b.quantity IS NULL
    THEN p.quantity
    ELSE p.quantity * b.quantity
    ENDIF
  FROM parts p LEFT OUTER JOIN bookcase b
  ON p.subcomponent = b.component
    WHERE p.subcomponent IS NOT NULL
 )
SELECT component, SUM( quantity ) AS quantity
FROM parts
WHERE subcomponent IS NULL
GROUP BY component
ORDER BY component;
```


递归公用表表达式中的数据类型声明

临时视图中各列的数据类型由初始子查询中的数据类型定义。递归子查询中各列的数据类型必须匹配。数据库服务器会自动尝试转换由递归子查询返回的值，以便与初始查询的那些值匹配。如果这无法实现，或者如果转换中可能会丢失信息，则会产生错误。

一般而言，当初始子查询返回实际值或 NULL 值时，通常需要进行显式转换。当初始子查询从与递归子查询不同的列中选择值时，也可能需要进行显式转换。

如果初始子查询的列与递归子查询的列具有不同的域，则可能需要进行转换。对于初始子查询中的 NULL 值，必须始终进行转换。

例如，书架部件分解示例会正确运行，因为初始子查询返回书架表中的行，并继承了所选列的数据类型。请参见“部件激增问题”一节第 414 页。

如果按照如下所示重写此查询，则需要进行显式转换。

```
WITH RECURSIVE parts ( component, subcomponent, quantity ) AS
( SELECT NULL, 'bookcase', 1          -- ERROR! Wrong domains!
    UNION ALL
  SELECT b.component, b.subcomponent,
         p.quantity * b.quantity
  FROM parts p JOIN bookcase b
    ON p.subcomponent = b.component )
SELECT * FROM parts
ORDER BY component, subcomponent;
```

如果没有进行转换，就会由于以下原因而导致错误：

- 组件名的正确数据类型是 VARCHAR，但是第一列为 NULL。
- 假定数字 1 是 SMALL INT，但是数量列的数据类型是 INT。

不需要对第二列进行转换，因为初始查询的该列已经是字符串。

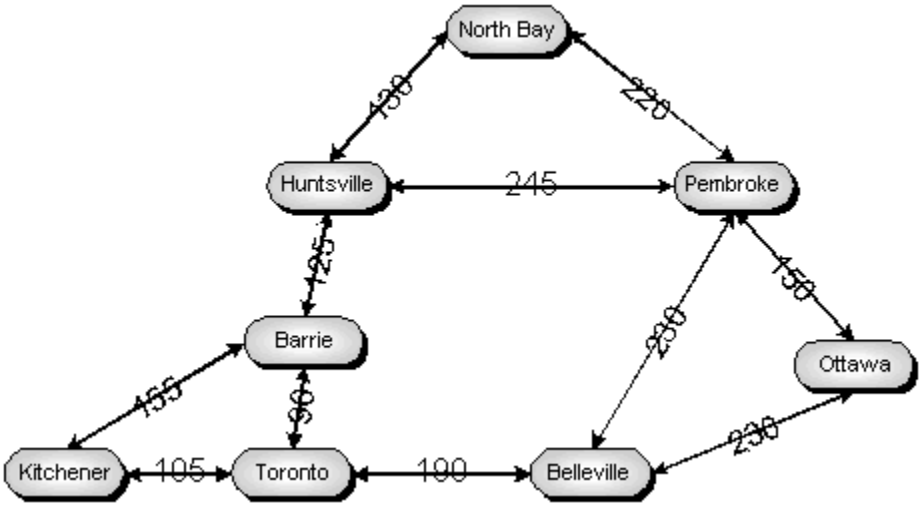
如果转换初始子查询中的数据类型，查询的行为可以像预期一样：

```
WITH RECURSIVE parts ( component, subcomponent, quantity ) AS
( SELECT CAST( NULL AS VARCHAR ),   -- CASTs must be used
         'bookcase',                -- to declare the
         CAST( 1   AS INT )         -- correct datatypes
      UNION ALL
  SELECT b.component, b.subcomponent,
         p.quantity * b.quantity
  FROM parts p JOIN bookcase b
    ON p.subcomponent = b.component )
SELECT * FROM parts
ORDER BY component, subcomponent;
```

最短距离问题

您可以使用递归公用表表达式在有向图上查找所需的路径。数据库表中的每一行表示一个有向边。每一行指定一个原点、一个终点，以及从原点到达终点的开销。根据问题的不同，开销可能表示距离、旅行时间或一些其它测量单位。使用递归，您可以通过此图形查看可能的路线。然后，您可以从可能的路线集中选择您感兴趣的路线。

例如，以查找 Kitchener 市和 Pembroke 市之间的可取驱车路线的问题为例。可能的路线有多条，每一条都会使您经过一组不同的中间城市。目标是找到最短的路线，以及将它们与合理的备用方案进行比较。



首先，定义一个表来表示此图形的边并为每个边插入一行。由于此图形的所有边都恰巧是双向的，所以也必须插入表示相反方向的边。通过选择初始行集可以实现这一点，但要将原点和终点互换。例如，一行必须表示从 Kitchener 到 Toronto 的行程，另一行表示从 Toronto 返回 Kitchener 的行程。

```
CREATE TABLE travel (
    origin      VARCHAR(10),
    destination VARCHAR(10),
    distance    INT,
  PRIMARY KEY ( origin, destination )
);
INSERT INTO travel
  SELECT 'Kitchener',  'Toronto',     105 UNION
  SELECT 'Kitchener',  'Barrie',      155 UNION
  SELECT 'North Bay',  'Pembroke',    220 UNION
  SELECT 'Pembroke',   'Ottawa',      150 UNION
  SELECT 'Barrie',     'Toronto',      90 UNION
  SELECT 'Toronto',    'Belleville',  190 UNION
  SELECT 'Belleville', 'Ottawa',      230 UNION
  SELECT 'Belleville', 'Pembroke',    230 UNION
  SELECT 'Barrie',     'Huntsville',  125 UNION
  SELECT 'Huntsville', 'North Bay',   130 UNION
  SELECT 'Huntsville', 'Pembroke',    245;
INSERT INTO travel   -- Insert the return trips
  SELECT destination, origin, distance
  FROM travel;
```


下一个任务是编写递归公用表表达式。由于旅程将从 Kitchener 开始，因此初始子查询会先选择从 Kitchener 出发的所有可能路线以及每一条路线的距离。

递归子查询可延伸路线。对于每一条路线，它都会添加从前面路段的终点继续延伸的路段，这样就增加了新路段的长度，从而可以保持每条路线的总运行开销。为了提高效率，如果路线满足以下任一条件，就会终止：

- 路径返回到起始位置。
- 路径返回到前面的位置。
- 路径到达最后的终点。

在当前示例中，任何路径都不应返回到 Kitchener，并且，如果到达 Pembroke，所有路径都应终止。

使用递归查询来研究循环图时，验证它们的结束条件是否完全正确非常重要。在这种情况下，由于一条路线可以包括两个中间城市间的任意大的往返行程数，因此上面的条件是不够的。下面的递归查询通过将任意给定路线中的最大路段数限制为七段来保证终止。

由于示例查询的关键是选择可行路线，因此主查询只选择比最短路线长小于 50% 的那些路线。

```
WITH RECURSIVE
    trip ( route, destination, previous, distance, segments ) AS
( SELECT CAST( origin || ', ' || destination AS VARCHAR(256) ),
         destination, origin, distance, 1
  FROM travel
  WHERE origin = 'Kitchener'
    UNION ALL
  SELECT route || ', ' || v.destination,
         v.destination,             -- current endpoint
         v.origin,                  -- previous endpoint
         t.distance + v.distance,   -- total distance
         segments + 1               -- total number of segments
  FROM trip t JOIN travel v ON t.destination = v.origin
  WHERE v.destination <> 'Kitchener'  -- Don't return to start
    AND v.destination <> t.previous   -- Prevent backtracking
    AND v.origin      <> 'Pembroke'   -- Stop at the end
    AND segments                      -- TERMINATE RECURSION!
          < ( SELECT count(*)/2 FROM travel ) )
SELECT route, distance, segments FROM trip
WHERE destination = 'Pembroke' AND
      distance < 1.5 * ( SELECT MIN( distance )
                         FROM trip
                         WHERE destination = 'Pembroke' )
ORDER BY distance, segments, route;
```

当针对上面的数据集运行此语句时，会生成下面的结果。

route	distance	segments
Kitchener, Barrie, Huntsville, Pembroke	525	3
Kitchener, Toronto, Belleville, Pembroke	525	3
Kitchener, Toronto, Barrie, Huntsville, Pembroke	565	4
Kitchener, Barrie, Huntsville, North Bay, Pembroke	630	4

route	distance	segments
Kitchener, Barrie, Toronto, Belleville, Pembroke	665	4
Kitchener, Toronto, Barrie, Huntsville, North Bay, Pembroke	670	5
Kitchener, Toronto, Belleville, Ottawa, Pembroke	675	4

使用多个递归公用表表达式

一个递归查询可以包括多个递归查询，只要这些查询没有交集。它还可以混合包括递归公用表表达式和非递归公用表表达式。如果至少有一个公用表表达式是递归的，则必须有 RECURSIVE 关键字。

例如，以下查询—它与上一个查询返回的结果相同—使用第二个非递归公用表表达式选择最短路线的长度。第二个公用表表达式的定义通过逗号与第一个公用表表达式的定义分隔开。

```
WITH RECURSIVE
  trip ( route, destination, previous, distance, segments ) AS
    ( SELECT CAST( origin || ', ' || destination AS VARCHAR(256) ),
             destination, origin, distance, 1
      FROM travel
      WHERE origin = 'Kitchener'
      UNION ALL
      SELECT route || ', ' || v.destination,
             v.destination,
             v.origin,
             t.distance + v.distance,
             segments + 1
      FROM trip t JOIN travel v ON t.destination = v.origin
      WHERE v.destination <> 'Kitchener'
        AND v.destination <> t.previous
        AND v.origin      <> 'Pembroke'
        AND segments
              < ( SELECT count(*)/2 FROM travel ) ),
  shortest ( distance ) AS                    -- Additional,
    ( SELECT MIN(distance)                    -- non-recursive
      FROM trip                               -- common table
      WHERE destination = 'Pembroke' )        -- expression
SELECT route, distance, segments FROM trip
WHERE destination = 'Pembroke' AND
      distance < 1.5 * ( SELECT distance FROM shortest )
ORDER BY distance, segments, route;
```

像非递归公用表表达式一样，递归表达式在用于存储过程中时可以包含对局部变量或过程参数的引用。例如，下面定义的 best_routes 过程可识别两个指定城市间的最短路线。

```
CREATE PROCEDURE best_routes (
   IN initial VARCHAR(10),
   IN final   VARCHAR(10)
)
BEGIN
  WITH RECURSIVE
      trip ( route, destination, previous, distance, segments ) AS
  ( SELECT CAST( origin || ', ' || destination AS VARCHAR(256) ),
           destination, origin, distance, 1
    FROM travel
    WHERE origin = initial
      UNION ALL
    SELECT route || ', ' || v.destination,
           v.destination,             -- current endpoint
           v.origin,                  -- previous endpoint
           t.distance + v.distance,   -- total distance
           segments + 1               -- total number of segments
    FROM trip t JOIN travel v ON t.destination = v.origin
    WHERE v.destination <> initial     -- Don't return to start
      AND v.destination <> t.previous  -- Prevent backtracking
      AND v.origin      <> final       -- Stop at the end
      AND segments                     -- TERMINATE RECURSION!
```

```
            < ( SELECT count(*)/2 FROM travel ) )
  SELECT route, distance, segments FROM trip
  WHERE destination = final AND
        distance < 1.4 * ( SELECT MIN( distance )
                           FROM trip
                           WHERE destination = final )
  ORDER BY distance, segments, route;
END;
```

以下语句调用上述过程。

```
CALL best_routes ( 'Pembroke', 'Kitchener' );
```


OLAP 支持

目录

联机分析处理（On-Line Analytical Processing，简称 OLAP）具有在一条 SQL 语句中执行复杂数据分析的功能，从而通过减少对数据库的查询次数来改进性能，而同时又能增加结果值。通过使用对 SQL 语句和窗口函数的扩展，可在 SQL Anywhere 中使用 OLAP 功能。这些 SQL 扩展和函数能够以一种简明的方式实现多维数据分析、数据挖掘、时间序列分析、趋势分析、开销分配、目标寻求以及异常警告，而这些又常常是通过一条 SQL 语句完成的。

- **SELECT 语句的扩展** 利用 SELECT 语句的扩展，您可以将输入行分组、对组执行分析以及将结果加入到最终结果集中。这些扩展包括对 GROUP BY 子句（GROUPING SETS、CUBE 和 ROLLUP 子句）以及 WINDOW 子句的扩展。

  利用 GROUP BY 子句的扩展，您可以采用多种方式来划分输入行，从而生成一个可将不同组连接在一起的结果集。您还可以创建一个稀疏多维结果集（也称为**数据立方体**）以实现数据挖掘分析。最后，这些扩展还提供了小计行和总计行，使得分析起来更为方便。请参见“GROUP BY 子句扩展”一节第 426 页。

  将 WINDOW 子句与窗口函数结合使用，能够对成组的输入行进行其它分析。请参见“窗口函数”一节第 436 页。

- **窗口集合函数** 几乎所有的 SQL Anywhere 集合函数均支持可配置滑动**窗口**概念，在处理输入行时该窗口会随着输入行向下移动。在窗口移动时可对其中的数据执行附加计算，这样便能够以一种比使用语义上等同的自连接查询或相关子查询更为有效的方式进行进一步分析。

  例如，窗口集合函数以及 GROUP BY 子句的 CUBE、ROLLUP 和 GROUPING SETS 扩展能够在一条 SQL 语句中有效地计算百分点、移动平均值、累计总和，否则可能会需要自连接、相关子查询、临时表或者所有这三项的某个组合。

  您可以使用窗口集合函数获得诸如此类信息：道琼斯工业平均指数的季度移动平均线或者各部门所有雇员及其累计薪资。还可以利用它们来计算方差、标准差、相关和回归测量。请参见“窗口集合函数”一节第 442 页。

- **窗口秩函数** 通过窗口秩函数可组成单语句的 SQL 查询以获得以下信息，如今年发运的产品中其总销售额排名前 10 的产品，或者将订单至少销售给 15 个不同公司的销售人员中排名前 5% 的销售人员。请参见“窗口秩函数”一节第 460 页。

提高 OLAP 性能

要提高 OLAP 性能，请将 optimization_workload 数据库选项设置为 OLAP，以指示优化程序在它调查的可能性中考虑使用 "聚簇散列分组依据" 运算符。也可在定义 OLAP 负载的索引时使用 FOR OLAP WORKLOAD 选项调整此索引。使用此选项会促使数据库服务器执行一定的优化，其中包括维护由 "聚簇散列分组依据" 运算符所使用的关于同一密钥中两行之间最大页面距离的统计信息。

另请参见

- “optimization_workload 选项 [数据库]”一节 《SQL Anywhere 服务器 - 数据库管理》
- “ClusteredHashGroupBy 算法 (GrByHClust)”一节第 560 页
- “CREATE INDEX 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “CREATE TABLE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “ALTER TABLE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》

GROUP BY 子句扩展

利用 SELECT 语句的标准 GROUP BY 子句，您可以按照提供的分组表达式对结果集中的行进行分组。例如，如果您指定 GROUP BY columnA, columnB，则这些行将按照 columnA 和 columnB 中唯一值的组合进行分组。在标准 GROUP BY 子句中，这些组反映对所有指定的 GROUP BY 表达式的组合所做的计算。

不过，您可能想要为结果集指定不同分组或子分组。例如，您可能希望结果中的数据先按 columnA 和 columnB 的唯一值进行分组，然后按 columnC 的唯一值再次重新分组。可通过对 GROUP BY 子句使用 GROUPING SETS 扩展实现此结果。

GROUP BY GROUPING SETS

GROUPING SETS 子句是 SELECT 语句的 GROUP BY 子句的扩展。通过 GROUPING SETS 子句，您可采用多种方式对结果分组，而不必使用多个 SELECT 语句来实现这一目的。这就意味着，能够减少响应时间并提高性能。

例如，以下两条查询语句在语义上是等效的。不过，第二个查询通过使用 GROUP BY GROUPING SETS 子句能够更有效地定义分组条件。

使用多个 SELECT 语句的多个分组：

```
SELECT NULL, NULL, NULL, COUNT( * ) AS Cnt
FROM Customers
WHERE State IN ( 'MB' , 'KS' )
   UNION ALL
SELECT City, State, NULL, COUNT( * ) AS Cnt
FROM Customers
WHERE State IN ( 'MB' , 'KS' )
GROUP BY City, State
   UNION ALL
SELECT NULL, NULL, CompanyName, COUNT( * ) AS Cnt
FROM Customers
WHERE State IN ( 'MB' , 'KS' )
GROUP BY CompanyName;
```

使用 GROUPING SETS 的多个分组：

```
SELECT City, State, CompanyName, COUNT( * ) AS Cnt
FROM Customers
WHERE State IN ( 'MB' , 'KS' )
GROUP BY GROUPING SETS( ( City, State ), ( CompanyName ) , ( ) );
```

两种方法均产生相同的结果，如下所示：

	City	State	CompanyName	Cnt
1	(NULL)	(NULL)	(NULL)	8
2	(NULL)	(NULL)	Cooper Inc.	1
3	(NULL)	(NULL)	Westend Dealers	1

	City	State	CompanyName	Cnt
4	(NULL)	(NULL)	Toto's Active Wear	1
5	(NULL)	(NULL)	North Land Trading	1
6	(NULL)	(NULL)	The Ultimate	1
7	(NULL)	(NULL)	Molly's	1
8	(NULL)	(NULL)	Overland Army Navy	1
9	(NULL)	(NULL)	Out of Town Sports	1
10	'Pembroke'	'MB'	(NULL)	4
11	'Petersburg'	'KS'	(NULL)	1
12	'Drayton'	'KS'	(NULL)	3

第 2-9 行是按照 CompanyName 分组生成的行，第 10-12 行是按照 City 和 State 的组合进行分组所生成的行，第 1 行是空分组集所表示的总计，它是使用一对成对的圆括号 () 指定的。空分组集表示 GROUP BY 输入中所有行的单个分区。

请注意 NULL 值如何在分组集中不使用的表达式中充当占位符，因为这些结果集必须可以组合。例如，第 2-9 行由查询 (CompanyName) 中的第二个分组集得到。因为分组集未将 City 或 State 作为表达式包含在内，所以对于第 2-9 行，City 和 State 的值中会含有占位符 NULL，而 CompanyName 中的值将含有在 CompanyName 中找到的明确值。

因为 NULL 用作占位符，所以很容易将占位符 NULL 与数据中找到的真正的 NULL 相混淆。为有助于将占位符 NULL 与 NULL 数据区分开来，请使用 GROUPING 函数。请参见“使用 GROUPING 函数检测占位符 NULL”一节第 433 页。

示例

下面的示例说明了如何使用 GROUPING SETS 定制从查询返回的结果，以及如何使用 ORDER BY 子句更好地组织这些结果。以下查询将按各年份 (Year) 中的季度 (Quarter) 返回订单总数以及各年份 (Year) 的总数。先按年份 (Year) 排序，再按季度 (Quarter) 排序可使结果更易于理解：

```
SELECT Year( OrderDate ) AS Year,
       Quarter( OrderDate ) AS Quarter,
       COUNT (*) AS Orders
FROM SalesOrders
GROUP BY GROUPING SETS ( ( Year, Quarter ), ( Year ) )
ORDER BY Year, Quarter;
```

此查询会返回以下结果：

	Year	Quarter	Orders
1	2000	(NULL)	380

	Year	Quarter	Orders
2	2000	1	87
3	2000	2	77
4	2000	3	91
5	2000	4	125
6	2001	(NULL)	268
7	2001	1	139
8	2001	2	119
9	2001	3	10

第 1 行和第 6 行分别是 2000 年和 2001 年的订单数小计。第 2-5 行和第 7-9 行是小计行的详细信息行。也就是说，它们按年、按季度显示订单总数。

结果集中没有所有年份中所有季度的总计。要实现此目的，查询必须在 GROUPING SETS 说明中包括空分组说明 '()'。

指定空分组说明

如果在 GROUP BY 子句中使用空 GROUPING SETS 说明 '()'，则会产生一个总计行，对结果中的所有项进行总计。使用总计行时，所有分组表达式的所有值均会包含占位符 NULL。可使用 GROUPING 函数将占位符 NULL 与计算行底层数据中的值后产生的实际 NULL 值区分开来。请参见“使用 GROUPING 函数检测占位符 NULL”一节第 433 页。

指定重复分组集

可在 GROUPING SETS 子句中指定重复分组说明。此时，SELECT 语句的结果将包含相同的行。

以下查询包括重复分组：

```
SELECT City, COUNT( * ) AS Cnt
FROM Customers
WHERE State IN ( 'MB' , 'KS' )
GROUP BY GROUPING SETS( ( City ), ( City ) );
```

此查询会返回以下结果。请注意，由于重复分组的原因，第 1-3 行与第 4-6 行相同：

	City	Cnt
1	'Drayton'	3
2	'Petersburg'	1
3	'Pembroke'	4

	City	Cnt
4	'Drayton'	3
5	'Petersburg'	1
6	'Pembroke'	4

实践良好的格式

GROUP BY GROUPING SETS 子句的分组语法的解释方式不同于简单的 GROUP BY 子句。例如，GROUP BY (X, Y) 返回按 X 和 Y 值的唯一组合进行分组的结果。而 GROUP BY GROUPING SETS (X, Y) 将指定两个单独的分组集，且这两个分组的结果会结合在一起。也就是说，结果将按 (X) 分组，然后合并到按 (Y) 分组的相同结果中。

为使格式良好，并在表达式复杂的情况下避免造成任何歧义，请在有可能出错的各种情况下，为说明中的每个单独分组集括上括号。例如，尽管以下两条语句都是正确的，且语义上等效，但建议采用第二种格式：

```
SELECT * FROM t GROUP BY GROUPING SETS ( X, Y );
SELECT * FROM t GROUP BY GROUPING SETS( ( X ), ( Y ) );
```

使用 ROLLUP 和 CUBE 作为 GROUPING SETS 的快捷方式

需要将几个不同的数据分区连接到单个结果集中时，使用 GROUPING SETS 十分有用。不过，如果要指定多个分组并包括小计，则可能要用到 ROLLUP 和 CUBE 扩展。

ROLLUP 和 CUBE 可视为预定义的 GROUPING SETS 说明的快捷方式。

ROLLUP 等效于指定一系列以空分组集 '()' 开始，依次接续其中将另一表达式与前一表达式相连的分组集的分组集说明。例如，如果有三个分组表达式 a、b 和 c，并且指定 ROLLUP，这就相当于指定具有以下集的 GROUPING SETS 子句：()、(a)、(a, b) 和 (a, b, c)。这种结构有时也称为分层分组。

CUBE 提供的分组更多。指定 CUBE 与指定所有可行的 GROUPING SETS 等效。例如，如果有相同的三个分组表达式 a、b 和 c，并且指定 CUBE，这就相当于指定具有以下集的 GROUPING SETS 子句：()、(a)、(a, b)、(a, c)、(b)、(b, c)、(c) 和 (a, b, c)。

指定 ROLLUP 或 CUBE 时，请使用 GROUPING 函数区分结果中的占位符 NULL，它是由隐含在 ROLLUP 或 CUBE 所形成的结果集中的小计行造成的。请参见“使用 GROUPING 函数检测占位符 NULL”一节第 433 页。

使用 ROLLUP

许多应用程序的一个常见要求是从左到右依次计算分组属性的小计。此模式称为层次，因为引入附加小计计算会额外产生信息详细程度更佳的行。在 SQL Anywhere 中，可使用用以指定 ROLLUP 子句的 ROLLUP 关键字来指定分组属性的层次。

使用 ROLLUP 子句的查询会产生如下的分组集分层系列。如果 ROLLUP 子句含有 *n* 个 GROUP BY 表达式，其格式为 $(X_1,X_2, ..., X_n)$，则 ROLLUP 子句将生成 *n* + 1 个分组集，如下所示：

```
{(), (X1), (X1,X2), (X1,X2,X3), ... , (X1,X2,X3, ... , Xn)}
```

示例

以下查询按年份和季度对销售订单进行了汇总，并返回下表中所示的结果集：

```
SELECT QUARTER( OrderDate ) AS Quarter,
       YEAR( OrderDate ) AS Year,
       COUNT( * ) AS Orders,
       GROUPING( Quarter ) AS GQ,
       GROUPING( Year ) AS GY
FROM SalesOrders
GROUP BY ROLLUP( Year, Quarter )
ORDER BY Year, Quarter;
```

此查询会返回以下结果：

	Quarter	Year	Orders	GQ	GY
1	(NULL)	(NULL)	648	1	1

	Quarter	Year	Orders	GQ	GY
2	(NULL)	2000	380	1	0
3	1	2000	87	0	0
4	2	2000	77	0	0
5	3	2000	91	0	0
6	4	2000	125	0	0
7	(NULL)	2001	268	1	0
8	1	2001	139	0	0
9	2	2001	119	0	0
10	3	2001	10	0	0

结果集中的第一行显示两年中所有季度全部订单的总计 (648)。

第 2 行显示 2000 年的订单总数 (380)，而第 3-6 行显示同年各季度的订单小计。同理，第 7 行显示 2001 年的订单总数 (268)，而第 8-10 行显示同年各季度的订单小计。

注意 GROUPING 函数返回的值是如何用于区分包含总计的行和小计行的。对于第 2 行和第 7 行，季度列中出现 NULL 且 GQ 列（Grouping by Quarter，按季度分组）的值为 1，这表示该行为（每年）所有季度的订单总计。

同理，在第 1 行中，季度 (Quarter) 和年份 (Year) 列中出现 NULL 且 GQ 和 GY 列中出现值 1，这表示该行为所有季度和所有年份的订单总计。

有关 ROLLUP 子句语法的详细信息，请参见“GROUP BY 子句”一节 《SQL Anywhere 服务器 - SQL 参考》。

支持 T-SQL WITH ROLLUP 语法

或者，也可使用 Transact-SQL 兼容语法 WITH ROLLUP 获得与 GROUP BY ROLLUP 相同的结果。不过，该语法稍有不同，只可在此语法中提供简单的 GROUP BY 表达式列表。

以下查询将产生与先前的 GROUP BY ROLLUP 示例相同的结果：

```
SELECT QUARTER( OrderDate ) AS Quarter,
       YEAR( OrderDate ) AS Year,
       COUNT( * ) AS Orders,
       GROUPING( Quarter ) AS GQ,
       GROUPING( Year ) AS GY
FROM SalesOrders
GROUP BY Year, Quarter WITH ROLLUP
ORDER BY Year, Quarter;
```

使用 CUBE

除了 ROLLUP 子句提供的分层分组模式外，您还可以通过 CUBE 子句创建数据立方体，即通过 GROUP BY 表达式的每种可能组合对输入进行 *n* 维汇总。CUBE 子句最终会生成每组值的元素的所有可能组合的积集。这在进行复杂数据分析时会很有用。

如果 CUBE 子句中有 *n* 个 GROUPING 表达式，其格式为 $(X_1,X_2, ...,X_n)$，则 CUBE 将生成 2^n 个分组集，如下所示：

```
{(), (X1), (X1,X2), (X1,X2,X3), ... , (X1,X2,X3, ...,Xn),
(X2), (X2,X3), (X2,X3,X4), ... , (X2,X3,X4, ... , Xn), ... , (Xn)}.
```

示例

以下查询按年份、季度和年份中每季对销售订单进行了汇总，并生成了下表中所示的结果集：

```
SELECT QUARTER( OrderDate ) AS Quarter,
       YEAR( OrderDate ) AS Year,
       COUNT( * ) AS Orders,
       GROUPING( Quarter ) AS GQ,
       GROUPING( Year ) AS GY
FROM SalesOrders
GROUP BY CUBE ( Year, Quarter )
ORDER BY Year, Quarter;
```

此查询会返回以下结果：

	Quarter	Year	Orders	GQ	GY
1	(NULL)	(NULL)	648	1	1
2	1	(NULL)	226	0	1
3	2	(NULL)	196	0	1
4	3	(NULL)	101	0	1
5	4	(NULL)	125	0	1
6	(NULL)	2000	380	1	0
7	1	2000	87	0	0
8	2	2000	77	0	0
9	3	2000	91	0	0
10	4	2000	125	0	0
11	(NULL)	2001	268	1	0
12	1	2001	139	0	0

	Quarter	Year	Orders	GQ	GY
13	2	2001	119	0	0
14	3	2000	10	0	0

结果集中的第一行显示 2000 和 2001 两年中所有季度全部订单的总计 (648)。

第 2-5 行按任何年份中的日历季度对销售订单进行了汇总。

第 6 和 11 行分别显示 2000 和 2001 年的总订单数。

第 7-10 行和第 12-14 行分别显示 2000 和 2001 年的季度订单总数。

注意 GROUPING 函数返回的值是如何用于区分包含总计的行和小计行的。对于第 6 行和第 11 行，季度 (Quarter) 列中出现 NULL 且 GQ 列的值为 1（Grouping by Quarter，按季度分组），这表示该行为该年所有季度的订单总计。

注意
使用 CUBE 生成的结果集可能会非常大，因为 CUBE 呈指数方式生成分组集。为此，SQL Anywhere 不允许 GROUP BY 子句包含的 GROUP BY 表达式超过 64 个。如果语句超出此限制，则语句将失败，并且会报告错误 SQLCODE -944 (SQLSTATE 42WA1)。

有关 CUBE 子句语法的详细信息，请参见“GROUP BY 子句”一节 《SQL Anywhere 服务器 - SQL 参考》。

支持 T-SQL WITH CUBE 语法

或者，也可使用 Transact-SQL 兼容语法 WITH CUBE 实现与 GROUP BY CUBE 相同的结果。不过，该语法稍有不同，只可在此语法中提供简单的 GROUP BY 表达式列表。

以下查询将产生与先前的 GROUP BY CUBE 示例相同的结果：

```
SELECT QUARTER( OrderDate ) AS Quarter,
       YEAR( OrderDate ) AS Year,
       COUNT( * ) AS Orders,
       GROUPING( Quarter ) AS GQ,
       GROUPING( Year ) AS GY
FROM SalesOrders
GROUP BY Year, Quarter WITH CUBE
ORDER BY Year, Quarter;
```

使用 GROUPING 函数检测占位符 NULL

ROLLUP 和 CUBE 创建的总计行和小计行在未用于分组的 SELECT 列表中指定的任意列中均含有占位符 NULL。这意味着，您在检查结果时，将无法区分小计行中的 NULL 是占位符 NULL 还是计算行的底层数据时所得的 NULL。结果，同样很难区分详细信息行、小计行和总计行。

通过 GROUPING 函数可将占位符 NULL 与底层数据产生的 NULL 区分开来。如果通过分组集说明中的一个 *group-by-expression* 指定 GROUPING 函数，则在结果为占位符 NULL 时该函数将返回 1，而在结果反映该行底层数据中的某个值（或许为 NULL）时返回 0。

例如，以下查询将返回下表中所示的结果集：

```
SELECT Employees.EmployeeID AS Employee,
   YEAR( OrderDate ) AS Year,
   COUNT( SalesOrders.ID ) AS Orders,
   GROUPING( Employee ) AS GE,
   GROUPING( Year ) AS GY
FROM Employees LEFT OUTER JOIN SalesOrders
   ON Employees.EmployeeID = SalesOrders.SalesRepresentative
WHERE Employees.Sex IN ( 'F' )
   AND Employees.State IN ( 'TX' , 'NY' )
GROUP BY GROUPING SETS ( ( Year, Employee ), ( Year ), ( ) )
ORDER BY Year, Employee;
```

此查询会返回以下结果：

	Employees	Year	Orders	GE	GY
1	(NULL)	(NULL)	54	1	1
2	(NULL)	(NULL)	0	1	0
3	102	(NULL)	0	0	0
4	390	(NULL)	0	0	0
5	1062	(NULL)	0	0	0
6	1090	(NULL)	0	0	0
7	1507	(NULL)	0	0	0
8	(NULL)	2000	34	1	0
9	667	2000	34	0	0
10	(NULL)	2001	20	1	0
11	667	2001	20	0	0

在本例中，因为指定了空分组集 '()'，所以第 1 行表示订单数总计 (54)。请注意，GE 和 GY 均含有 1，这表明雇员 (Employees) 和年份 (Year) 列中的 NULL 分别为雇员 (Employees) 和年份 (Year) 列的占位符 NULL。

第 2 行为小计行。GE 列中的 1 表示雇员 (Employees) 列中的 NULL 是占位符 NULL。GY 列中的 0 表示年份 (Year) 列中的 NULL 为计算底层数据所产生的结果，而不是占位符 NULL；这时，此行表示没有订单的雇员。

第 3-7 行显示了年份 (Year) 列为 NULL 时每位雇员的订单总数。即，这些行是居住在德克萨斯和纽约且没有订单的女雇员。这些行是第 2 行的详细信息行。即，第 2 行为第 3-7 行的总计。

第 8 行为显示 2000 年中所有雇员的订单数的小计行。第 9 行为第 8 行的详细信息单行。

第 10 行为显示 2001 年中所有雇员的订单数的小计行。第 11 行为第 10 行的详细信息单行。

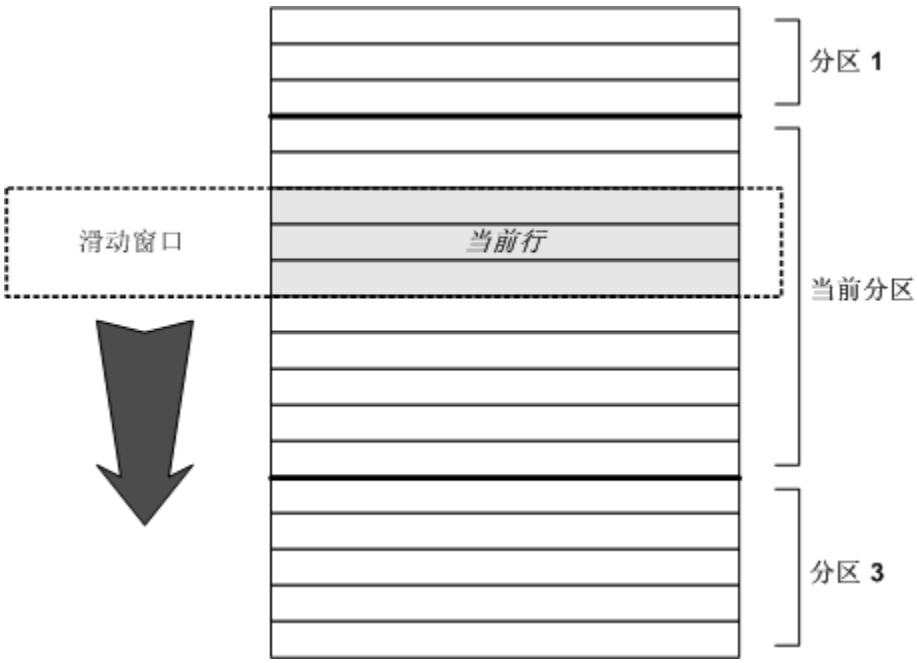
有关 GROUPING 函数的语法的详细信息，请参见“GROUPING 函数 [Aggregate]”一节 《SQL Anywhere 服务器 - SQL 参考》。

窗口函数

OLAP 功能中具有滑动**窗口**概念，即在处理输入行时该窗口会向下移动。在窗口移动时可对其中的数据进行附加计算，这样便能够以一种比使用（语义上）等同于自连接查询或相关子查询更为有效的方式进行进一步分析。

可基于要从数据中抽取的信息来配置窗口边界。窗口可以是输入数据中的一行、多行或所有行，该输入数据已根据窗口定义中提供的分组说明进行了分区。窗口向下遍历输入数据，合并执行所请求计算所需的行。

下图所示为处理输入行时窗口的移动。数据分区反映出窗口定义中指定的输入行分组。如果未指定分组，则将所有输入行视为一个分区。窗口的长度（即所包含的行数）和窗口相对于当前行的偏移反映出窗口定义中所指定的边界。



定义窗口

可以使用 SQL 窗口扩展来配置窗口边界，以及输入行的分区和排序。逻辑上，分区作为计算查询说明结果的语义的一部分，将在 GROUP BY 子句所定义的组创建后，在对最终 SELECT 列表和查询的 ORDER BY 子句进行计算之前创建。因此，SQL 语句中子句的计算顺序为：

1. FROM
2. WHERE
3. GROUP BY

4. HAVING
5. WINDOW
6. DISTINCT
7. ORDER BY

构建查询时，应考虑计算顺序的影响。例如，在引用同一 SELECT 查询块中窗口函数的表达式中不能没有谓语。然而，通过在派生表中放置查询块，就可以在派生表中指定谓语。以下查询失败，并发出一条消息，指出导致失败的原因是在窗口函数中指定了谓语：

```
SELECT DepartmentID, Surname, StartDate, Salary,
       SUM( Salary ) OVER ( PARTITION BY DepartmentID
          ORDER BY StartDate
          RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW ) AS
"Sum_Salary"
   FROM Employees
   WHERE State IN ( 'CA', 'UT', 'NY', 'AZ' )
      AND DepartmentID IN ( '100', '200' )
   GROUP BY DepartmentID, Surname, StartDate, Salary
   HAVING Salary > 0 AND "Sum_Salary" > 200
   ORDER BY DepartmentID, StartDate;
```

使用派生表 (DT) 并在其中指定谓语以获得所需的结果：

```
SELECT * FROM ( SELECT DepartmentID, Surname, StartDate, Salary,
                       SUM( Salary ) OVER ( PARTITION BY DepartmentID
                         ORDER BY StartDate
                         RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW )
AS "Sum_Salary"
                  FROM Employees
                  WHERE State IN ( 'CA', 'UT', 'NY', 'AZ' )
                     AND DepartmentID IN ( '100', '200' )
                  GROUP BY DepartmentID, Surname, StartDate, Salary
                  HAVING Salary > 0
                  ORDER BY DepartmentID, StartDate ) AS DT
   WHERE DT.Sum_Salary > 200;
```

因为窗口分区跟在 GROUP BY 运算符之后，所以任何集合函数（如 SUM、AVG 或 VARIANCE）的结果均可用于针对分区所进行的计算。因此，除了通过查询的 GROUP BY 和 ORDER BY 子句执行分组和排序操作外，窗口还提供了另外一种方法。

定义窗口说明

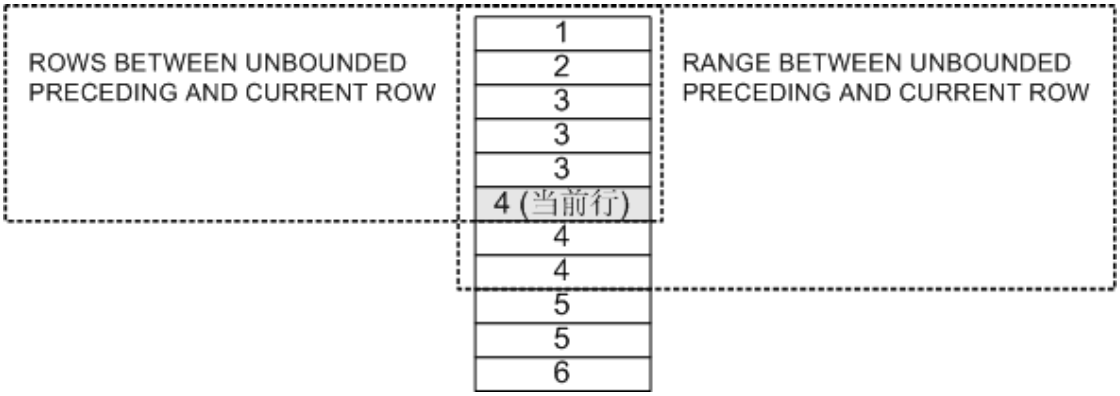
定义在其上运行窗口函数的窗口时，应指定以下一项或多项：

- **分区（PARTITION BY 子句）** PARTITION BY 子句定义输入行的分组方式。如果省略，则将整个输入视为单个分区。分区可以是一个、多个或所有输入行，这视您指定的内容而定。来自两个分区的数据从不混合。也就是说，当窗口到达两分区间的边界时，它先处理完一个分区中的数据，然后再开始下一分区中数据的处理。这意味着窗口大小在分区的开始和结束处可能有所变化，这要根据窗口边界如何进行的定义而定。
- **排序（ORDER BY 子句）** ORDER BY 子句定义输入行在由窗口函数处理之前如何进行排序。ORDER BY 子句仅在使用 RANGE 子句指定边界时或秩函数引用窗口时才必须使用。否则，ORDER BY 子句是可选的。如果省略，数据库服务器将以最有效的方式处理输入行。

- **边界（RANGE 和 ROWS 子句）** 当前行为确定窗口的开始行和结束行提供参考点。可以使用窗口定义的 RANGE 和 ROWS 子句设置这些边界。RANGE 根据偏移当前行值的*数据值范围*定义窗口。因此，如果指定 RANGE，还必须指定 ORDER BY 子句，因为范围计算要求对数据进行排序。

  ROWS 根据偏移当前行的*行数*定义窗口。

  由于 RANGE 根据数据值范围定义一组行，所以 RANGE 窗口中包含的行可包括超出当前行的行。这与 ROWS 的处理方式有所不同。下图说明了 ROWS 子句与 RANGE 子句之间的差异：



  在 ROWS 和 RANGE 子句中，可以（有选择地）指定窗口的开始行和结束行（相对于当前行）。为此，可使用 PRECEDING、BETWEEN 和 FOLLOWING 子句。这些子句采用表达式以及关键字 UNBOUNDED 和 CURRENT ROW。如果未给窗口定义边界，则缺省的窗口边界设置如下：

  - 如果窗口说明中含有 ORDER BY 子句，则相当于指定 RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW。
  - 如果窗口说明中不含有 ORDER BY 子句，则相当于指定 ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING。

  下表包含了一些示例窗口边界及其所包含的行的说明：

说明	含义
ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW	从分区起始处开始，以当前行结束。当计算累计结果（如累计总和）时使用此示例。
ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING	使用分区中的所有行。当希望集合函数的值与分区中每行相同时，使用此示例。

说明	含义
ROWS BETWEEN *x* PRECEDING AND *y* FOLLOWING	创建一个大小固定的移动窗口，窗口的行从距离当前行 *x* 处开始，在距离当前行 *y* 处结束（包括起始行和结束行）。在您要计算移动平均值或者要计算相邻行的差值时，可使用此示例。 对于包括多个行的移动窗口，当计算分区中第一行或最后一行时会出现 NULL 值。之所以出现这种情况，是因为当前行是分区的第一行或最后一行时，该行没有相应的上一行或下一行可以在计算中使用。因此，使用 NULL 值来代替。
ROWS BETWEEN CURRENT ROW AND CURRENT ROW	具有一行（即当前行）的窗口。
RANGE BETWEEN 5 PRECEDING AND 5 FOLLOWING	创建一个以行中的值为基础的窗口。例如，假定 ORDER BY 子句为当前行所指定的列中包含值 10。如果指定窗口大小为 `RANGE BETWEEN 5 PRECEDING AND 5 FOLLOWING`，则所指定的窗口大小，将会确保窗口中的第一行在列中包含 5，最后一行在列中包含 15。窗口在分区中向下移动时，窗口大小可能会相应满足范围说明所需的尺寸而增大或缩小。

使窗口说明尽可能明确。否则，缺省值可能不会返回期望的结果。

在一组值不连续时，使用 RANGE 子句可以避免由于窗口函数的输入有间距而产生的问题。使用 RANGE 子句设置窗口边界时，数据库服务器会自动处理相邻行以及包含重复值的行。

RANGE 使用无符号的整数值。根据 ORDER BY 表达式的域和 RANGE 子句所指定的值域而定，可能会出现截断范围表达式的情况。

当使用秩函数或行编号函数时，不要指定窗口边界。

窗口定义：内置与 WINDOW 子句

定义窗口的方法有三种：

- 内置（在窗口函数的 OVER 子句中）
- 在 WINDOW 子句中
- 部分内置，部分在 WINDOW 子句中

然而，有些方法存在限制，如以下各节中所述。

内置定义

窗口定义可以放在窗口函数的 OVER 子句中。这称为以*内置*方式定义窗口。

例如，以下语句在 SQL Anywhere 示例数据库中查询 2001 年七月和八月发运的所有产品，以及截至发运日期的累计发运量。窗口采用内置方式定义。

```
SELECT p.ID, p.Description, s.Quantity, s.ShipDate,
   SUM( s.Quantity ) OVER ( PARTITION BY s.ProductID
       ORDER BY s.ShipDate
       ROWS BETWEEN UNBOUNDED PRECEDING
       AND CURRENT ROW ) AS Cumulative_qty
FROM SalesOrderItems s JOIN Products p
   ON ( s.ProductID = p.ID )
WHERE s.ShipDate BETWEEN '2001-07-01' AND '2001-08-31'
ORDER BY p.ID;
```

此查询会返回以下结果：

	ID	Description	Quantity	ShipDate	Cumulative_qty
1	301	V-neck	24	2001-07-16	24
2	302	Crew Neck	60	2001-07-02	60
3	302	Crew Neck	36	2001-07-13	96
4	400	Cotton Cap	48	2001-07-05	48
5	400	Cotton Cap	24	2001-07-19	72
6	401	Wool Cap	48	2001-07-09	48
7	500	Cloth Visor	12	2001-07-22	12
8	501	Plastic Visor	60	2001-07-07	60
9	501	Plastic Visor	12	2001-07-12	72
10	501	Plastic Visor	12	2001-07-22	84
11	601	Zipped Sweatshirt	60	2001-07-19	60
12	700	Cotton Shorts	24	2001-07-26	24

在此示例中，要在连接两个表和应用查询的 WHERE 子句之后，才执行 SUM 窗口函数的计算。查询会按如下方式进行：

1. 根据值 ProductID 分区（分组）输入行。
2. 在每个分区内，根据 ShipDate 的值对行进行排序。
3. 对于分区中的每一行，通过由各分区中（经过排序的）第一行直到当前行并包括当前行所组成的滑动窗口，使用 SUM 函数对 Quantity 中的值进行求值。

WINDOW 子句定义

上述查询的另一种结构是，使用 WINDOW 子句在使用窗口的函数中单独指定窗口，然后在各函数的 OVER 子句内引用窗口。

在此示例中，WINDOW 子句创建名为 Cumulative、按 ProductID 对数据分区并按 ShipDate 进行排序的窗口。SUM 函数在其 OVER 子句中引用窗口，并使用 ROWS 子句定义窗口大小。

```
SELECT p.ID, p.Description, s.Quantity, s.ShipDate,
    SUM( s.Quantity ) OVER ( Cumulative
    ROWS BETWEEN UNBOUNDED PRECEDING
    AND CURRENT ROW ) AS cumulative_qty
FROM SalesOrderItems s
JOIN Products p ON ( s.ProductID = p.ID )
WHERE s.ShipDate BETWEEN '2001-07-01' AND '2001-08-31'
WINDOW Cumulative AS ( PARTITION BY s.ProductID ORDER BY s.ShipDate )
ORDER BY p.ID;
```

当使用 WINDOW 子句语法时，以下限制将适用：

- 如果指定 PARTITION BY 子句，则必须将其置于 WINDOW 子句内。
- 如果指定 ROWS 或 RANGE 子句，则必须将其置于引用函数的 OVER 子句中。
- 如果为窗口指定 ORDER BY 子句，可将该子句置于 WINDOW 子句中，或者置于引用函数的 OVER 子句中，但不能同时置于二者之中。
- WINDOW 子句必须位于 SELECT 语句的 ORDER BY 子句之前。

内置与 WINDOW 子句定义相结合

可以内置一部分窗口定义，然后在 WINDOW 子句中定义剩余部分。例如：

```
AVG() OVER ( windowA
             ORDER BY expression )...
...
WINDOW windowA AS ( PARTITION BY expression )
```

使用此方式分隔窗口定义时有以下限制：

- 不能在窗口函数语法中使用 PARTITION BY 子句。
- 可以在窗口函数语法或 WINDOW 子句中使用 ORDER BY 子句，但不能在二者中同时使用。
- 不能在 WINDOW 子句中包括 RANGE 或 ROWS 子句。

另请参见

- “WINDOW 子句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “窗口集合函数”一节第 442 页
- “窗口秩函数”一节第 460 页
- “定义窗口”一节第 436 页

SQL Anywhere 中的窗口函数

允许对一组输入行进行分析操作的函数称为窗口函数。例如，所有秩函数以及大部分集合函数均为**窗口函数**。可以使用窗口函数对数据进行其它分析。具体实现方式为：处理输入行之前对输入行进行分区和排序，然后在遍历输入的、可配置大小的窗口中处理这些行。

有三种类型的窗口函数：窗口集合函数、窗口秩函数和行编号函数。

窗口集合函数

窗口集合函数为输入中一组指定的行返回值。例如，您可以使用窗口函数计算一家公司在一段指定时间内销售数据的移动平均值。

窗口集合函数分为以下三类：

- **基本集合函数** 以下是所支持的基本集合函数的列表：
  - SUM
  - AVG
  - MAX
  - MIN
  - FIRST_VALUE
  - LAST_VALUE
  - COUNT

  有关基本集合函数的详细信息，请参见“基本集合函数”一节第 443 页。
- **标准差和方差函数** 以下是所支持的标准差和方差函数的列表：
  - STDDEV
  - STDDEV_POP
  - STDDEV_SAMP
  - VAR_POP
  - VAR_SAMP
  - VARIANCE

  有关标准差和方差函数的详细信息，请参见“标准差和方差函数”一节第 454 页。

- **相关和线性回归函数** 以下是所支持的相关和线性回归函数的列表：
  - COVAR_POP
  - COVAR_SAMP
  - REGR_AVGX
  - REGR_AVGY
  - REGR_COUNT
  - REGR_INTERCEPT
  - REGR_R2
  - REGR_SLOPE
  - REGR_SXX
  - REGR_SXY
  - REGR_SYY

  有关相关和线性回归函数的详细信息，请参见“相关和线性回归函数”一节第 458 页。

基本集合函数

复杂的数据分析常常需要多级集合。窗口划分和排序，可以补充或者替代 GROUP BY 子句，为组成复杂的 SQL 查询提供了相当大的灵活性。例如，通过将简单的集合函数与窗口结构组合使用，可以计算诸如移动平均值、移动总和、最小或最大移动以及累计总和的值。

以下是 SQL Anywhere 中的基本集合函数：

- **SUM 函数** 返回每一组行的指定表达式总数。
- **AVG 函数** 为一组行返回一个数字表达式或一组唯一值的平均值。
- **MAX 函数** 返回在每一组行中找到的最大表达式值。
- **MIN 函数** 返回在每一组行中找到的最小表达式值。
- **FIRST_VALUE 函数** 从窗口第一行返回值。此函数需要窗口说明。
- **LAST_VALUE 函数** 从窗口最后一行返回值。此函数需要窗口说明。
- **COUNT 函数** 返回满足指定表达式的行的数量。

另请参见

- “窗口函数”一节第 436 页

SUM 函数示例

以下示例显示充当窗口函数的 SUM 函数。查询将返回按 DepartmentID 划分数据的结果集，然后提供员工薪水的累计总和 (Sum_Salary)（从在公司时间最长的员工开始）。结果集只包括居住在 California、Utah、New York 或 Arizona 的那些雇员。Sum_Salary 列提供雇员薪水的累计总额。

```
SELECT DepartmentID, Surname, StartDate, Salary,
SUM( Salary ) OVER ( PARTITION BY DepartmentID
    ORDER BY StartDate
```

```
    RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW )
AS "Sum_Salary"
FROM Employees
WHERE State IN ( 'CA', 'UT', 'NY', 'AZ' )
   AND DepartmentID IN ( '100', '200' )
ORDER BY DepartmentID, StartDate;
```

下表是查询的结果集。结果集按 DepartmentID 划分。

	DepartmentID	Surname	StartDate	Salary	Sum_Salary
1	100	Whitney	1984-08-28	45700.00	45700.00
2	100	Cobb	1985-01-01	62000.00	107700.00
3	100	Shishov	1986-06-07	72995.00	180695.00
4	100	Driscoll	1986-07-01	48023.69	228718.69
5	100	Guevara	1986-10-14	42998.00	271716.69
6	100	Wang	1988-09-29	68400.00	340116.69
7	100	Soo	1990-07-31	39075.00	379191.69
8	100	Diaz	1990-08-19	54900.00	434091.69
9	200	Overbey	1987-02-19	39300.00	39300.00
10	200	Martel	1989-10-16	55700.00	95000.00
11	200	Savarino	1989-11-07	72300.00	167300.00
12	200	Clark	1990-07-21	45000.00	212300.00
13	200	Goggin	1990-08-05	37900.00	250200.00

对于 DepartmentID 100 而言，来自 California、Utah、New York 和 Arizona 的雇员的薪水累计总额为 $434,091.69，而 DepartmentID 200 的雇员的薪水累计总额是 $250,200.00。

有关 SUM 函数的确切语法的详细信息，请参见“SUM 函数 [Aggregate]”一节 《SQL Anywhere 服务器 - SQL 参考》。

计算相邻行间的增量

使用两个窗口——一个窗口在当前行上，另一个在前一行上，可以计算相邻行之间的增量或更改。例如，以下查询将计算结果中一名雇员与前一名雇员之间的薪水增量 (Delta)：

```
SELECT EmployeeID AS EmployeeNumber,
   Surname AS LastName,
   SUM( Salary ) OVER ( ORDER BY BirthDate
      ROWS BETWEEN CURRENT ROW AND CURRENT ROW )
      AS CurrentRow,
   SUM( Salary ) OVER ( ORDER BY BirthDate
```



```
      ROWS BETWEEN 1 PRECEDING AND 1 PRECEDING )
      AS PreviousRow,
   ( CurrentRow - PreviousRow ) AS Delta
FROM Employees
WHERE State IN ( 'NY' );
```

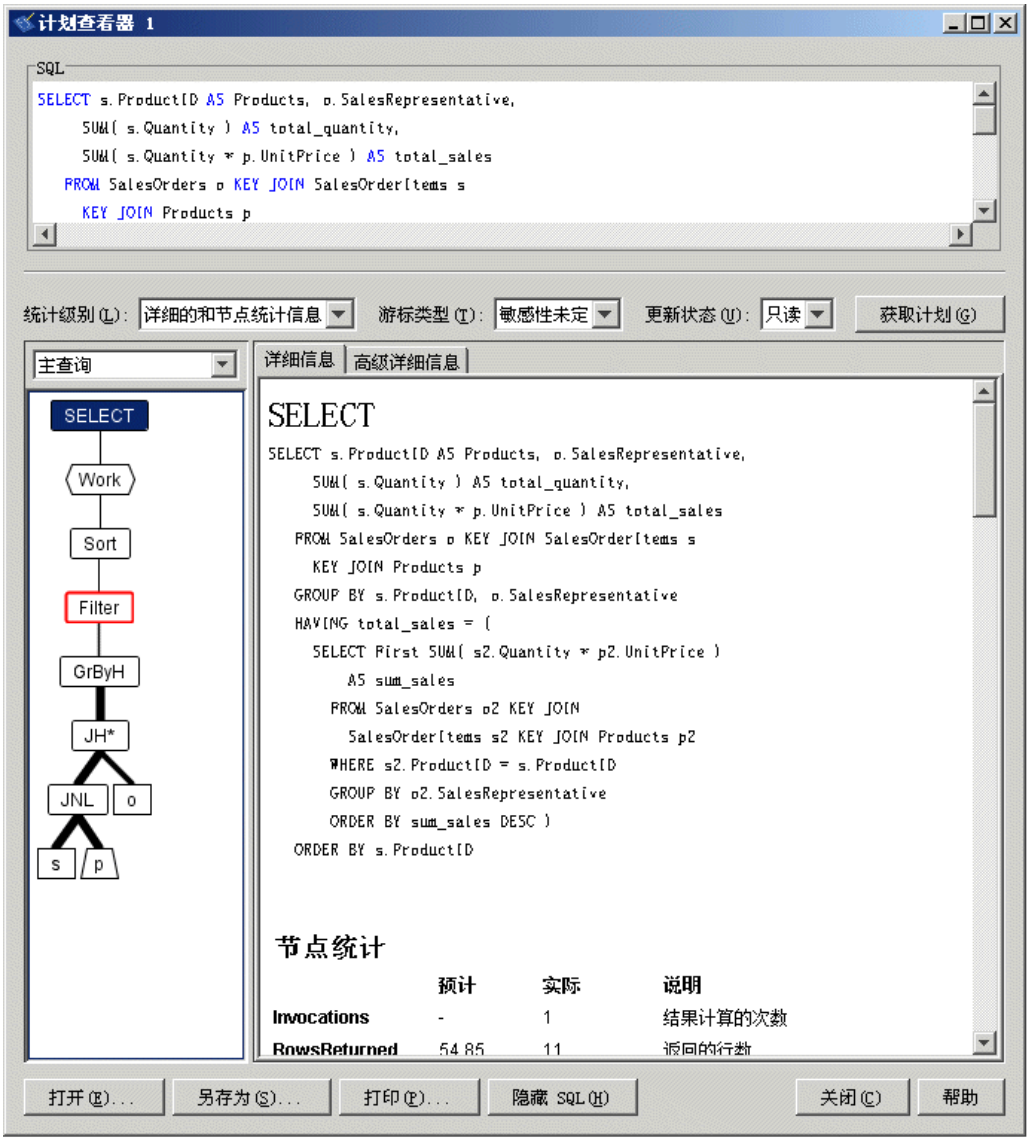
	EmployeeNumber	LastName	CurrentRow	PreviousRow	Delta
1	913	Martel	55700.000	(NULL)	(NULL)
2	1062	Blaikie	54900.000	55700.000	-800.000
3	249	Guevara	42998.000	54900.000	-11902.000
4	390	Davidson	57090.000	42998.000	14092.000
5	102	Whitney	45700.000	57090.000	-11390.000
6	1507	Wetherby	35745.000	45700.000	-9955.000
7	1751	Ahmed	34992.000	35745.000	-753.000
8	1157	Soo	39075.000	34992.000	4083.000

请注意，SUM 仅在 CurrentRow 窗口的当前行上执行，因为窗口大小已设置为 `ROWS BETWEEN CURRENT ROW AND CURRENT ROW`。同理，对于 PreviousRow 窗口，SUM 仅在前一行上执行，因为窗口大小已设置为 `ROWS BETWEEN 1 PRECEDING AND 1 PRECEDING`。在第一行中 PreviousRow 的值为 NULL，因为第一行没有前一行；因此 Delta 值也为 NULL。

复杂情况分析

请考虑以下查询，该查询列出数据库中各产品的最佳销售人员（由总销售额定义）：

```
SELECT s.ProductID AS Products, o.SalesRepresentative,
     SUM( s.Quantity ) AS total_quantity,
     SUM( s.Quantity * p.UnitPrice ) AS total_sales
  FROM SalesOrders o KEY JOIN SalesOrderItems s
    KEY JOIN Products p
  GROUP BY s.ProductID, o.SalesRepresentative
  HAVING total_sales = (
    SELECT First SUM( s2.Quantity * p2.UnitPrice )
        AS sum_sales
      FROM SalesOrders o2 KEY JOIN
        SalesOrderItems s2 KEY JOIN Products p2
      WHERE s2.ProductID = s.ProductID
      GROUP BY o2.SalesRepresentative
      ORDER BY sum_sales DESC )
  ORDER BY s.ProductID;
```



此查询会返回如下结果：

	Products	SalesRepresentative	total_quantity	total_sales
1	300	299	660	5940.00
2	301	299	516	7224.00
3	302	299	336	4704.00

	Products	SalesRepresentative	total_quantity	total_sales
4	400	299	458	4122.00
5	401	902	360	3600.00
6	500	949	360	2520.00
7	501	690	360	2520.00
8	501	949	360	2520.00
9	600	299	612	14688.00
10	601	299	636	15264.00
11	700	299	1008	15120.00

原始查询由可确定任何特定产品最高销售额的相关子查询构成，其中 ProductID 是子查询的相关外部引用。但是，使用嵌套查询通常是一种开销巨大的选择，此示例中即是如此。这是因为，子查询不仅涉及 GROUP BY 子句，还涉及 GROUP BY 子句内的 ORDER BY 子句。这样一来，查询优化程序就不能在保留相同语义的情况下，将此嵌套查询重新编写为连接。因此，在查询执行期间，将对在外部块中计算的每个派生行计算子查询。

注意图形式计划中开销巨大的 Filter 谓语：优化程序估计，执行查询 99% 的开销由此计划运算符带来的。子查询的计划清晰地阐释了主块中过滤器运算符开销如此巨大的原因：子查询涉及两个嵌套循环连接，即散列的 GROUP BY 操作和一个分类。

使用秩函数重新编写

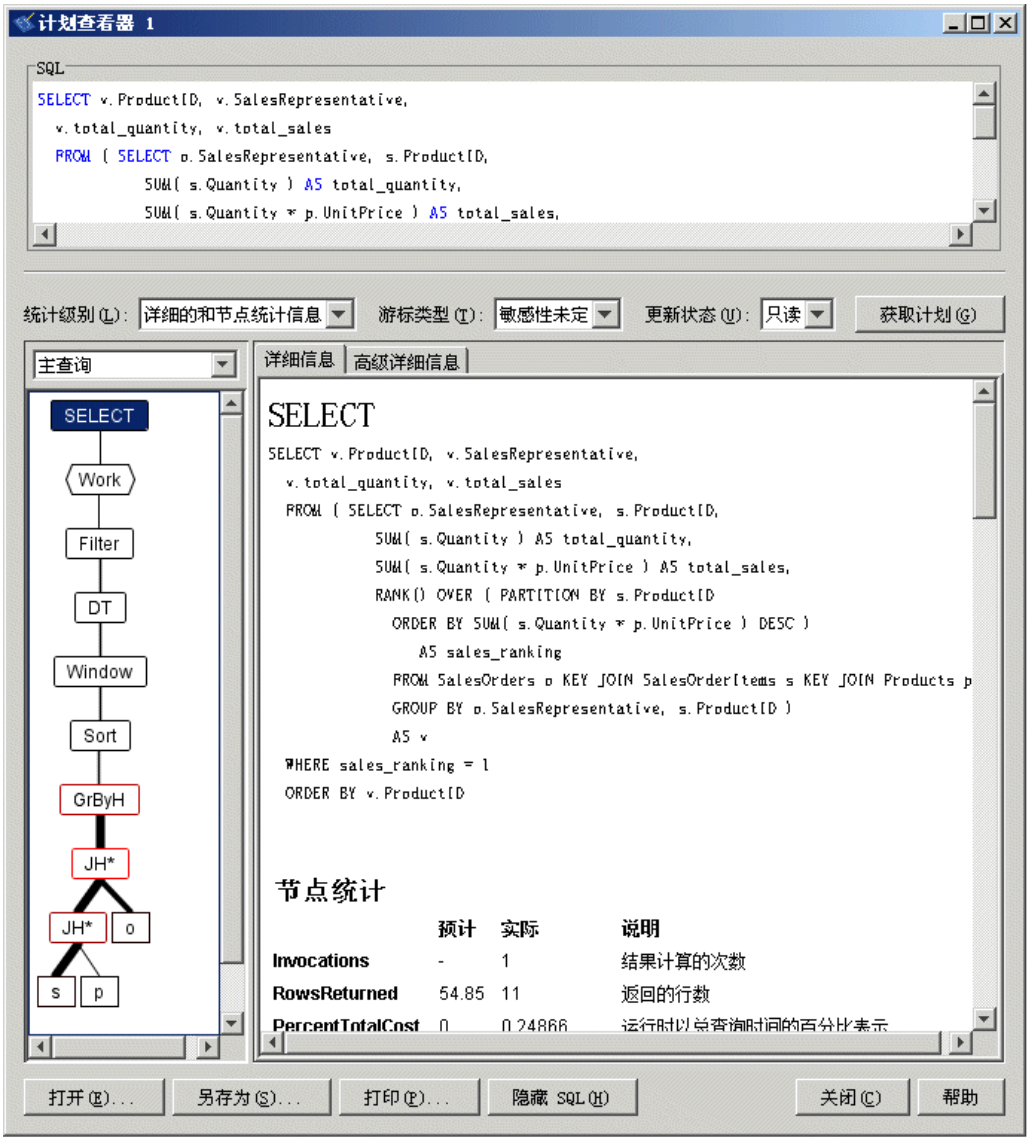
使用秩函数对同一查询进行重新编写，将更高效的计算出同一结果：

```
SELECT v.ProductID, v.SalesRepresentative,
  v.total_quantity, v.total_sales
  FROM ( SELECT o.SalesRepresentative, s.ProductID,
            SUM( s.Quantity ) AS total_quantity,
            SUM( s.Quantity * p.UnitPrice ) AS total_sales,
```



```
            RANK() OVER ( PARTITION BY s.ProductID
              ORDER BY SUM( s.Quantity * p.UnitPrice ) DESC )
              AS sales_ranking
            FROM SalesOrders o KEY JOIN SalesOrderItems s KEY JOIN Products p
            GROUP BY o.SalesRepresentative, s.ProductID )
            AS v
WHERE sales_ranking = 1
ORDER BY v.ProductID;
```

此重新编写的查询最终会生成一个更简单的计划：



请回忆一下，窗口运算符是在处理 GROUP BY 子句之后，计算选择列表项和查询的 ORDER BY 子句之前进行计算的。如同在图形式计划中所看到的，连接三个表以后，将按 SalesRepresentative 和

ProductID 属性的组合对所连接行进行分组。因此，可为每个 SalesRepresentative 和 ProductID 组合计算 total_quantity 和 total_sales 的 SUM 集合函数。

计算 GROUP BY 子句之后，就会使用窗口计算 RANK 函数，以按 total_sales 降序排列中间结果中的行。请注意，WINDOW 说明包括 PARTITION BY 子句。通过执行此操作，将重新划分（重新分组）GROUP BY 子句的结果—此次按 ProductID 划分。这样，RANK 函数以总销售额的降序排序各个产品的行，但是将针对已销售该产品的所有销售代表。通过此排序，确定最佳销售人员只需将派生表的结果限制为仅接受秩为 1 的那些行。对于出现并列的情况（如结果集中的第 7 行和第 8 行），RANK 将返回相同值。因此，销售人员 690 和 949 都会出现在最终结果中。

另请参见

- “SUM 函数 [Aggregate]”一节 《SQL Anywhere 服务器 - SQL 参考》

AVG 函数示例

在此示例中，AVG 用作窗口函数，来按月计算 2000 年所有产品销售额的移动平均值。请注意，WINDOW 说明使用 RANGE 子句，这会导致根据月值计算窗口边界，而不是像 ROWS 子句那样按相邻行的数目来计算边界。例如，如果部分或全部产品在某个特定的月完全没有销售额，则使用 ROWS 将产生不同的结果。

```
SELECT *
  FROM ( SELECT s.ProductID,
         Month( o.OrderDate ) AS julian_month,
         SUM( s.Quantity ) AS sales,
         AVG( SUM( s.Quantity ) )
         OVER ( PARTITION BY s.ProductID
           ORDER BY Month( o.OrderDate ) ASC
           RANGE BETWEEN 1 PRECEDING AND 1 FOLLOWING )
         AS average_sales
         FROM SalesOrderItems s KEY JOIN SalesOrders o
         WHERE Year( o.OrderDate ) = 2000
         GROUP BY s.ProductID, Month( o.OrderDate ) )
  AS DT
  ORDER BY 1,2;
```

另请参见

- “AVG 函数 [Aggregate]”一节 《SQL Anywhere 服务器 - SQL 参考》

MAX 函数示例

消除相关子查询

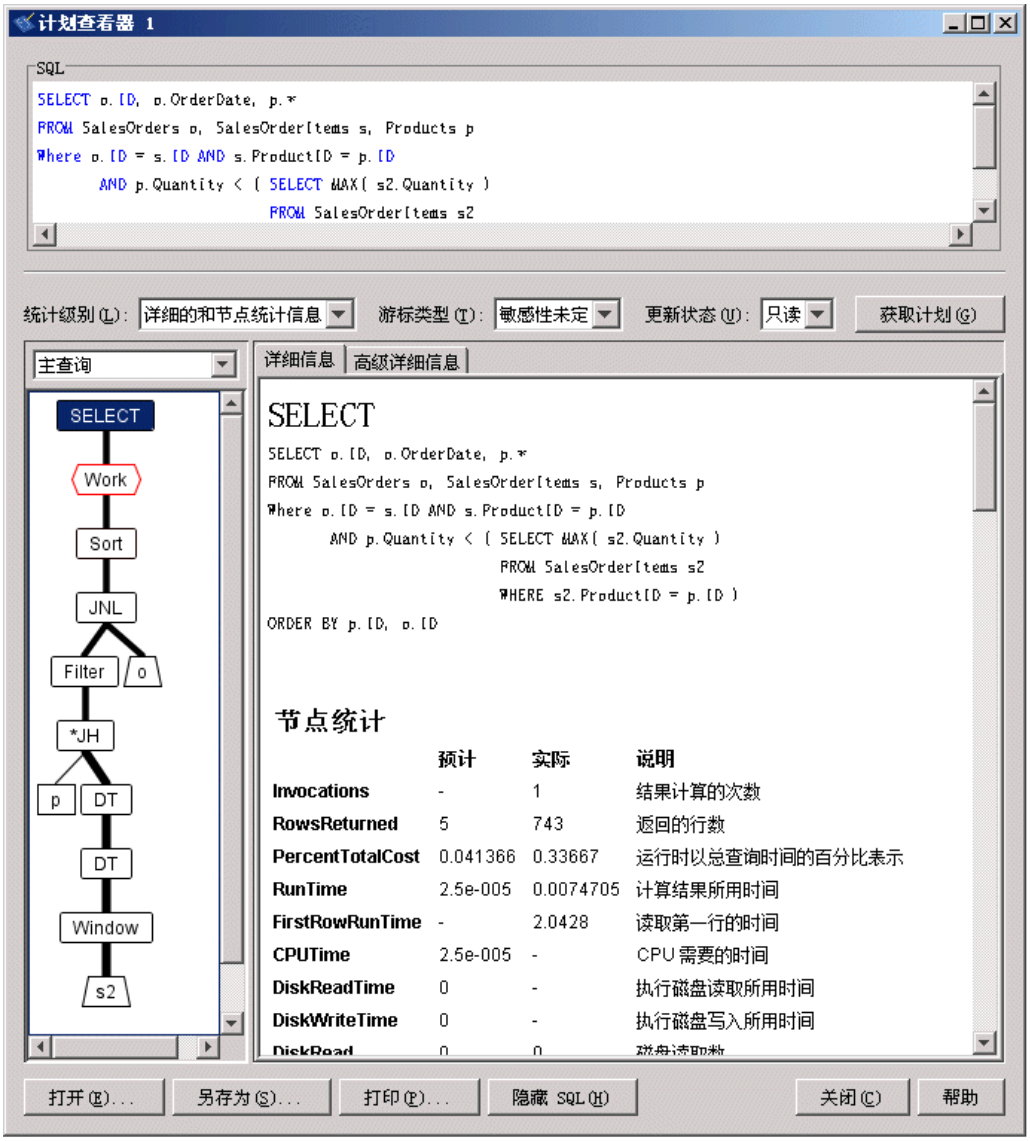
在某些情况下，可能需要具备将特定列值与最大或最小值进行比较的能力。通常会以涉及相关属性的嵌套查询（也称为外部引用）的形式来构成这些查询。例如，请考虑以下查询，该查询列出包括产品信息的所有订单，其中现有产品数量不能包括该产品最大的单个订单量：

```
SELECT o.ID, o.OrderDate, p.*
FROM SalesOrders o, SalesOrderItems s, Products p
WHERE o.ID = s.ID AND s.ProductID = p.ID
       AND p.Quantity < ( SELECT MAX( s2.Quantity )
                          FROM SalesOrderItems s2
```



```
                                WHERE s2.ProductID = p.ID )
ORDER BY p.ID, o.ID;
```

此查询的图形式计划显示在 [计划查看器] 中，如下所示。注意查询优化程序如何将此嵌套查询转换成带有派生表的 Products 和 SalesOrders 表的连接，该派生表由相关名 DT 表示，它包含窗口函数。



最好不依赖于优化程序将相关子查询转换成带有派生表的连接—这种方法由于语义分析的复杂性而仅能在简单情况下实现—您可以使用窗口函数来构成此类查询：

```
SELECT order_qty.ID, o.OrderDate, p.*
  FROM ( SELECT s.ID, s.ProductID,
```

```
            MAX( s.Quantity ) OVER (
              PARTITION BY s.ProductID
              ORDER BY s.ProductID )
            AS max_q
            FROM SalesOrderItems s )
    AS order_qty, Products p, SalesOrders o
    WHERE p.ID = ProductID
      AND o.ID = order_qty.ID
      AND p.Quantity < max_q
    ORDER BY p.ID, o.ID;
```

另请参见

- “MIN 函数 [Aggregate]”一节 《SQL Anywhere 服务器 - SQL 参考》
- “MAX 函数 [Aggregate]”一节 《SQL Anywhere 服务器 - SQL 参考》

FIRST_VALUE 和 LAST_VALUE 函数示例

FIRST_VALUE 和 LAST_VALUE 函数从窗口的第一行和最后一行返回值。这允许查询在无需自连接的情况下一次访问多个行的值。

这两个函数不同于其它窗口集合函数，因为它们必须配合窗口一起使用。同样，不像其它窗口集合函数，这些函数允许使用 IGNORE NULLS 子句。如果指定 IGNORE NULLS，将返回所求表达式的第一个或最后一个非 NULL 值。否则，将返回第一个值或最后一个值。

示例 1：组中第一个条目

FIRST_VALUE 函数可用于在一组有序值中检索第一个条目。对于每张订单，以下查询返回订单中第一项的产品标识符；也就是说，返回每张订单中具有最小 LineID 的项目的 ProductID。

请注意，查询使用 DISTINCT 关键字删除重复项，若不使用 DISTINCT 关键字，每张订单中的每一项都将返回重复行。

```
SELECT DISTINCT ID,
FIRST_VALUE( ProductID ) OVER ( PARTITION BY ID ORDER BY LineID )
FROM SalesOrderItems
ORDER BY ID;
```

示例 2：最高销售额百分比

FIRST_VALUE 函数常见的用途是将每行中的一个值与当前组内的最大或最小值进行比较。以下查询计算每个销售代表的总销售额，然后比较同一产品中该销售代表的销售额和总销售额。结果以总销售额的百分比表示。

```
SELECT s.ProductID AS prod_id, o.SalesRepresentative AS sales_rep,
    SUM( s.Quantity * p.UnitPrice ) AS total_sales,
    100 * total_sales / ( FIRST_VALUE( SUM( s.Quantity * p.UnitPrice ) )
                          OVER Sales_Window ) AS total_sales_percentage
  FROM SalesOrders o KEY JOIN SalesOrderItems s KEY JOIN Products p
  GROUP BY o.SalesRepresentative, s.ProductID
    WINDOW Sales_Window AS ( PARTITION BY s.ProductID
                             ORDER BY SUM( s.Quantity * p.UnitPrice ) DESC )
  ORDER BY s.ProductID;
```

示例 3：填充 NULL 值，使数据更紧凑

FIRST_VALUE 和 LAST_VALUE 函数在使数据更紧凑和需要填充值（代替 NULL）时很有用。例如，假设每天完成最高销售额的销售代表获得日销售冠军的称号。下面的查询列出了 2001 年四月的第一个星期的获胜销售代表：

```
SELECT v.OrderDate, v.SalesRepresentative AS rep_of_the_day
FROM ( SELECT o.SalesRepresentative, o.OrderDate,
              RANK() OVER ( PARTITION BY o.OrderDate
                            ORDER BY SUM( s.Quantity *
                              p.UnitPrice ) DESC ) AS sales_ranking
       FROM SalesOrders o KEY JOIN SalesOrderItems s KEY JOIN Products p
       GROUP BY o.SalesRepresentative, o.OrderDate ) AS v
WHERE v.sales_ranking = 1
AND v.OrderDate BETWEEN '2001-04-01' AND '2001-04-07'
ORDER BY v.OrderDate;
```

此查询会返回以下结果：

OrderDate	rep_of_the_day
2001-04-01	949
2001-04-02	856
2001-04-05	902
2001-04-06	467
2001-04-07	299

但是，请注意，没有销售额的日子不返回结果。以下查询使数据更紧凑，为没有销售额的日子创建记录。此外，它使用 LAST_VALUE 函数用上次获胜代表的 ID 为 rep_of_the_day（在非获胜日）填充 NULL 值，直到新的获胜者在结果中出现。

```
SELECT d.dense_order_date,
                   LAST_VALUE( v.SalesRepresentative IGNORE NULLS )
                       OVER ( ORDER BY d.dense_order_date )
                       AS rep_of_the_day
FROM ( SELECT o.SalesRepresentative, o.OrderDate,
              RANK() OVER ( PARTITION BY o.OrderDate
                            ORDER BY SUM( s.Quantity *
                             p.UnitPrice ) DESC ) AS sales_ranking
       FROM SalesOrders o KEY JOIN SalesOrderItems s KEY JOIN Products p
       GROUP BY o.SalesRepresentative, o.OrderDate ) AS v
RIGHT OUTER JOIN ( SELECT DATEADD( day, row_num, '2001-04-01' )
                        AS dense_order_date
                   FROM sa_rowgenerator( 0, 6 )) AS d
ON v.OrderDate = d.dense_order_date AND sales_ranking = 1
ORDER BY d.dense_order_date;
```

此查询会返回以下结果：

OrderDate	rep_of_the_day
2001-04-01	949
2001-04-02	856
2001-04-03	856
2001-04-04	856
2001-04-05	902
2001-04-06	467
2001-04-07	299

上一个查询的派生表 v 连接到派生表 d，表 d 包含需要考虑的所有日期。这为每个目标日生成一行，但是此外连接在没有销售额的日期的 SalesRepresentative 列中包含 NULL。使用 LAST_VALUE 函数通过将给定行的 rep_of_the_day 定义为到相应日子一直胜出的 SalesRepresentative 的最后非 NULL 值来解决此问题。

另请参见

- “FIRST_VALUE 函数 [Aggregate]”一节 《SQL Anywhere 服务器 - SQL 参考》
- “LAST_VALUE 函数 [Aggregate]”一节 《SQL Anywhere 服务器 - SQL 参考》
- “窗口函数”一节第 436 页

标准差和方差函数

SQL Anywhere 支持两个版本的方差和标准差函数：一个样本版本，一个总体版本。根据使用函数的统计上下文来在两种版本间进行选择。

所有方差和标准差函数都是真集合函数，因为它们可以按照查询的 GROUP BY 子句所确定的方式来计算行分区的值。与其它基本集合函数（例如 MAX 或 MIN）一样，它们执行的计算也会忽略输入中的 NULL 值。

为提高性能，SQL Anywhere 在一步中计算平均值和离均差。这意味着只需进行一次数据传递。

此外，无论被分析表达式的域如何，都将使用 IEEE 双精度浮点型完成所有方差和标准差计算。如果任何方差或标准差函数的输入是空集，则每个函数将返回 NULL 作为其结果。如果对单个行计算 VAR_SAMP，则它将返回 NULL，而计算 VAR_POP 将返回值 0。

以下是 SQL Anywhere 中所提供的标准差和方差函数：

- STDDEV 函数
- STDDEV_POP 函数
- STDDEV_SAMP 函数
- VARIANCE 函数
- VAR_POP 函数
- VAR_SAMP 函数

要查看这些函数所表示的数学公式，请参见“集合函数的数学公式”一节第 468 页。

STDDEV 函数

此函数是 STDDEV_SAMP 函数的别名。请参见“STDDEV_SAMP 函数 [Aggregate]”一节 《SQL Anywhere 服务器 - SQL 参考》。

STDDEV_POP 函数

此函数计算由数字表达式组成的总体的标准差，类型为 DOUBLE。

示例 1

下面的查询返回一个结果集，该结果集显示了其薪水比其部门平均薪水高一个标准差的雇员。标准差是反映数据与平均值之间的差异的测量单位。

```
SELECT *
FROM ( SELECT
    Surname AS Employee,
    DepartmentID AS Department,
    CAST( Salary as DECIMAL( 10, 2 ) )
        AS Salary,
    CAST( AVG( Salary )
        OVER ( PARTITION BY DepartmentID ) AS DECIMAL ( 10, 2 ) )
        AS Average,
    CAST( STDDEV_POP( Salary )
        OVER ( PARTITION BY DepartmentID ) AS DECIMAL ( 10, 2 ) )
        AS StandardDeviation
    FROM Employees
    GROUP BY Department, Employee, Salary )
    AS DerivedTable
WHERE Salary > Average + StandardDeviation
ORDER BY Department, Salary, Employee;
```

下表是查询的结果集。每个部门至少有一名雇员的薪水远远偏离于平均值。

	Employee	Department	Salary	Average	StandardDeviation
1	Lull	100	87900.00	58736.28	16829.60
2	Scheffield	100	87900.00	58736.28	16829.60
3	Scott	100	96300.00	58736.28	16829.60
4	Sterling	200	64900.00	48390.95	13869.60

	Employee	Department	Salary	Average	StandardDeviation
5	Savarino	200	72300.00	48390.95	13869.60
6	Kelly	200	87500.00	48390.95	13869.60
7	Shea	300	138948.00	59500.00	30752.40
8	Blaikie	400	54900.00	43640.67	11194.02
9	Morris	400	61300.00	43640.67	11194.02
10	Evans	400	68940.00	43640.67	11194.02
11	Martinez	500	55500.00	33752.20	9084.50

雇员 Scott 的收入为 $96,300.00，而部门平均收入为 $58,736.28。该部门的标准差是 $16,829.00，这表示低于 $75,565.88 (`58736.28 + 16829.60 = 75565.88`) 的薪水会在平均值的一个标准差范围内。雇员 Scott 的薪水 $96,300.00 远远高于该数字。

此示例假设每个雇员的 Surname 和 Salary 唯一，而实际情况不一定如此。为确保唯一性，应向 GROUP BY 子句添加 EmployeeID。

示例 2

以下语句列出不同时间段每个订单的产品数目的平均值和方差：

```
SELECT YEAR( ShipDate ) AS Year,
    QUARTER( ShipDate ) AS Quarter,
    AVG( Quantity ) AS Average,
    STDDEV_POP( Quantity ) AS Variance
FROM SalesOrderItems
GROUP BY Year, Quarter
ORDER BY Year, Quarter;
```

此查询会返回以下结果：

Year	Quarter	Average	Variance
2000	1	25.775148	14.2794...
2000	2	27.050847	15.0270...
...	...	...	...

有关此函数语法的详细信息，请参见“STDDEV_SAMP 函数 [Aggregate]”一节 《SQL Anywhere 服务器 - SQL 参考》。

STDDEV_SAMP 函数

此函数计算由数字表达式组成的样本的标准差，类型为 DOUBLE。例如，以下语句返回不同季度每个订单的产品数目的平均值和方差：


```
SELECT YEAR( ShipDate ) AS Year,
    QUARTER( ShipDate ) AS Quarter,
    AVG( Quantity ) AS Average,
    STDDEV_SAMP( Quantity ) AS Variance
FROM SalesOrderItems
GROUP BY Year, Quarter
ORDER BY Year, Quarter;
```

此查询会返回以下结果：

Year	Quarter	Average	Variance
2000	1	25.775148	14.3218...
2000	2	27.050847	15.0696...
...	...	...	...

有关此函数语法的详细信息，请参见“STDDEV_POP 函数 [Aggregate]”一节 《SQL Anywhere 服务器 - SQL 参考》。

VARIANCE 函数

此函数是 VAR_SAMP 函数的别名。请参见“VAR_SAMP 函数 [Aggregate]”一节 《SQL Anywhere 服务器 - SQL 参考》。

VAR_POP 函数

此函数计算由数字表达式组成的总体的统计方差，类型为 DOUBLE。例如，以下语句列出不同时间段每个订单的产品数目的平均值和方差：

```
SELECT YEAR( ShipDate ) AS Year,
    QUARTER( ShipDate ) AS Quarter,
    AVG( Quantity ) AS Average,
    VAR_POP( quantity ) AS Variance
FROM SalesOrderItems
GROUP BY Year, Quarter
ORDER BY Year, Quarter;
```

此查询会返回以下结果：

Year	Quarter	Average	Variance
2000	1	25.775148	203.9021...
2000	2	27.050847	225.8109...
...	...	...	...

如果对单个行计算 VAR_POP，则它将返回值 0。

有关此函数语法的详细信息，请参见“VAR_POP 函数 [Aggregate]”一节 《SQL Anywhere 服务器 - SQL 参考》。

VAR_SAMP 函数

此函数计算由数字表达式组成的样本的统计方差，类型为 DOUBLE。

例如，以下语句列出不同时间段每个订单的产品数目的平均值和方差：

```
SELECT YEAR( ShipDate ) AS Year,
    QUARTER( ShipDate ) AS Quarter,
    AVG( Quantity ) AS Average,
    VAR_SAMP( Quantity ) AS Variance
FROM SalesOrderItems
GROUP BY Year, Quarter
ORDER BY Year, Quarter;
```

此查询会返回以下结果：

Year	Quarter	Average	Variance
2000	1	25.775148	205.1158...
2000	2	27.050847	227.0939...
...	...	...	...

如果对单个行计算 VAR_SAMP，则它将返回 NULL。

有关此函数语法的详细信息，请参见“VAR_SAMP 函数 [Aggregate]”一节 《SQL Anywhere 服务器 - SQL 参考》。

相关和线性回归函数

SQL Anywhere 支持多种统计函数，这些函数的结果可用于协助分析线性回归的质量。

有关这些函数所表示的数学公式的详细信息，请参见“集合函数的数学公式”一节第 468 页。

每个函数的第一个参数是相关表达式（由 Y 表示），第二个参数是独立表达式（由 X 表示）。

- **COVAR_SAMP 函数** COVAR_SAMP 函数返回一组 (Y, X) 对的样本协方差。

  有关此函数语法的详细信息，请参见“COVAR_SAMP 函数 [Aggregate]”一节 《SQL Anywhere 服务器 - SQL 参考》。

- **COVAR_POP 函数** COVAR_POP 函数返回一组 (Y, X) 对的总体协方差。

  有关此函数语法的详细信息，请参见“COVAR_POP 函数 [Aggregate]”一节 《SQL Anywhere 服务器 - SQL 参考》。

- **CORR 函数** CORR 函数返回一组 (Y, X) 对的相关系数。

  有关此函数语法的详细信息，请参见“CORR 函数 [Aggregate]”一节 《SQL Anywhere 服务器 - SQL 参考》。

- **REGR_AVGX 函数** REGR_AVGX 函数从所有非 NULL (Y, X) 值对返回 x 值的平均值。

有关此函数语法的详细信息，请参见“REGR_AVGX 函数 [Aggregate]”一节 《SQL Anywhere 服务器 - SQL 参考》。

- **REGR_AVGY 函数** REGR_AVGY 函数从所有非 NULL (Y, X) 值对返回 y 值的平均值。

  有关此函数语法的详细信息，请参见“REGR_AVGY 函数 [Aggregate]”一节 《SQL Anywhere 服务器 - SQL 参考》。

- **REGR_SLOPE 函数** REGR_SLOPE 函数计算拟合到非 NULL 数对的线性回归线的斜率。

  有关此函数语法的详细信息，请参见“REGR_SLOPE 函数 [Aggregate]”一节 《SQL Anywhere 服务器 - SQL 参考》。

- **REGR_INTERCEPT 函数** REGR_INTERCEPT 函数计算可最佳拟合非独立和独立变量的线性回归线的 y 截距。

  有关此函数语法的详细信息，请参见“REGR_INTERCEPT 函数 [Aggregate]”一节 《SQL Anywhere 服务器 - SQL 参考》。

- **REGR_R2 函数** REGR_R2 函数计算回归线的确定系数（也称为 **R 平方**或**适配度**统计）。

  有关此函数语法的详细信息，请参见“REGR_R2 函数 [Aggregate]”一节 《SQL Anywhere 服务器 - SQL 参考》。

- **REGR_COUNT 函数** REGR_COUNT 函数返回输入中非 NULL (Y, X) 值对的数量。仅当给定对中的 X 和 Y 都为非 NULL 时，才应在所有线性回归计算中使用此观测。

  有关此函数语法的详细信息，请参见“REGR_COUNT 函数 [Aggregate]”一节 《SQL Anywhere 服务器 - SQL 参考》。

- **REGR_SXX 函数** 该函数返回 (Y, X) 对中 x 值的平方和。

  此函数等式等效于样本或总体方差公式中的分子。请注意，与其它线性回归函数一样，REGR_SXX 将忽略输入中 X 为 NULL 或 Y 为 NULL 的任何 (Y, X) 值对。

  有关此函数语法的详细信息，请参见“REGR_SXX 函数 [Aggregate]”一节 《SQL Anywhere 服务器 - SQL 参考》。

- **REGR_SYY 函数** 该函数返回 (Y, X) 对中 y 值的平方和。

  有关此函数语法的详细信息，请参见“REGR_SYY 函数 [Aggregate]”一节 《SQL Anywhere 服务器 - SQL 参考》。

- **REGR_SXY 函数** 该函数返回 (Y, X) 对集中两项积和的差值。

  有关此函数语法的详细信息，请参见“REGR_SXY 函数 [Aggregate]”一节 《SQL Anywhere 服务器 - SQL 参考》。

窗口秩函数

窗口秩函数返回分区中某个行相对于其它行的秩。SQL Anywhere 所支持的秩函数有：

- CUME_DIST
- DENSE_RANK
- PERCENT_RANK
- RANK

秩函数不被视为集合函数，因为它们并不采用与集合函数（例如 SUM 集合函数）相同的方式来计算多个输入行的结果。而是由上述每个函数根据特定表达式的值，计算分区内行的秩或相对排序。分区内的每组行均独立排序；如果 OVER 子句不包含 PARTITION BY 子句，则将整个输入视为单个分区。因此，不能为秩函数所使用的窗口指定 ROWS 或 RANGE 子句。可以构成包含多个秩函数的查询，其中的每个秩函数以不同方式划分或排序输入行。

所有秩函数都需要有一个 ORDER BY 子句，以指定秩函数所依赖的输入行的排序顺序。当 ORDER BY 子句包括多个表达式时，若第一个表达式与相邻行具有相同值，则第二个及后续表达式将用于区分并列情况。在 SQL Anywhere 中，始终将 NULL 值排在任何其它值的前面（在升序序列中）。

RANK 函数

当与其它行中的值进行比较时，使用 RANK 函数返回当前行中值的秩。值的秩反映值在已排序的值列表中出现的位置。

当使用 RANK 函数时，会计算在窗口的 ORDER BY 子句中所指定的表达式的秩。当 ORDER BY 子句包括多个表达式时，若第一个表达式与相邻行具有相同值，则第二个及后续表达式将用于区分并列情况。NULL 值排在任何其它值的前面（在升序序列中）。

示例 1

以下查询确定数据库中三种最贵的产品。由于为窗口指定了降序排序，因此最贵产品的秩最低，也就是秩从 1 开始。

```
SELECT Top 3 *
       FROM ( SELECT Description, Quantity, UnitPrice,
              RANK() OVER ( ORDER BY UnitPrice DESC ) AS Rank
              FROM Products ) AS DT
ORDER BY Rank;
```

此查询会返回以下结果：

	Description	Quantity	UnitPrice	Rank
1	Zipped Sweatshirt	32	24.00	1
2	Hooded Sweatshirt	39	24.00	1
3	Cotton Shorts	80	15.00	3

请注意，第 1 行和第 2 行中的 Unit Price 值相同，因此具有相同的秩。这称为并列。

使用 RANK 函数时，在出现并列之后秩值将发生跳跃。例如，第 3 行的秩已跳跃到 3 而不是 2。这一点与 DENSE_RANK 函数不同，DENSE_RANK 函数在出现并列之后不发生跳跃。请参见“DENSE_RANK 函数”一节第 462 页。

示例 2

以下 SQL 查询查找 Utah 的男女雇员并按薪水以降序顺序排列他们。

```
SELECT Surname, Salary, Sex,
     RANK() OVER ( ORDER BY Salary DESC ) "Rank"
     FROM Employees
WHERE State IN ( 'UT' );
```

下表是查询的结果集：

	Surname	Salary	Sex	Rank
1	Shishov	72995.00	F	1
2	Wang	68400.00	M	2
3	Cobb	62000.00	M	3
4	Morris	61300.00	M	4
5	Diaz	54900.00	M	5
6	Driscoll	48023.69	M	6
7	Hildebrand	45829.00	F	7
8	Goggin	37900.00	M	8
9	Rebeiro	34576.00	M	9
10	Bigelow	31200.00	F	10
11	Lynch	24903.00	M	11

示例 3

您可以划分数据以提供不同的结果。使用示例 2 中的查询，您可以按性别划分数据，从而更改数据。以下示例按薪水以降序对雇员进行排位并按性别划分雇员。

```
SELECT Surname, Salary, Sex,
     RANK ( ) OVER ( PARTITION BY Sex
     ORDER BY Salary DESC ) "Rank"
     FROM Employees
WHERE State IN ( 'UT' );
```

下表是查询的结果集：

	Surname	Salary	Sex	Rank
1	Wang	68400.00	M	1
2	Cobb	62000.00	M	2
3	Morris	61300.00	M	3
4	Diaz	54900.00	M	4
5	Driscoll	48023.69	M	5
6	Goggin	37900.00	M	6
7	Rebeiro	34576.00	M	7
8	Lynch	24903.00	M	8
9	Shishov	72995.00	F	1
10	Hildebrand	45829.00	F	2
11	Bigelow	31200.00	F	3

有关 RANK 函数语法的详细信息，请参见“RANK 函数 [Ranking]”一节 《SQL Anywhere 服务器 - SQL 参考》。

DENSE_RANK 函数

与 RANK 函数类似，当与其它行中的值进行比较时，使用 DENSE_RANK 函数返回当前行中值的秩。值的秩反映值在已排序的值列表中出现的位置。将为在窗口的 ORDER BY 子句中所指定的表达式计算秩。

DENSE_RANK 函数将返回一系列的秩，这些秩值将单调递增而不会有空位或跳跃。因为在秩值中没有跳跃（与 RANK 函数不同），所以使用了术语紧凑 (dense)。

随着窗口在输入行中向下移动，会计算在窗口的 ORDER BY 子句中所指定的表达式的秩。当 ORDER BY 子句包括多个表达式时，若第一个表达式与相邻行具有相同值，则第二个及后续表达式将用于区分并列情况。NULL 值排在任何其它值的前面（在升序序列中）。

示例 1

以下查询确定数据库中三种最贵的产品。由于在窗口中指定了降序排序，因此最贵产品的秩最低（秩从 1 开始）。

```
SELECT Top 3 *
  FROM ( SELECT Description, Quantity, UnitPrice,
    DENSE_RANK( ) OVER ( ORDER BY UnitPrice DESC ) AS Rank
    FROM Products ) AS DT
  ORDER BY Rank;
```


此查询会返回以下结果：

	Description	Quantity	UnitPrice	Rank
1	Hooded Sweatshirt	39	24.00	1
2	Zipped Sweatshirt	32	24.00	1
3	Cotton Shorts	80	15.00	2

请注意，第 1 行和第 2 行中的 Unit Price 值相同，因此具有相同的秩。这称为并列。

使用 DENSE_RANK 函数时，在出现并列之后不发生跳跃。例如，第 3 行的秩值是 2。这一点与 RANK 函数不同，RANK 函数在出现并列之后秩值会发生跳跃。请参见“RANK 函数”一节 第 460 页。

示例 2

因为执行查询的 GROUP BY 子句之后会对窗口进行求值，所以您可以指定根据集合函数的值确定秩的复杂请求。

以下查询根据销售人员在其地区的总销售额，查询出各地区的前三名销售人员以及各地区的总销售额：

```
SELECT *
  FROM ( SELECT o.SalesRepresentative, o.Region,
              SUM( s.Quantity * p.UnitPrice ) AS total_sales,
              DENSE_RANK( ) OVER ( PARTITION BY o.Region,
                GROUPING( o.SalesRepresentative )
                ORDER BY total_sales DESC ) AS sales_rank
            FROM Products p, SalesOrderItems s, SalesOrders o
            WHERE p.ID = s.ProductID AND s.ID = o.ID
            GROUP BY GROUPING SETS( ( o.SalesRepresentative, o.Region ),
               o.Region ) ) AS DT
  WHERE sales_rank <= 3
  ORDER BY Region, sales_rank;
```

此查询会返回以下结果：

	SalesRepresentative	Region	total_sales	sales_rank
1	299	Canada	9312.00	1
2	(NULL)	Canada	24768.00	1
3	1596	Canada	3564.00	2
4	856	Canada	2724.00	3
5	299	Central	32592.00	1
6	(NULL)	Central	134568.00	1

	SalesRepresentative	Region	total_sales	sales_rank
7	856	Central	14652.00	2
8	467	Central	14352.00	3
9	299	Eastern	21678.00	1
10	(NULL)	Eastern	142038.00	1
11	902	Eastern	15096.00	2
12	690	Eastern	14808.00	3
13	1142	South	6912.00	1
14	(NULL)	South	45262.00	1
15	667	South	6480.00	2
16	949	South	5782.00	3
17	299	Western	5640.00	1
18	(NULL)	Western	37632.00	1
19	1596	Western	5076.00	2
20	667	Western	4068.00	3

此查询通过使用 GROUPING SETS 将多个分组组合到了一起。因此，窗口的 WINDOW PARTITION 子句使用 GROUPING 函数来区分详细信息行和小计行，其中详细信息行表示特定销售人员，而小计行列出整个地区的总销售额。按地区的小计行中，销售代表属性的值为 NULL，而每行的秩值均为 1，这是由于结果的秩排序在输入的每个分区中都重新开始；这样就确保了详细信息行都能从 1 开始正确排序。

最后要注意的是，在此示例中 DENSE_RANK 函数对输入的排序优先于总销售额集合。带别名的选择列表项在 WINDOW ORDER 子句中用作速记。

有关 DENSE_RANK 函数语法的详细信息，请参见“DENSE_RANK 函数 [Ranking]”一节 《SQL Anywhere 服务器 - SQL 参考》。

CUME_DIST 函数

累计分布函数 CUME_DIST，有时被定义为反百分点函数。CUME_DIST 计算特定值相对于窗口中一组值的规范化位置。该函数的范围在 0 到 1 之间。

随着窗口在输入行中向下移动，会计算在窗口的 ORDER BY 子句中所指定表达式的累计分布。当 ORDER BY 子句包括多个表达式时，若第一个表达式与相邻行具有相同值，则第二个及后续表达式将用于区分并列情况。NULL 值排在任何其它值的前面（在升序序列中）。

以下示例返回一个包含居住在 California 的职员的薪水累计分布的结果集。

```
SELECT DepartmentID, Surname, Salary,
    CUME_DIST( ) OVER ( PARTITION BY DepartmentID
      ORDER BY Salary DESC ) "Rank"
  FROM Employees
  WHERE State IN ( 'CA' );
```

此查询会返回以下结果：

DepartmentID	Surname	Salary	Rank
200	Savarino	72300.00	0.333333333333333
200	Clark	45000.00	0.666666666666667
200	Overbey	39300.00	1

有关 CUME_DIST 函数语法的详细信息，请参见“CUME_DIST 函数 [Ranking]”一节 《SQL Anywhere 服务器 - SQL 参考》。

PERCENT_RANK 函数

Similar to the PERCENT function, the PERCENT_RANK function returns the rank for the value in the column specified in the window's ORDER BY clause, but expressed as a fraction between 0 an 1, calculated as (RANK - 1)/(- 1).

随着窗口在输入行中向下移动，会计算在窗口的 ORDER BY 子句中所指定的表达式的秩。当 ORDER BY 子句包括多个表达式时，若第一个表达式与相邻行具有相同值，则第二个及后续表达式将用于区分并列情况。NULL 值排在任何其它值的前面（在升序序列中）。

示例 1

以下示例返回按性别显示 New York 雇员薪水排位的结果集。该结果使用百分比数按降序列出排位并按性别分区。

```
SELECT DepartmentID, Surname, Salary, Sex,
    PERCENT_RANK( ) OVER ( PARTITION BY Sex
      ORDER BY Salary DESC ) AS PctRank
  FROM Employees
  WHERE State IN ( 'NY' );
```

此查询会返回以下结果：

	DepartmentID	Surname	Salary	Sex	PctRank
1	200	Martel	55700.000	M	0.0

	DepartmentID	Surname	Salary	Sex	PctRank
2	100	Guevara	42998.000	M	0.333333333
3	100	Soo	39075.000	M	0.666666667
4	400	Ahmed	34992.000	M	1.0
5	300	Davidson	57090.000	F	0.0
6	400	Blaikie	54900.000	F	0.333333333
7	100	Whitney	45700.000	F	0.666666667
8	400	Wetherby	35745.000	F	1.0

由于按性别 (Sex) 划分输入，所以分别对男雇员和女雇员执行 PERCENT_RANK 计算。

示例 2

以下示例返回 Utah 和 Arizona 的一些女雇员的列表并根据薪水以降序顺序排列她们。这里的 PERCENT_RANK 函数用于以降序顺序提供累计总数。

```
SELECT Surname, Salary,
     PERCENT_RANK ( ) OVER ( ORDER BY Salary DESC ) "Rank"
     FROM Employees
WHERE State IN ( 'UT', 'AZ' ) AND Sex IN ( 'F' );
```

此查询会返回以下结果：

	Surname	Salary	Rank
1	Shishov	72995.00	0
2	Jordan	51432.00	0.25
3	Hildebrand	45829.00	0.5
4	Bigelow	31200.00	0.75
5	Bertrand	29800.00	1

使用 PERCENT_RANK 查找最高和最低百分点

您可以使用 PERCENT_RANK 在数据集中查找最高或最低的百分点。在以下示例中，查询会返回其薪水在数据集的最高的五个百分点之内的男雇员。

```
SELECT *
FROM ( SELECT Surname, Salary,
       PERCENT_RANK ( ) OVER ( ORDER BY Salary DESC ) "Rank"
       FROM Employees
       WHERE Sex IN ( 'M' )  )
```



```
        AS DerivedTable ( Surname, Salary, Percent )
WHERE Percent < 0.05;
```

此查询会返回以下结果：

	Surname	Salary	Percent
1	Scott	96300.00	0
2	Sheffield	87900.00	0.025
3	Lull	87900.00	0.025

有关 PERCENT_RANK 函数语法的详细信息，请参见“PERCENT_RANK 函数 [Ranking]”一节《SQL Anywhere 服务器 - SQL 参考》。

行编号函数

行编号函数对分区中的行进行唯一编号。SQL Anywhere 支持两种行编号函数，即 NUMBER 和 ROW_NUMBER。建议您使用 ROW_NUMBER 函数，因为该函数符合 ANSI 标准，从而能够提供许多与 SQL Anywhere NUMBER(*) 函数相同的功能。这是因为，虽然这两个函数执行类似的任务，但 NUMBER 函数有多项限制，而 ROW_NUMBER 函数却没有这些限制。

另请参见

- “NUMBER 函数 [Miscellaneous]”一节 《SQL Anywhere 服务器 - SQL 参考》
- “ROW_NUMBER 函数”一节第 467 页

ROW_NUMBER 函数

ROW_NUMBER 函数唯一地编号其结果中的行。该函数不是秩函数；不过，可在任何可以使用秩函数的情况下使用它，且其行为类似于秩函数。

例如，可在派生表中使用 ROW_NUMBER，以便可以对 ROW_NUMBER 值进行其它限制（甚至可以是连接）。

```
SELECT *
FROM ( SELECT Description, Quantity,
       ROW_NUMBER( ) OVER ( ORDER BY ID ASC ) AS RowNum
FROM Products ) AS DT
WHERE RowNum <= 3
ORDER BY RowNum;
```

此查询会返回以下结果：

Description	Quantity	RowNum
Tank Top	28	1

Description	Quantity	RowNum
V-neck	54	2
Crew Neck	75	3

与秩函数一样，ROW_NUMBER 需要使用 ORDER BY 子句。

当窗口的 ORDER BY 子句遇到非唯一表达式时，ROW_NUMBER 还可返回非确定性结果；在并列情况下，行顺序是不可预知的。

ROW_NUMBER 用于处理整个分区；因此，不能使用 ROW_NUMBER 函数指定 ROWS 或 RANGE 子句。

有关 ROW_NUMBER 函数语法的详细信息，请参见“ROW_NUMBER 函数 [Miscellaneous]”一节 《SQL Anywhere 服务器 - SQL 参考》。

集合函数的数学公式

下面两个表提供了 SQL Anywhere 中支持的所有窗口集合函数的等效数学公式，以供参考。

简单集合函数

Function	Symbol	Formula
SUM(X)		$\sum_{i=1}^{n} x_i$
MAX(X)		$x_i : x_i \geq x_j, i \neq j \; \forall i,j \in n$
MIN(X)		$x_i : x_i \leq x_j, i \neq j \; \forall i,j \in n$
AVG(X)	$\bar{x}$	$\frac{\sum x_i}{n}$
COUNT(*)		n
VAR_SAMP(X)	s_x^2	$\frac{\sum (x_i - \bar{x})^2}{(n-1)}$
VAR_POP(X)	σ_x^2	$\frac{\sum (x_i - \bar{x})^2}{n}$
VARIANCE(X)		identical to VAR_SAMP(X)
STDDEV_SAMP(X)	s_x	$\sqrt{\frac{\sum (x_i - \bar{x})^2}{(n-1)}}$
STDDEV_POP(X)	σ_x	$\sqrt{\frac{\sum (x_i - \bar{x})^2}{n}}$
STDDEV(X)		identical to STDDEV_SAMP(X)

统计集合函数

COVAR_SAMP(Y,X)	*Co-variance*	$s_{xy} = \frac{\sum xy - \frac{(\sum x)(\sum y)}{n}}{(n-1)}$
COVAR_POP(Y,X)	*Co-variance*	$\sigma_{xy} = \frac{\sum xy - \frac{(\sum x)(\sum y)}{n}}{n}$
CORR(Y,X)	*Correlation Coefficient*	$r = \frac{\sum xy - \frac{1}{n}(\sum x)(\sum y)}{(n-1)s_x s_y}$
REGR_AVGX(Y,X)	*Independent mean*	$\bar{x}$
REGR_AVGY(Y,X)	*Dependent mean*	$\bar{y}$
REGR_SLOPE(Y,X)	*Regression Slope*	$b = r\frac{s_y}{s_x}$
REGR_INTERCEPT(Y,X)	*Regression Intercept*	$a = \bar{y} - b\bar{x}$
REGR_R2(Y,X)	*'Goodness-of-fit'*	r^2
REGR_COUNT(Y,X)	*Sample size*	n (non-null (Y, X) pairs)
REGR_SXX(Y,X)	*Sum of squares* (x)	$\sum x^2 - \frac{(\sum x)^2}{n}$
REGR_SYY(Y,X)	*Sum of squares* (y)	$\sum y^2 - \frac{(\sum y)^2}{n}$
REGR_SXY(Y,X)	*Sum of products*	$\sum xy - \frac{(\sum y)(\sum x)}{n}$

使用子查询

目录

使用关系数据库可以在多个表中存储相关的数据。除了可以使用连接从相关表中提取数据外，还可以使用**子查询**进行提取。子查询是指在父 SQL 语句的 SELECT、WHERE 或 HAVING 子句中嵌套的 SELECT 语句。

子查询使某些查询比连接更易于编写，而且某些查询如果不使用子查询将无法进行编写。

子查询可按以下不同的方式进行分类：

- 子查询返回一行还是多行（单行与多行子查询）
- 子查询是相关的还是不相关的
- 子查询是否嵌套于另一子查询中

单行和多行子查询

只能向外部语句返回一行或零行的子查询称为**单行子查询**。单行子查询是在 WHERE 或 HAVING 子句中与比较运算符一起使用的子查询。

可向外部语句返回多行（但只有一列）的子查询称为**多行子查询**。多行子查询与 IN、ANY 或 ALL 子句一起使用。

示例 1：单行子查询

您将产品特有的信息存储在一个名为 Products 的表中，将与销售订单有关的信息存储在另一个名为 SalesOrdersItems 的表中。Products 表包含与各种产品有关的信息。SalesOrdersItems 表包含与客户订单有关的信息。如果公司在产品库存量不足 50 的情况下再次订购产品，就可以通过以下查询回答“哪些产品将要缺货？”这一问题：

```
SELECT ID, Name, Description, Quantity
FROM Products
WHERE Quantity < 50;
```

然而，我们应考虑某产品的订购频率，其结果会对我们更有帮助，因为与很少订购的产品相比，我们更关心频繁订购的产品的库存量是否不足。

您可以使用子查询来确定客户订购的产品的平均数，然后在主查询中使用该平均数查找将要缺货的产品。对于数目小于客户订购的每一类产品的平均数两倍的产品，以下查询将查找这些产品的名称和说明。

```
SELECT Name, Description
FROM Products WHERE Quantity <  2 * (
   SELECT AVG( Quantity )
   FROM SalesOrderItems
   );
```

在 WHERE 子句中，子查询帮助从 FROM 子句中列出的表中选择出现在查询结果中的行。在 HAVING 子句中，子查询帮助选择在查询结果中出现的行组，这些行组是由主查询的 GROUP BY 子句指定的。

示例 2：单行子查询

以下是一个单行子查询的示例，用于计算 Products 表中产品的平均价格。该平均价格随后被传递给外部查询的 WHERE 子句。外部查询会返回低于平均价格的所有产品的 ID、Name 和 UnitPrice。

```
SELECT ID, Name, UnitPrice
FROM Products
WHERE UnitPrice <
  ( SELECT AVG( UnitPrice ) FROM Products )
ORDER BY UnitPrice DESC;
```

ID	Name	UnitPrice
401	Baseball Cap	10.00
300	Tee Shirt	9.00
400	Baseball Cap	9.00

ID	Name	UnitPrice
500	Visor	7.00
501	Visor	7.00

示例 3：使用 IN 的简单多行子查询

假设您想要找出库存不足的产品，还要同时找出这些产品的订单。您可以执行一个 SELECT 语句，其 WHERE 子句中包含一个子查询，如下所示：

```
SELECT *
FROM SalesOrderItems
WHERE ProductID IN
   (  SELECT ID
       FROM Products
       WHERE Quantity < 20 )
ORDER BY ShipDate DESC;
```

在此示例中，该子查询会生成一个列表，其中包含 Products 表内 ID 列中的所有值，这些值符合 WHERE 子句搜索条件。该子查询随后会返回一组行，但只返回一个列。IN 关键字将每个值都视为集合的一个成员，并测试主查询的每个行是否都是该集合的成员。

示例 4：多行子查询与使用 IN、ANY 和 ALL 的对比

SQL Anywhere 示例数据库中有两个表包含财务结果数据。FinancialCodes 表是一个保存财务数据的不同代码及其含义的表。要列出 FinancialData 表中的收入项，请执行以下查询：

```
SELECT *
FROM FinancialData
WHERE Code IN
    ( SELECT Code
        FROM FinancialCodes
        WHERE type = 'revenue' );
```

Year	Quarter	Code	Amount
1999	Q1	r1	1023
1999	Q2	r1	2033
1999	Q3	r1	2998
1999	Q4	r1	3014
2000	Q1	r1	3114
...	...	...	...

可采用类似方式使用 ANY 和 ALL 关键字。例如，以下查询和上一个查询返回同样的结果，但是使用了 ANY 关键字：

```
SELECT *
FROM FinancialData
```

```
WHERE FinancialData.Code = ANY
    (  SELECT FinancialCodes.Code
        FROM FinancialCodes
        WHERE type = 'revenue' );
```

虽然 =ANY 条件与 IN 条件完全相同，但 ANY 还可以与不等式（如 < 或 >）一起使用，从而能够更灵活地使用子查询。

ALL 关键字与 ANY 类似。例如，以下查询列出非收入财务数据：

```
SELECT *
FROM FinancialData
WHERE FinancialData.Code <> ALL
    (  SELECT FinancialCodes.Code
        FROM FinancialCodes
       WHERE type = 'revenue' );
```

这等同于下面使用 NOT IN 的命令：

```
SELECT *
FROM FinancialData
WHERE FinancialData.Code NOT IN
    (  SELECT FinancialCodes.Code
       FROM FinancialCodes
       WHERE type = 'revenue' );
```

相关和不相关子查询

子查询可包含对父语句中所定义对象的引用。这称为**外部引用**。包含外部引用的子查询称为**相关子查询**。相关子查询无法独立于外部查询进行计算，因为子查询使用了父语句的值。即，会针对父语句中的每一行来执行子查询。因此，子查询的结果取决于父语句中正在计算的活动行。

例如，以下语句中的子查询返回的值取决于 Products 表中的活动行。

```
SELECT Name, Description
FROM Products
WHERE Quantity < 2 * (
   SELECT AVG( Quantity )
   FROM SalesOrderItems
   WHERE Products.ID=SalesOrderItems.ProductID );
```

在此示例中，该子查询中的 Products.ID 列就是外部引用。该查询会提取库存量少于某产品平均订购量两倍的产品的名称和说明（具体地讲，是指由主查询中 WHERE 子句测试的产品）。该子查询通过扫描 SalesOrderItems 表来实现此目的。但子查询 WHERE 子句中的 Products.ID 列引用的是*主*查询（而非子查询）的 FROM 子句所指定表中的列。在数据库服务器遍历 Products 表每一行的过程中，它会在对子查询的 WHERE 子句进行计算时使用当前行的 ID 值。

如果子查询中引用的列不存在于由子查询的 FROM 子句引用的表中，但存在于由外部查询的 FROM 子句引用的表中，则执行查询时不会出现错误。SQL Anywhere 通过外部查询中的表名隐式限定子查询中的列。

不包含对父语句中对象的引用的子查询称为**不相关子查询**。在以下示例中，子查询仅计算一个值：SalesOrderItems 表中的平均数量。在计算该查询时，数据库服务器会计算一次该值，然后将 Products 表 Quantity 字段中的每个值与该值进行比较，以确定是否选择相应的行。

```
SELECT Name, Description
FROM Products
WHERE Quantity <  2 * (
   SELECT AVG( Quantity )
   FROM SalesOrderItems );
```

嵌套的子查询

嵌套子查询是指嵌在另一子查询中的子查询。尽管对于您可以定义的嵌套子查询的层数没有限制，但与嵌套层数较少的查询相比，嵌套层数在三层或三层以上的查询的运行时间要长很多。

以下示例使用嵌套子查询来确定，在收费部门的产品订购当天发货的订单的订单 ID 和行 ID。

```
SELECT ID, LineID
FROM SalesOrderItems
WHERE ShipDate = ANY (
   SELECT OrderDate
   FROM SalesOrders
   WHERE FinancialCode IN (
      SELECT Code
      FROM FinancialCodes
      WHERE ( Description = 'Fees' ) ) );
```

ID	LineID
2001	1
2001	2
2001	3
2002	1
...	...

在此示例中，最里面的子查询生成一列其说明是 "Fees" 的财务代码。

```
SELECT Code
FROM FinancialCodes
WHERE ( Description = 'Fees' );
```

下一个子查询查找其代码与最里面的子查询中选择的某一代码相匹配的产品的订单日期。

```
SELECT OrderDate
FROM SalesOrders
WHERE FinancialCode
IN ( subquery-expression );
```

最后，最外面的查询查找在此子查询中找到的某一日期发运的订单的订单 ID 和行 ID。

```
SELECT ID, LineID
FROM SalesOrderItems
WHERE ShipDate = ANY ( subquery-expression );
```

使用子查询代替连接

假设您需要一个按时间顺序排列的订单和下达这些订单的公司的列表，但您希望列出的是公司名而不是它们的客户 ID，则您可以使用连接来获得此结果。

使用连接

要列出 2001 年初以来每个订单的订单 ID、日期和公司名，请执行以下查询：

```
SELECT SalesOrders.ID,
            SalesOrders.OrderDate,
            Customers.CompanyName
FROM SalesOrders
   KEY JOIN Customers
WHERE OrderDate > '2001/01/01'
ORDER BY OrderDate;
```

使用子查询

以下语句使用子查询代替连接来得到相同的结果：

```
SELECT SalesOrders.ID,
   SalesOrders.OrderDate,
   (  SELECT CompanyName FROM Customers
       WHERE Customers.ID = SalesOrders.CustomerID )
FROM SalesOrders
WHERE OrderDate > '2001/01/01'
ORDER BY OrderDate;
```

尽管 SalesOrders 表不是子查询的一部分，子查询也会引用 SalesOrders 表中的 CustomerID 列。而 SalesOrders.CustomerID 列在语句的主体部分引用 SalesOrders 表。

如果只需要另一个表中的一个列，就可以使用子查询代替连接。（回想一下，子查询只能返回一列。）在本例中，您只需要 CompanyName 列，因此可以将连接更改为子查询。

使用外连接

要列出华盛顿州的所有客户以及他们最近的订单 ID，请执行以下查询：

```
SELECT  CompanyName, State,
   ( SELECT MAX( ID )
       FROM SalesOrders
      WHERE SalesOrders.CustomerID = Customers.ID )
FROM Customers
WHERE State = 'WA';
```

CompanyName	State	MAX(SalesOrders.ID)
Custom Designs	WA	2547
It's a Hit!	WA	(NULL)

It's a Hit! 公司没有下任何订单，因此子查询为此客户返回 NULL。使用内连接时，不会列出没有下订单的公司。

您还可以显式地指定一个外连接。在本例中，还需要 GROUP BY 子句。

```
SELECT CompanyName, State,
   MAX( SalesOrders.ID )
FROM Customers
   KEY LEFT OUTER JOIN SalesOrders
WHERE State = 'WA'
GROUP BY CompanyName, State;
```


WHERE 子句中的子查询

WHERE 子句中的子查询是行选择过程的一部分。当行选择条件取决于另一个表的结果时，可以在 WHERE 子句中使用子查询。

示例

查找其库存量小于平均订购量的两倍的产品。

```
SELECT Name, Description
FROM Products WHERE Quantity <  2 * (
   SELECT AVG( Quantity )
   FROM SalesOrderItems );
```

这是一个两步查询：第一步是查找单位订单所请求产品的平均数，然后查找库存数小于该平均数两倍的产品。

由两个步骤组成的查询

SalesOrderItems 表的 Quantity 列存储单位产品类型、客户和订单请求的产品*数量*。该子查询为

```
SELECT AVG( Quantity )
FROM SalesOrderItems;
```

它返回 SalesOrderItems 表中各产品的平均数量，即 25.851413。

下一个查询返回其库存数量小于以前提取的值的两倍的产品的名称和说明。

```
SELECT Name, Description
FROM Products
WHERE Quantity < 2*25.851413;
```

使用一个子查询将两个步骤合并成一个操作。

在 WHERE 子句中使用子查询的目的

WHERE 子句中的子查询是搜索条件的一部分。“查询数据”第 253 页一章介绍了可以在 WHERE 子句中使用的简单搜索条件。

HAVING 子句中的子查询

尽管您通常在 WHERE 子句中使用子查询作为搜索条件，但有时也可以在查询的 HAVING 子句中使用子查询。当子查询出现在 HAVING 子句中时，像 HAVING 子句中的任何表达式一样，它被用作行组选择的一部分。

下面的请求通过在 HAVING 子句中使用子查询来满足查询条件："哪些产品的平均库存数量超过了每个客户订购的每个产品平均数量的两倍？"

示例

```
SELECT Name, AVG( Quantity )
FROM Products
GROUP BY Name
HAVING AVG( Quantity ) > 2* (
   SELECT AVG( Quantity )
   FROM SalesOrderItems
 );
```

name	AVG(Products.Quantity)
Baseball Cap	62.000000
Shorts	80.000000
Tee Shirt	52.333333

该查询的执行过程如下：

- 该子查询计算 SalesOrderItems 表中各产品的平均数量。
- 接着主查询遍历 Products 表，计算每种产品的平均数量，并将它们按产品名分组。
- HAVING 子句随后检查每一平均数量是否超过了子查询所找到的数量的两倍。如果是这样，则主查询返回该行组；否则，将不返回该行组。
- SELECT 子句为每一组生成一个摘要行，并且显示每一产品的名称及其库存平均数量。

如下例所示，您也可以在 HAVING 子句中使用外部引用，但与前一个例子稍有不同。

示例

此示例查找平均订购数量超过库存数量一半的那些产品的产品 ID 号和行 ID 号。

```
SELECT ProductID, LineID
FROM SalesOrderItems
GROUP BY ProductID, LineID
HAVING 2* AVG( Quantity ) > (
   SELECT Quantity
   FROM Products
   WHERE Products.ID = SalesOrderItems.ProductID );
```

ProductID	LineID
601	3
601	2
601	1
600	2
...	...

在此示例中，子查询必须生成与 HAVING 子句所测试的行组相对应的产品的库存量。子查询使用外部引用 SalesOrderItems.ProductID 来选择该特定产品的记录。

使用比较符号的子查询返回单个值

此查询使用比较运算符 >，暗示该子查询必须只返回一个值。在此例中，它恰好返回了一个值。由于 Products 表的 ID 字段是主键，因此 Products 表中只有一条记录与任何特定产品 ID 相对应。

测试子查询

“查询数据”第 253 页一章介绍了可以在 HAVING 子句中使用的简单搜索条件。因为子查询只是在 WHERE 或 HAVING 子句中出现的表达式，所以子查询中的搜索条件看上去很常见。

这些子查询测试包括：

- **子查询比较测试** 将表达式的值与子查询为主查询的表中的每一记录生成的单个值进行比较。比较测试使用随子查询一起提供的运算符（=、<>、<、<=、>、>=）。
- **定量比较测试** 将表达式的值与子查询生成的每一组值进行比较。
- **子查询集合成员资格测试** 检查表达式的值是否与子查询生成的某一组值匹配。
- **存在测试** 检查子查询是否生成一些行。

子查询比较测试

子查询比较测试（=、<>、<、<=、>、>=）是在简单比较测试基础上进行修改后所得到的版本。这两者的唯一区别在于：在子查询比较测试中，在运算符之后的表达式是子查询。此测试用于将主查询某行中的值与子查询生成的*单个*值进行比较。

示例

下面的查询包含一个子查询比较测试的示例：

```
SELECT Name, Description, Quantity
FROM Products
WHERE Quantity <  2 * (
   SELECT AVG( Quantity )
   FROM SalesOrderItems );
```

name	Description	Quantity
Tee Shirt	Tank Top	28
Baseball Cap	Wool cap	12
Visor	Cloth Visor	36
Visor	Plastic Visor	28
...	...	...

以下子查询从 SalesOrderItems 表中检索单个值，即单位客户订单每一类型产品的平均数量。

```
SELECT AVG( Quantity )
FROM SalesOrderItems;
```

然后，主查询将每一库存产品的数量与该值进行比较。

比较测试中的子查询返回一个值

比较测试中的子查询必须只返回一个值。假定有以下查询，其子查询从 SalesOrderItems 表提取两列：

```
SELECT Name, Description, Quantity
FROM Products
WHERE Quantity <  2 * (
   SELECT AVG( Quantity ), MAX( Quantity )
   FROM SalesOrderItems);
```

它返回错误 [子查询只允许一个选择列表项]。

子查询和 IN 测试

您可以使用子查询集合成员资格测试将来自主查询的值与子查询中的多个值进行比较。

子查询集合成员资格测试将主查询中每一行的单个数据值与该子查询生成的单列数据值进行比较。如果主查询的数据值与该列中的*某*一数据值匹配，则该子查询返回 TRUE。

示例

选择领导运输部门或财务部门的雇员的姓名：

```
SELECT GivenName, Surname
FROM Employees
WHERE EmployeeID IN (
   SELECT DepartmentHeadID
   FROM Departments
   WHERE ( DepartmentName='Finance' OR
          DepartmentName = 'Shipping' ) );
```

GivenName	Surname
Mary Anne	Shea
Jose	Martinez

此示例中的子查询从 Departments 表提取与运输部和财务部领导对应的 ID 号。主查询随后返回 ID 号与该子查询找到的两个 ID 号中的一个匹配的雇员的姓名。

```
SELECT DepartmentHeadID
FROM Departments
WHERE ( DepartmentName='Finance' OR
       DepartmentName = 'Shipping' );
```

集合成员资格测试等效于 =ANY 测试

子查询集合成员资格测试等同于 =ANY 测试。以下查询等同于上面示例中的查询。

```
SELECT GivenName, Surname
FROM Employees
WHERE EmployeeID = ANY (
   SELECT DepartmentHeadID
   FROM Departments
```

```
    WHERE ( DepartmentName='Finance' OR
           DepartmentName = 'Shipping' ) );
```

集合成员资格测试的取非

您还可以使用子查询集合成员资格测试来提取列值与由子查询生成的任何值均不相等的那些行。要对集合成员资格测试取非，请在关键字 IN 之前插入 NOT 一词。

示例

下面的查询中的子查询返回不是财务部门或运输部门领导的雇员的名和姓。

```
SELECT GivenName, Surname
FROM Employees
WHERE EmployeeID NOT IN (
   SELECT DepartmentHeadID
   FROM Departments
   WHERE ( DepartmentName='Finance' OR
           DepartmentName = 'Shipping' ) );
```

子查询和 ANY 测试

ANY 测试与 SQL 比较运算符（=、>、<、>=、<=、!=、<>、!>、!<）之一联合使用，它可以将单个值与子查询生成的一列数据值进行比较。为了执行该测试，SQL 使用指定的比较运算符来将测试值与该列中的每一数据值进行比较。如果*任*一比较运算生成的结果为 TRUE，ANY 测试便返回 TRUE。

与 ANY 一起使用的子查询必须返回单列。

示例

查找在订单 #2005 的第一批产品已发出之后所发的那些订单的订单 ID 和客户 ID。

```
SELECT ID, CustomerID
FROM SalesOrders
WHERE OrderDate > ANY (
   SELECT ShipDate
   FROM SalesOrderItems
   WHERE ID=2005 );
```

ID	CustomerID
2006	105
2007	106
2008	107
2009	108
...	...

执行此查询时，主查询根据订单 #2005 *每个*产品的发运日期测试每个订单的订单日期。如果某个订单日期晚于订单 #2005 *一批*货物的发运日期，则 SalesOrders 中的该 ID 和客户 ID 将成为结果集的一部分。ANY 测试与 OR 运算符类似：可以将上述查询理解为 "此销售订单是在订单 #2005 的第一批产品发运之后所下，是在订单 #2005 的第二批产品发运之后所下，还是……"

理解 ANY 运算符

ANY 运算符可能会造成一点混淆。它很容易使人将该查询理解为 [返回在已发运订单 #2005 的任何产品之后所下的那些订单]。但这意味着该查询将返回订单 #2005 的*所有*产品都已发运后所下订单的订单 ID 和客户 ID—这并不是该查询的结果。

请试着改为这样理解该查询："对于在发运了订单 #2005 的*至少一个*产品后所下的订单，返回它们的订单 ID 和客户 ID。"使用关键字 SOME 可以更直观地表达出该查询。以下查询等同于上一查询。

```
SELECT ID, CustomerID
FROM SalesOrders
WHERE OrderDate > SOME (
   SELECT ShipDate
   FROM SalesOrderItems
   WHERE ID=2005 );
```

关键字 SOME 等同于关键字 ANY。

与 ANY 运算符有关的说明

ANY 测试还有两个重要特性：

- **子查询结果集为空**　如果子查询生成空结果集，则 ANY 测试返回 FALSE。这是有意义的，因为如果没有任何结果，则至少有一个结果满足比较测试就不成立了。
- **子查询结果集中的值为 NULL**　假设在子查询结果集中至少有一个 NULL 值。如果对于结果集中的所有非 NULL 数据值比较测试均为 FALSE，则 ANY 搜索返回 UNKNOWN。原因在于，在此情况下，您无法明确断定其比较测试为真的子查询是否具有值。可能有也可能没有值，具体情况取决于结果集中 NULL 数据的*正确*值。有关 ANY 搜索条件的详细信息，请参见“ANY 和 SOME 搜索条件”一节 《SQL Anywhere 服务器 - SQL 参考》。

子查询和 ALL 测试

ALL 测试与 SQL 比较运算符（=、>、<、>=、<=、!=、<>、!>、!<）之一配合使用，它可以将单个值与子查询生成的数据值进行比较。为了执行该测试，SQL 使用指定的比较运算符将测试值与结果集中的每一数据值进行比较。如果所有比较运算生成的结果都为 TRUE，则 ALL 测试返回 TRUE。

示例

本示例查找在订单 #2001 的所有产品都已发运之后所下的那些订单的订单 ID 和客户 ID。

```
SELECT ID, CustomerID
FROM SalesOrders
WHERE OrderDate > ALL (
   SELECT ShipDate
   FROM SalesOrderItems
   WHERE ID=2001 );
```

ID	CustomerID
2002	102
2003	103
2004	104
2005	101
...	...

执行此查询时，主查询根据订单 #2001 *每个*产品的发运日期测试每个订单的订单日期。如果某个订单日期晚于订单 #2001 *每批*货物的发运日期，则 SalesOrders 表中的该 ID 和客户 ID 将成为结果集的一部分。ALL 测试与 AND 运算符类似：可以将上述查询理解为 "此销售订单是在订单 #2001 的第一个产品发运之前所下，是在订单 #2001 的第二个产品发运之前所下，还是……"

有关 ALL 运算符的说明

ALL 测试还有三个重要特性：

- **子查询结果集为空**　如果子查询生成空结果集，则 ALL 测试返回 TRUE。这是有意义的，因为如果没有结果，则对结果集中每一个值的比较测试便为真。
- **子查询结果集中的值为 NULL**　如果对结果集中任意值的比较测试为假，则 ALL 搜索返回 FALSE。如果所有值均为真，则它返回 TRUE。否则，它返回 UNKNOWN—举例来说，如果子查询结果集中有 NULL 值，但搜索条件对所有非 NULL 值均为 TRUE，就可能会发生这种情况。
- **对 ALL 测试取非**　以下表达式*不*等同。

```
NOT a = ALL (subquery)
a <> ALL (subquery)
```

有关此测试的详细信息，请参见“跟在 ANY、ALL 或 SOME 之后的子查询”一节第 490 页。

子查询和 EXISTS 测试

子查询比较测试中使用的子查询和集合成员资格测试中使用的子查询都返回子查询表中的数据值。不过，有时候您可能更关心子查询是否返回*一些*结果，而不是返回*哪些*结果。存在测试 (EXISTS) 就是用来检查子查询是否生成一些查询结果行的。如果子查询生成一行或多行结果，则 EXISTS 测试就返回 TRUE。否则，它会返回 FALSE。

示例

下面的示例是一个使用子查询表示的请求："哪些客户在 2001 年 7 月 13 日以后下了订单？"

```
SELECT GivenName, Surname
FROM Customers
WHERE EXISTS (
   SELECT *
   FROM SalesOrders
```



```
WHERE ( OrderDate > '2001-07-13' ) AND
      ( Customers.ID = SalesOrders.CustomerID ) );
```

GivenName	Surname
Almen	de Joie
Grover	Pendelton
Ling Ling	Andrews
Bubba	Murphy

有关存在测试的说明

在本例中，对于 Customers 表中的每一行，该子查询检查该客户 ID 是否与在 2001 年 7 月 13 日之后下了订单的某位客户相对应。如果对应，则该查询将从主表提取该客户的名和姓。

EXISTS 测试不使用子查询的结果，它只检查子查询是否生成了任何行。因此，应用于以下两个子查询的存在测试返回相同的结果。它们是子查询，无法自行处理，因为它们引用的是属于主查询（但不属于子查询）的 Customers 表。

有关详细信息，请参见“相关和不相关子查询”一节第 475 页。

```
SELECT *
FROM Customers, SalesOrders
WHERE ( OrderDate > '2001-07-13' ) AND
      ( Customers.ID = SalesOrders.CustomerID )

SELECT OrderDate
FROM Customers, SalesOrders
WHERE ( OrderDate > '2001-07-13' ) AND
      ( Customers.ID = SalesOrders.CustomerID );
```

SalesOrders 表中的哪些列出现在 SELECT 语句中并无多大关系，不过按照惯例，将使用 "SELECT *" 表示法。

对存在测试取非

您可以使用 NOT EXISTS 形式颠倒 EXISTS 测试的逻辑。在此情况下，如果子查询没有生成任何行，则该测试返回 TRUE，否则返回 FALSE。

相关子查询

您可能已经注意到，子查询中包含对 Customers 表中 ID 列的引用。对主表中的列或表达式的引用称作**外部引用**，并且这种子查询是**相关的**。从概念上说，SQL 处理上述查询的方式是遍历 Customers 表并为每个客户执行子查询。如果 SalesOrders 表中的订单日期晚于 2001 年 7 月 13 日，并且 Customers 表和 SalesOrders 表中的客户 ID 匹配，就会出现 Customers 表中的名和姓。因为子查询引用主查询，所以，本节中的子查询与前几节中的那些子查询不同，如果试图由子查询本身自行运行，将会返回错误。

优化程序自动将子查询转换为连接

查询优化程序会自动将许多使用子查询的查询改写为连接。执行这一转换不需要用户执行任何操作。本节介绍哪些子查询可以转换为连接，以便您可以了解数据库中查询的性能。

要使多层查询能够改写为连接的形式，必须满足一定条件，而对不同的运算符类型以及不同的查询和子查询结构，这些条件也会有差异。请回想一下，在查询的 WHERE 子句中出现子查询时的情况，其形式如下

SELECT *select-list*
FROM *table*
WHERE
[**NOT**] *expression comparison-operator* **(** *subquery-expression* **)**
| [**NOT**] *expression comparison-operator* { **ANY** | **SOME** } **(** *subquery-expression* **)**
| [**NOT**] *expression comparison-operator* **ALL (** *subquery-expression* **)**
| [**NOT**] *expression* **IN (** *subquery-expression* **)**
| [**NOT**] **EXISTS (** *subquery-expression* **)**
GROUP BY *group-by-expression*
HAVING *search-condition*

例如，考虑以下请求："Clarke 女士和 Suresh 女士何时下的订单，是通过哪些销售代表下的订单？"可以通过以下查询回答上述请求：

```
SELECT OrderDate, SalesRepresentative
FROM SalesOrders
WHERE CustomerID IN (
   SELECT ID
   FROM Customers
   WHERE Surname = 'Clarke' OR GivenName = 'Suresh' );
```

OrderDate	SalesRepresentative
2001-01-05	1596
2000-01-27	667
2000-11-11	467
2001-02-04	195
...	...

该子查询生成与其姓名在 WHERE 子句中列出的两个客户相对应的客户 ID 的列表，并且主查询查找与这两位女士的订单相对应的订单日期和销售代表。

可以使用连接来回答同样的问题。下面是该查询的替代形式，使用了两表连接：

```
SELECT OrderDate, SalesRepresentative
FROM SalesOrders, Customers
WHERE CustomerID=Customers.ID AND
  ( Surname = 'Clarke' OR GivenName = 'Suresh' );
```

此查询形式将 SalesOrders 表连接到 Customers 表，以查找每个客户的订单，然后只返回 Suresh 和 Clarke 的记录。

子查询有效但连接无效的情形

在某些情况下，子查询有效，但连接无效。例如：

```
SELECT Name, Description, Quantity
FROM Products
WHERE Quantity <  2 * (
   SELECT AVG( Quantity )
   FROM SalesOrderItems );
```

name	Description	Quantity
Tee Shirt	Tank Top	28
Baseball Cap	Wool cap	12
Visor	Cloth Visor	36
...	...	...

在此例中，内部查询是汇总查询，而外部查询不是汇总查询，因此，无法通过一个简单连接来合并这两个查询。

另请参见

- “连接：从多个表检索数据”第 361 页

跟在比较运算符之后的子查询

跟在比较运算符（=、>、<、>=、<=、!=、<>、!>、!<）之后的子查询称为比较。如果这些子查询满足以下条件，则优化程序会将其转换为连接：

- 对于主查询的每一行只返回一个值。
- 不包含 GROUP BY 子句
- 不包含关键字 DISTINCT
- 不是 UNION 查询
- 不是汇总查询

示例

假设 "Suresh 的产品是何时订购的，是通过哪一个销售代表订购的？" 这一请求被表示为以下子查询

```
SELECT OrderDate, SalesRepresentative
FROM SalesOrders
WHERE CustomerID = (
   SELECT ID
   FROM Customers
   WHERE GivenName = 'Suresh' );
```

此查询满足上述条件，因此可转换为使用连接的查询：

```
SELECT OrderDate, SalesRepresentative
FROM SalesOrders, Customers
WHERE CustomerID=Customers.ID AND
  ( Surname = 'Clarke' OR GivenName = 'Suresh' );
```

不过，"查找其库存量小于平均订购量的两倍的产品" 这一请求无法转换为连接，因为该子查询包含集合函数 AVG：

```
SELECT Name, Description
FROM Products
WHERE Quantity <  2 * (
   SELECT AVG( Quantity )
   FROM SalesOrderItems );
```

跟在 ANY、ALL 或 SOME 之后的子查询

跟在关键字 ALL、ANY 或 SOME 之后的子查询称为定量比较。如果这些子查询满足以下条件，则优化程序会将其转换为连接：

- 主查询不包含 GROUP BY 子句，并且不是集合查询，或子查询只返回一个值。
- 子查询不包含 GROUP BY 子句。
- 子查询不包含关键字 DISTINCT。
- 子查询不是 UNION 查询。
- 子查询不是集合查询。
- 不得对合取式 '*expression comparison-operator* { **ANY** | **SOME** } **(** *subquery-expression* **)**' 取非。
- 必须对合取式 '*expression comparison-operator* **ALL (** *subquery-expression* **)**' 取非。

其中的前 4 个条件相对易于理解。

示例

"Clarke 女士和 Suresh 女士何时下的订单，并且是向哪些销售代表下的订单" 这一请求可按以下子查询形式进行处理：

```
SELECT OrderDate, SalesRepresentative
FROM SalesOrders
WHERE CustomerID = ANY (
   SELECT ID
   FROM Customers
   WHERE Surname = 'Clarke' OR GivenName = 'Suresh' );
```

上述子查询也可以使用连接形式表示

```
SELECT OrderDate, SalesRepresentative
FROM SalesOrders, Customers
WHERE CustomerID=Customers.ID AND
  ( Surname = 'Clarke' OR GivenName = 'Suresh' );
```

不过，"Clarke 女士和 Suresh 女士以及同样是客户的任何雇员何时下的订单？" 这一请求将以联合查询形式表示，并且无法转换为连接：

```
SELECT OrderDate, SalesRepresentative
FROM SalesOrders
WHERE CustomerID = ANY (
   SELECT ID
   FROM Customers
   WHERE Surname = 'Clarke' OR GivenName = 'Suresh'
   UNION
   SELECT EmployeeID
   FROM Employees );
```

类似地，"查找在所有产品第一个发运日期后未发运的那些订单的订单 ID 和客户 ID" 这一请求将表示为集合查询，因此无法转换为连接：

```
SELECT ID, CustomerID
FROM SalesOrders
WHERE NOT OrderDate > ALL (
   SELECT FIRST ( ShipDate )
   FROM SalesOrderItems
   ORDER BY ShipDate );
```

对具有 ANY 和 ALL 运算符的子查询取非

第五个条件更令人费解一些。采用以下形式的查询将被转换为连接：

SELECT *select-list*
FROM *table*
WHERE NOT *expression comparison-operator* **ALL (** *subquery-expression* **)**

SELECT *select-list*
FROM *table*
WHERE *expression comparison-operator* **ANY (** *subquery-expression* **)**

不过，以下查询不会被转换为连接：

SELECT *select-list*
FROM *table*
WHERE *expression comparison-operator* **ALL (** *subquery-expression* **)**

SELECT *select-list*
FROM *table*
WHERE NOT *expression comparison-operator* **ANY (** *subquery-expression* **)**

前两个查询是等同的，后两个查询也是如此。请回想一下，ANY 运算符类似于 OR 运算符，但其参数数量是可变的；类似地，ALL 运算符也与 AND 运算符类似。例如，以下两个表达式是等同的：

```
NOT ( ( X > A ) AND ( X > B ) )
( X <= A ) OR ( X <= B )
```

以下两个表达式也是等同的：

```
WHERE NOT OrderDate > ALL (
   SELECT FIRST ( ShipDate )
   FROM SalesOrderItems
   ORDER BY ShipDate )

WHERE OrderDate <= ANY (
   SELECT FIRST ( ShipDate )
   FROM SalesOrderItems
   ORDER BY ShipDate )
```

对 ANY 和 ALL 表达式取非

一般来说，以下表达式是等同的：

NOT *column-name operator* **ANY (** *subquery-expression* **)**

column-name inverse-operator **ALL (** *subquery-expression* **)**

以下表达式通常也是等同的：

NOT *column-name operator* **ALL (** *subquery-expression* **)**

column-name inverse-operator **ANY (** *subquery-expression* **)**

其中 *inverse-operator* 是通过对 *operator* 取非而获得的，如下表所示：

运算符	反向运算符
=	<>
<	=>
>	=<
=<	>
=>	<
<>	=

跟在 IN 之后的子查询

仅当满足以下条件时，优化程序才会对跟在 IN 关键字之后的子查询进行转换：

- 主查询不包含 GROUP BY 子句，并且不是集合查询，或子查询只返回一个值。
- 子查询不包含 GROUP BY 子句。
- 子查询不包含关键字 DISTINCT。
- 子查询不是 UNION 查询。
- 子查询不是集合查询。
- 不得对合取式 `'expression IN ( subquery-expression )'` 取非。

示例

因此，由以下查询表示的请求 "查找还是部门领导的雇员的姓名"，由于符合条件，将转化为连接查询。

```
SELECT GivenName, Surname
FROM Employees
WHERE EmployeeID IN (
```



```
   SELECT DepartmentHeadID
   FROM Departments
   WHERE ( DepartmentName ='Finance' OR
          DepartmentName = 'Shipping' ) );
```

不过，如果通过 UNION 查询表示请求 "查找自身是部门领导或客户的雇员的姓名"，则不会将该查询转换为连接。

无法转换跟随在 IN 运算符之后的 UNION 查询

```
SELECT GivenName, Surname
FROM Employees
WHERE EmployeeID IN (
   SELECT DepartmentHeadID
   FROM Departments
   WHERE ( DepartmentName='Finance' OR
          DepartmentName = 'Shipping' )
   UNION
   SELECT CustomerID
   FROM SalesOrders);
```

类似地，如下所示，请求 "查找不是部门领导的雇员的姓名" 是以取非的子查询来表示，因此无法进行转换

```
SELECT GivenName, Surname
FROM Employees
  WHERE NOT EmployeeID IN (
   SELECT DepartmentHeadID
   FROM Departments
   WHERE ( DepartmentName='Finance' OR
          DepartmentName = 'Shipping' ) );
```

IN 或 ANY 子查询转化为连接所必需具备的条件是相同的。这是因为两个表达式逻辑上是等同的。

具有 IN 运算符的查询转换为具有 ANY 运算符的查询

在某些情况下，SQL Anywhere 将具有 IN 运算符的查询转换为具有 ANY 运算符的查询，并决定是否将子查询转换为连接。例如，以下两个表达式是等同的：

WHERE *column-name* **IN(** *subquery-expression* **)**

WHERE *column-name* **= ANY(** *subquery-expression* **)**

同样，以下两个查询是等同的：

```
SELECT GivenName, Surname
FROM Employees
WHERE EmployeeID IN (
   SELECT DepartmentHeadID
   FROM Departments
   WHERE ( DepartmentName='Finance' OR
          DepartmentName = 'Shipping' ) );

SELECT GivenName, Surname
FROM Employees
WHERE EmployeeID = ANY (
   SELECT DepartmentHeadID
   FROM Departments
   WHERE ( DepartmentName='Finance' OR
          DepartmentName = 'Shipping' ) );
```

跟在 EXISTS 之后的子查询

仅当满足以下条件时，优化程序才会对跟在 EXISTS 关键字之后的子查询进行转换：

- 主查询不包含 GROUP BY 子句，并且不是集合查询，或子查询只返回一个值。
- 联合 [EXISTS（子查询）] 不被取非。
- 该子查询是相关的；即，它包含外部引用。

示例

"哪些客户在 2001 年 7 月 13 日之后下了订单？" 这一请求可使用这样的查询来表示，该查询未取非的子查询包含外部引用 **Customers.ID = SalesOrders.CustomerID**，因此可以使用以下连接来表示：

```
SELECT GivenName, Surname
FROM Customers
WHERE EXISTS (
   SELECT *
   FROM SalesOrders
   WHERE ( OrderDate > '2001-07-13' ) AND
         ( Customers.ID = SalesOrders.CustomerID ) );
```

EXISTS 关键字用于指示数据库服务器检查是否存在空结果集。在使用内连接时，数据库服务器只自动显示这样的行：这些行的数据来自 FROM 子句中的所有表。因此，此查询返回的行与以下具有子查询的查询所返回的行相同：

```
SELECT DISTINCT GivenName, Surname
FROM Customers, SalesOrders
WHERE ( SalesOrders.OrderDate > '2001-07-13' ) AND
      ( Customers.ID = SalesOrders.CustomerID );
```

添加、更改和删除数据

目录

数据修改语句

用于添加、更改或删除数据的语句称为**数据修改**语句，也称为 SQL 的**数据修改语言** (DML) 部分。三种主要的 DML 语句是：

- **INSERT 语句** 向表中添加新行
- **UPDATE 语句** 更改表中现有的行
- **DELETE 语句** 从表中删除特定的行

任何单个 INSERT、UPDATE 或 DELETE 语句都只更改一个表或视图中的数据。

除了以上这些语句之外，如果要批量装载和删除数据，则 LOAD TABLE 和 TRUNCATE TABLE 语句将十分有用。

另请参见

- “INSERT 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “UPDATE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “DELETE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》

数据修改的权限

只有在要修改的数据库表上具有适当的权限之后，才可以执行数据修改语句。数据库管理员和数据库对象的所有者使用 GRANT 和 REVOKE 语句来决定哪些用户有权使用哪些数据修改功能。

可以将权限授予个别用户、组或 PUBLIC 组。有关权限的详细信息，请参见“管理用户 ID、特权和权限” 《SQL Anywhere 服务器 - 数据库管理》。

事务和数据修改

修改数据时，回滚日志存储受每个数据修改语句影响的每一行的旧状态和新状态的副本。这意味着如果您开始一个事务，但意识到犯了一个错误，于是将事务回退，将数据库恢复到它以前的状态。请参见“使用事务和隔离级别”第 99 页。

使更改成为永久更改

执行 COMMIT 语句可使所有更改成为永久更改。

应在各组语句组合在一起有意义后使用 COMMIT 语句。例如，如果您要将钱款从一个客户的帐户转到另一个帐户，则应将这笔钱款加到一个客户的帐户上，然后从另一个帐户上删除这笔钱款，然后再进行提交；因为在这种情况下，数据库中的钱款多于或少于原来的数额都是没有意义的。

可以将 auto_commit 选项设置为 On，从而指示 Interactive SQL 自动提交更改。这是 Interactive SQL 选项。如果将 auto_commit 设置为 On，则 Interactive SQL 在每次执行插入、更新和删除语句

后都发出 COMMIT 语句。这会大大降低性能。因此，最好还是将 auto_commit 选项设置保留为缺省值 Off。

小心使用 COMMIT
尝试使用本教程中的示例时，请务必注意，在确定您要对数据库进行永久更改前，不要提交任何更改。请参见“COMMIT 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

另请参见

- “Interactive SQL 选项”一节 《SQL Anywhere 服务器 - 数据库管理》

取消更改

可以取消所做的任何未提交的更改。SQL 允许您使用 ROLLBACK 语句撤消最后一次提交后进行的所有更改。此语句可撤消您最后一次使更改成为永久更改后对数据库所做的所有更改。请参见“ROLLBACK 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

事务和数据恢复

在发生系统故障或电源中断的情况下，SQL Anywhere 可保护数据库的完整性。可以通过多种方法恢复数据库服务器。例如，SQL Anywhere 在一个单独的驱动器上存储的日志文件可用于恢复数据。使用日志文件进行恢复时，SQL Anywhere 不必频繁更新数据库，从而可以提高数据库服务器的性能。

事务处理允许数据库服务器标识数据处于一致状态时的情况。事务处理可确保在某一事务因任何原因而未成功完成的情况下，撤消或回退整个事务。数据库完全不受失败事务的影响。

SQL Anywhere 中的事务处理功能可以确保安全地处理事务内容，即使在事务处理过程中出现系统故障的情况下也是如此。

另请参见

- “备份和数据恢复” 《SQL Anywhere 服务器 - 数据库管理》

参照完整性

SQL Anywhere 自动检查数据，以在插入、更新和删除时找出数据中的某些常见错误。此类有效性检查可检查数据库中各表间的数据完整性，因而称作**实施参照完整性**。请参见“实施实体完整性和参照完整性”一节第 92 页。

使用 INSERT 添加数据

可以使用 INSERT 语句向数据库中添加行。INSERT 语句具有两种形式：可以使用 VALUES 关键字或 SELECT 语句：

使用 VALUES 的 INSERT

VALUES 关键字指定新行中某些列或所有列的值。使用 VALUES 关键字的 INSERT 语句语法的简化形式如下：

INSERT [**INTO**] *table-name* [(*column-name*, ...)]
VALUES (*expression*, ...)

如果您为表中的每列都提供一个值，并且所提供值的顺序是使用 SELECT * 执行查询时它们显示的顺序，则可以省略列名列表。

通过 SELECT 执行 INSERT

可以在 INSERT 语句中使用 SELECT 语句从一个或多个表中取得值。如果您正向其中插入数据的表有大量的列，您还可以使用 WITH AUTO NAME 以简化语法。使用 WITH AUTO NAME 时，您只需在 SELECT 语句中而不必在 INSERT 和 SELECT 语句中指定列名。SELECT 语句中的名称必须是列引用或带别名的表达式。

使用 SELECT 语句的 INSERT 语句语法的简化版本如下：

INSERT [**INTO**] *table-name*
[**WITH AUTO NAME**] *select-statement*

有关 INSERT 语句的详细信息，请参见“INSERT 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

将值插入行的所有列

以下 INSERT 语句将一个新行添加到 Departments 表中，为该行中的每列都提供一个值：

```
INSERT INTO Departments
VALUES ( 702, 'Eastern Sales', 902 );
```

注意

- 键入值的顺序应该和原始 CREATE TABLE 语句中列名的顺序相同，即第一个是 ID 编号，然后是名称，最后是部门主管 ID。
- 将值用括号括起来。
- 将所有字符数据括在单引号中。
- 对每个要添加的行使用单独的 INSERT 语句。

将值插入特定的列

可以将数据添加到行的某些列中，方法是只指定这些列和它们的值。将不包括在列列表中的所有其它列定义为允许使用 NULL 值或具有缺省值。如果您跳过某个具有缺省值的列，缺省值将出现在该列中。

如果只在两列（例如，DepartmentID 和 DepartmentName）中添加数据，则需要如下语句：

```
INSERT INTO Departments ( DepartmentID, DepartmentName )
VALUES ( 703, 'Western Sales' );
```

DepartmentHeadID 没有缺省值，但是接受空值。因此将为该列自动指派 NULL。

输入 ROLLBACK 语句取消对数据库的这些更改：

```
ROLLBACK;
```

尽管所指定列的顺序不需要与表中列的顺序匹配，但是它必须与所指定插入值的顺序匹配。

指定列和未指定列的插入值

根据在 INSERT 语句中指定的内容在行中插入值。如果没有为列指定值，插入的值将视列设置而定，如是否允许 NULL 值、是否使用 DEFAULT 等。在某些情况下，插入操作可能失败并返回错误。下表显示根据要插入的值（如果有）和列设置可能得出的结果：

要插入的值	可为空	不可为空	可为空，带有 DEFAULT	不可为空，带有 DEFAULT	不可为空，带有 DEFAULT AUTOINCREMENT
<无>	NULL	SQL 错误	DEFAULT 值	DEFAULT 值	DEFAULT 值
NULL	NULL	SQL 错误	NULL	SQL 错误	DEFAULT 值
指定值	指定值	指定值	指定值	指定值	指定值

缺省情况下，除非您在创建表时在列定义中显式规定 NOT NULL，否则列允许使用 NULL 值。可以使用 allow_nulls_by_default 选项变更此缺省设置。也可以使用 ALTER TABLE 语句更改指定列是否允许使用 NULL 值。请参见“allow_nulls_by_default 选项 [兼容性]”一节 《SQL Anywhere 服务器 - 数据库管理》和“ALTER TABLE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

使用约束限制列数据

可以为列或域创建约束。约束控制是否能够添加某个数据类型。请参见“使用表和列约束”一节 第 85 页。

显式插入空值

可以通过输入 NULL 将空值显式插入列中。不要将 NULL 括在引号中，否则它将被当做一个字符串。例如，以下语句将 NULL 显式插入 DepartmentHeadID 列中：

```
INSERT INTO Departments
VALUES ( 703, 'Western Sales', NULL );
```

使用缺省值提供值

可以将某个列定义为即使该列未接收到任何值，也会在插入行时自动填上缺省值。这可通过为该列提供缺省值来实现。请参见“使用列缺省值”一节第 79 页。

使用 SELECT 添加新行

若要将值从一个或多个表中取出并添加到另一个表中，您可以在 INSERT 语句中使用 SELECT 子句。SELECT 子句可以将值插入一行中的某些列或所有列中。

仅为某些列插入值这种做法在从现有表中取值时会用得上。然后，可以使用 UPDATE 为其它列添加值。

在为表中的某些列（但不是全部列）插入值之前，对于不插入值的那些列，请确保其存在缺省值，或者为这些列指定 NULL 值。否则，将出现错误。

当将一个表的行插入到另一个表时，这两个表的结构必须兼容—也就是说，匹配的列必须是相同的数据类型或者 SQL Anywhere 可以自动转换的数据类型。

示例

如果两个表中列的顺序相同，则不需要在任何一个表中指定列名。例如，假设有一个名为 NewProducts 的表，其模式与 Products 表相同，并且其中包含一些您想要添加到 Products 表的产品信息行。可以执行以下语句：

```
INSERT Products
SELECT *
FROM NewProducts;
```

将数据插入某些列中

就像使用 VALUES 子句一样，您可以使用 SELECT 语句将数据添加到行中的某些列（但不是所有列）。只需在 INSERT 子句中指定要向其中添加数据的列即可。

插入同一表中的数据

您可以根据表中的其它数据将数据插入同一表中。这实质上意味着复制全部或部分行。

例如，您可以根据现有产品将新产品插入 Products 表中。以下语句将新的 "Extra Large" T 恤衫（短背心、V 型领和水手领等样式）添加到 Products 表中。标识号比现有大小的衬衫大 30：

```
INSERT INTO Products
SELECT ID + 30, Name, Description,
    'Extra large', Color, 50, UnitPrice, NULL
FROM Products
WHERE Name = 'Tee Shirt';
```

插入文档和图像

如果要在数据库中存储文档或图像，可以编写一个应用程序，该应用程序将文件的内容读入一个变量中，然后将该变量作为 INSERT 语句的值提供。请参见“如何使用预准备语句”一节 《SQL Anywhere 服务器 - 编程》和“SET 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

还可以使用 xp_read_file 系统函数将文件内容插入表中。如果要从 Interactive SQL（或者不提供完整编程语言的某些其它环境）插入文件内容，该函数将非常有用。

使用此函数需要具有 DBA 权限。

示例

在本示例中，创建一个表，然后将一个图像插入该表的某个列中。可以通过 Interactive SQL 执行这些步骤。

1. 创建一个表以保存某些图像。

```
CREATE TABLE Pictures
( C1 INT DEFAULT AUTOINCREMENT PRIMARY KEY,
   Filename VARCHAR(254),
   Picture LONG BINARY );
```

2. 将 *portrait.gif*（位于数据库服务器的当前工作目录中）的内容插入表中。

```
INSERT INTO Pictures ( Filename, Picture )
VALUES ( 'portrait.gif',
   xp_read_file( 'portrait.gif' ) );
```

另请参见

- “xp_read_file 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》
- “搭配使用 openxml 和 xp_read_file”一节第 645 页
- “存储 BLOB”一节第 5 页
- “CREATE TABLE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “INSERT 语句”一节 《SQL Anywhere 服务器 - SQL 参考》

使用 UPDATE 更改数据

可以使用 UPDATE 语句（后接表或视图的名称）更改表中的单个行、一组行或所有行。像在所有数据修改语句中一样，一次只能更改一个表或视图中的数据。

UPDATE 语句指定您要更改的行和新数据。新数据可以是一个常量、指定的表达式或从其它表取出的数据。

如果 UPDATE 语句违反了完整性约束，则不会进行更新并将显示一条错误消息。例如，如果正被添加的一个值是错误的数据类型，或者如果它违反为所涉及的某个列或数据类型定义的约束，则将不会进行更新。

UPDATE 语法

UPDATE 语法的简化版本如下：

UPDATE *table-name*
SET *column_name = expression*
WHERE *search-condition*

如果 Newton Ent. 公司（在 SQL Anywhere 示例数据库的 Customers 表中）被 Einstein, Inc. 接管，则可以使用如下语句更新该公司的名称：

```
UPDATE Customers
SET CompanyName = 'Einstein, Inc.'
WHERE CompanyName = 'Newton Ent.';
```

可以在 WHERE 子句中使用任何表达式。如果不确定公司名称是如何拼写的，则可以使用如下所示的语句尝试更新名为 Newton 的任何公司：

```
UPDATE Customers
SET CompanyName = 'Einstein, Inc.'
WHERE CompanyName LIKE 'Newton%';
```

搜索条件不需要引用要更新的列。Newton Entertainments 的公司 ID 是 109。由于 ID 值是表的主键，所以可以确定使用以下语句更新正确的行：

```
UPDATE Customers
SET CompanyName = 'Einstein, Inc.'
WHERE ID = 109;
```

提示
还可以根据 Interactive SQL 中的结果集修改行。请参见“在 Interactive SQL 中编辑结果集”一节《SQL Anywhere 服务器 - 数据库管理》。

SET 子句

SET 子句指定要更新的列和它们的新值。WHERE 子句确定将被更新的行。如果不具有 WHERE 子句，则将用 SET 子句中提供的值更新所有行中指定的列。

您可以在 SET 子句中提供数据类型正确的任何表达式。

WHERE 子句

WHERE 子句指定要更新的行。例如，以下语句用 "Extra Large" T 恤衫替换 "One Size Fits All" T 恤衫。

```
UPDATE Products
SET Size  = 'Extra Large'
WHERE Name = 'Tee Shirt'
   AND Size = 'One Size Fits All';
```

FROM 子句

您可以使用 FROM 子句从一个或多个表中取出数据并将它们放入正在更新的表中。

使用 INSERT 更改数据

可以使用 INSERT 语句的 ON EXISTING 子句用新值更新表中现有的行（根据主键查寻）。此子句仅可用于拥有主键的表。如果尝试对没有主键的表或代理表使用此子句，则将产生语法错误。

指定 ON EXISTING 子句会使服务器查找每个输入行的主键。如果相应的行不存在，则会插入新行。对于表中已经存在的行，可以选择：

- 为重复的键值生成错误。如果不指定 ON EXISTING 子句，此行为是缺省行为。
- 在没有任何提示的情况下忽略输入行，不生成任何错误。
- 用输入行中的值更新现有的行

有关详细信息，请参见“INSERT 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

使用 DELETE 删除数据

简单的 DELETE 语句格式如下：

DELETE [**FROM**] *table-name*
WHERE *column-name = expression*

您也可以使用更复杂的格式，如下所示

DELETE [**FROM**] *table-name*
FROM *table-list*
WHERE *search-condition*

WHERE 子句

使用 WHERE 子句指定要删除的行。如果未出现任何 WHERE 子句，则 DELETE 语句将删除表中所有的行。

FROM 子句

DELETE 语句第二个位置上的 FROM 子句是一个特殊的功能，允许您从一个或多个表中选择数据，然后从指定的第一个表中删除相应的数据。在 FROM 子句中选择的行指定删除条件。请参见“DELETE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

示例

本示例使用 SQL Anywhere 示例数据库。为了执行示例中的语句，应将选项 wait_for_commit 设置为 On。以下语句只为当前连接执行此操作：

```
SET TEMPORARY OPTION wait_for_commit = 'On';
```

这允许您删除某些行，即使这些行包含被外键引用的主键，但是除非将相应的外键也删除，否则不允许执行 COMMIT。

以下视图显示产品和已售出的产品的价值：

```
CREATE VIEW ProductPopularity as
SELECT  Products.ID,
   SUM( Products.UnitPrice * SalesOrderItems.Quantity )
   AS "Value Sold"
FROM  Products JOIN SalesOrderItems
ON Products.ID = SalesOrderItems.ProductID
GROUP BY Products.ID;
```

使用此视图，可以从 Products 表中删除销售额小于 $20,000 的那些产品。

```
DELETE
FROM Products
FROM Products NATURAL JOIN ProductPopularity
WHERE "Value Sold" < 20000;
```

输入 ROLLBACK 语句取消对数据库的这些更改：

```
ROLLBACK;
```

提示
还可以根据 Interactive SQL 结果集从数据库表中删除行。请参见“在 Interactive SQL 中编辑结果集”一节 《SQL Anywhere 服务器 - 数据库管理》。

从表中删除所有行

可以使用 TRUNCATE TABLE 语句作为快速删除表中所有行的方法。这样要比不带任何条件的 DELETE 语句更快，因为 DELETE 会记录每个更改，而 TRUNCATE 不会记录被删除的单个行。

除非执行 DROP TABLE 语句，否则使用 TRUNCATE TABLE 语句清空的表的表定义仍将保留在数据库中，同时还保留其索引和其它关联的对象。

如果其它表的行通过参照完整性约束引用该表，则不能使用 TRUNCATE TABLE。请从外表中删除这些行，或截断外表然后再截断主表。

截断基表或执行批量装载操作会引起索引（常规或文本）和相关实例化视图中的数据失效。您应该首先截断索引和相关实例化视图中的数据，执行 INPUT 语句，然后重建或刷新索引和实例化视图。请参见“TRUNCATE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》和“TRUNCATE TEXT INDEX 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

TRUNCATE TABLE 语法

TRUNCATE TABLE 的语法如下：

TRUNCATE TABLE *table-name*

例如，要删除 SalesOrders 表中的所有数据，请输入以下语句：

```
TRUNCATE TABLE SalesOrders;
```

TRUNCATE TABLE 语句不触发在该表上定义的触发器。

输入 ROLLBACK 语句取消对数据库的这些更改：

```
ROLLBACK;
```


查询处理

本节介绍查询优化程序以及它如何工作，包括：查询处理的各个阶段、优化程序使用的策略以及如何实现优化程序最佳性能的提示。此外还介绍如何查看和分析查询的执行计划。

查询优化与执行

目录

优化实质上就是为查询生成一个合适的访问计划。当对每个查询完成语法分析之后，优化程序将对该查询进行分析并决定一个使用尽可能少的资源来计算结果的访问计划。优化正好在执行之前开始。如果您正在应用程序中使用游标，优化将在游标打开时开始。与其它许多商业数据库系统不同，SQL Anywhere 通常恰好在每个语句执行之前对该语句进行优化。由于 SQL Anywhere 对每个语句执行实时优化，因此优化程序可以访问主机变量和存储过程变量的值，从而可以更好地进行选择性估计分析。此外，实时优化还允许优化程序根据执行先前的查询后所保存的统计信息来调整其选择。

查询处理阶段

本节介绍一个语句从标注阶段开始到其执行结束所要经过的各阶段。还将介绍作为优化程序设计基础的假定，并讨论选择性估计、开销估计以及查询处理的步骤。

没有结果集的语句（如 UPDATE 或 DELETE 语句）会经历查询处理阶段。

- **标注阶段** 当数据库服务器收到一个查询时，它将使用分析器对该语句进行分析，然后将其转换为查询的代数表示形式，也称为分析树。在此阶段，**分析树**用于语义和语法检查（例如，校验查询中所引用的对象是否存在于目录中）、权限检查、使用定义的参照约束进行 KEY JOIN 和 NATURAL JOIN 转换以及进行非实例化视图扩展。此阶段的输出为一个分析树形式的重写查询，其中包含对初始查询中所引用的所有对象的标注。

- **语义转换阶段** 在此阶段中，查询将进行迭代语义转换。虽然查询仍以加标注的分析树的形式表示，但在此阶段将应用诸如连接排除、DISTINCT 排除和谓语规范化等重写优化。此阶段中的语义转换根据语义转换规则执行，这些规则启发式地应用于分析树表示。请参见“语义查询转换”一节第 513 页。

  包含已由数据库服务器高速缓存的计划的查询将跳过此查询处理阶段。简单语句也可以跳过此查询处理阶段。例如，在语义转换阶段，不会对许多在优化程序跳过中采用启发式计划选择的语句进行处理。SQL 语句的复杂性确定是否对语句应用此阶段。请参见“计划高速缓存”一节第 534 页和“跳过查询处理阶段的资格”一节第 511 页。

- **优化阶段** 优化阶段使用查询的另一种内部表示形式，即查询优化结构，此结构通过分析树构建。请参见“优化程序的工作原理”一节第 525 页。

  包含已由数据库服务器高速缓存的计划的查询将跳过此查询处理阶段。此外，简单语句也可以跳过此查询处理阶段。请参见“计划高速缓存”一节第 534 页和“跳过查询处理阶段的资格”一节第 511 页。

  此阶段分为两个子阶段：

  - **预优化阶段** 预优化阶段将使用稍后在枚举阶段中所需的信息完成优化结构。在此阶段，将对查询进行分析，以查找在查询访问计划中可能使用的所有相关索引和实例化视图。例如，在此阶段，视图匹配算法将确定可能用于满足全部或部分查询的所有实例化视图。此外，优化程序会基于查询谓语分析建立可能用于枚举阶段的替代连接方法，以连接查询的表。在此阶段中，并不会为查询确定最佳的访问计划；此阶段的目的是为枚举阶段做准备。

  - **枚举阶段** 在此阶段中，优化程序使用预优化阶段中生成的构件块为查询枚举可能的访问计划。搜索空间非常大，优化程序将使用专有的枚举算法来生成和修整已生成的访问计划。会对每个计划的开销估计进行计算，以用于将当前计划与目前得到的最佳计划进行比较。在进行这些比较的过程中，高开销计划将被放弃。进行开销估计时会考虑资源利用问题，例如，磁盘和 CPU 操作、预计的中间结果的行数、优化目标、高速缓存大小等等。枚举阶段的输出即为查询的最佳访问计划。

- **计划构建阶段** 计划构建阶段将利用最佳访问计划并构建用于执行查询的查询执行计划的相应最终表示形式。可以在 Interactive SQL 的 [计划查看器] 中看到图形版本的计划。图形式计划具有树形结构，其中每个节点为一个实现特定关系代数操作的物理运算符，例如，[散列连接] 和 [排序分组依据] 分别为实现连接操作和分组依据操作的物理运算符。请参见“读取图形式计划”一节第 571 页。

包含已由数据库服务器高速缓存的计划的查询将跳过此查询处理阶段。请参见“计划高速缓存”一节第 534 页和“跳过查询处理阶段的资格”一节第 511 页。

- **执行阶段** 使用在计划构建阶段中构建的查询执行计划计算查询的结果。

跳过查询处理阶段的资格

几乎所有语句都要经历所有的查询处理阶段。但主要有以下两种例外：从**计划高速缓存**中受益的查询（其计划已由数据库服务器高速缓存的查询），以及**跳过查询**。

- **计划高速缓存** 对于包含在存储过程和用户定义函数中的查询，数据库服务器可能会高速缓存执行计划，以便它们可以被重新使用。对于这类查询，查询执行计划将在执行之后进行高速缓存。下次执行此查询时，会对计划进行检索，执行阶段之前的所有阶段都会被跳过。请参见“计划高速缓存”一节第 534 页。
- **跳过查询** 跳过查询是具有某些特性的简单查询的子类，数据库服务器认为这些特性使得查询能够符合跳过优化程序的条件。跳过优化可以减少构建执行计划所需的时间。

  如果查询被识别为跳过查询，则将采用启发式优化（而不是基于开销的优化）—即，可以跳过语义转换和优化阶段，直接通过查询的分析树表示来构建查询执行计划。

简单查询

简单查询是具有单个查询块和以下特性的 SELECT、INSERT、DELETE 或 UPDATE 语句：

- 查询块不包含子查询或其它诸如 UNION、EXCEPT 和公用表表达式之类的查询块。
- 查询块引用单个基表或实例化视图。
- 查询块可以包括 TOP N、FIRST、ORDER BY 或 DISTINCT 子句。
- 在没有 GROUP BY 或 HAVING 子句的情况下，查询块可以包括集合函数。
- 查询块不包含窗口函数。
- 查询块表达式不包含 NUMBER、IDENTITY 或子查询。
- 在基表中定义的约束为简单表达式。

复杂语句可以在语义转换阶段之后转换为简单语句。出现此转换时，查询可由优化程序跳过进行处理，或使其计划由 SQL Anywhere 服务器进行高速缓存。

强制优化与强制不优化

可以强制满足计划高速缓存条件或跳过优化程序条件的查询由 SQL Anywhere 优化程序进行处理。为此，请将 FORCE OPTIMIZATION 子句与任何 SQL 语句一起使用。

您还可以尝试强制语句跳过优化程序。为此，请使用该语句的 FORCE NO OPTIMIZATION 子句。如果该语句过于复杂以至于无法跳过优化程序（可能是由于数据库选项设置或者模式或查询的特性所导致），则查询会失败并返回错误。

以下语句的 OPTION 子句中允许使用 FORCE OPTIMIZATION 和 FORCE NO OPTIMIZATION 子句：

- “SELECT 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “UPDATE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “INSERT 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “DELETE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》

语义查询转换

为了高效运行，SQL Anywhere 将查询重写为在语义上等效但在语法上不同的形式。SQL Anywhere 可执行许多不同的重写操作。

如果读取访问计划，则经常会发现其并不对应于初始语句的文字解释。例如，为使 SQL 语句更加有效，优化程序会尽可能多地尝试用连接重写子查询。

在查询重写阶段，SQL Anywhere 会执行多个转换以搜索更高效和更方便的查询表示。由于查询可能被重写为语义上等效的查询，则计划看起来可能会与初始查询的文字解释有很大不同。常见的操作包括：

- 排除不必要的 DISTINCT 条件
- 取消子查询嵌套
- 将谓语推入执行了 UNION 或 GROUP 的视图和派生表中
- 优化 OR 和 IN 列表谓语
- 优化 LIKE 谓语
- 将外连接转换为内连接
- 排除外连接和内连接
- 通过谓语推导发现可利用的条件
- 排除不必要的大小写转换
- 将子查询重写为 EXISTS 谓语

注意

如果游标是可更新的，则不能对主查询块执行某些查询重写优化。将游标声明为只读以利用优化。请参见“选择游标类型”一节 《SQL Anywhere 服务器 - 编程》和“DECLARE CURSOR 语句 [ESQL] [SP]”一节 《SQL Anywhere 服务器 - SQL 参考》。

有关在主查询块是可更新游标时不能执行的优化的示例，请参见“排除不必要的内连接和外连接”一节第 519 页。

一些在查询重写阶段执行的重写优化可以在由 REWRITE 函数返回的结果中观察到。请参见“REWRITE 函数 [Miscellaneous]”一节 《SQL Anywhere 服务器 - SQL 参考》。

示例

与 SQL 语言定义不同，一些语言对 AND 和 OR 运算有严格的行为要求。一些语言可确保将首先对左侧条件求值。如果随后可以确定整个条件为真，编译器将确保不对右侧条件求值。

这样，可以将那些在其它情况下需要两个嵌套 IF 语句的条件合并成一个条件。例如，在 C 语言中，您可以按如下方式在使用指针之前测试该指针是否为 NULL。可以使用下面第二个语句中显示的语法代替第一个语句中的嵌套条件：

```
if ( X != NULL ) {
   if ( X->var != 0 ) {
      ... statements ...
```

```
    }
}

if ( X != NULL && X->var != 0 ) {
      ... statements ...
}
```

与 C 不同，SQL 没有此类有关执行顺序的规则。SQL Anywhere 会在适当的时候自由地重新排列这些条件的顺序。由于 SQL 语言规范不区分这些顺序，因此初始形式与重新排序后的形式在语义上是等效的。特别是，查询优化程序可以任意地将 WHERE、HAVING 或 ON 子句中的谓语重新排序。

排除不必要的 DISTINCT 条件

有时 DISTINCT 条件是不必要的。例如，结果中的一个或多个列的属性可能（显式或隐式地）包含了 UNIQUE 条件，因为它是主键。

示例

由于 Products 表包含主键 p.ID（它是结果集的一部分），因此以下命令中的 DISTINCT 关键字是不必要的。

```
SELECT DISTINCT p.ID, p.Quantity
FROM Products p;
```

```
Products<seq>
```

数据库服务器将执行语义上等效的查询：

```
SELECT p.ID, p.Quantity
FROM Products p;
```

同样，以下查询的结果包含两个表的主键，因此结果中的每一行肯定是不重复的。因此，数据库服务器执行此查询时不会在结果集上执行 DISTINCT。

```
SELECT DISTINCT *
FROM SalesOrders o JOIN Customers c
   ON o.CustomerID = c.ID
WHERE c.State = 'NY';
```

```
Work[ HF[ c<seq> ] *JH o<seq> ]
```

取消子查询嵌套

利用 SQL 语言提供的方便语法，可以将语句表达为嵌套查询。但是，由于 SQL Anywhere 可以更为有效地利用子查询的 WHERE 子句中的高选择性条件，因此将嵌套查询重写为连接通常会提高执行和优化的效率。一般情况下，会始终对 FROM 子句中最多包含一个表的相关子查询（用于 ANY、ALL 和 EXISTS 谓语中）执行取消子查询嵌套。如果根据查询语义可以确定子查询最多返回一行，则不相关子查询或在 FROM 子句中有多个表的子查询将被展平。

示例

对于外部块中的每个行，以下示例中的子查询最多可以匹配一行。由于该子查询最多可以匹配一行，SQL Anywhere 将发现可以将其转换为内连接。

```
SELECT s.*
FROM SalesOrderItems s
WHERE EXISTS
    ( SELECT *
      FROM Products p
      WHERE s.ProductID = p.ID
         AND p.ID = 300 AND p.Quantity > 20);
```

转换之后，这一相同语句将使用连接语法在内部表达：

```
SELECT s.*
FROM Products p JOIN SalesOrderItems s
   ON p.ID = s.ProductID
WHERE p.ID = 300 AND p.Quantity > 20;
```

```
p<Products> JNL s<FK_ProductID_ID>
```

同样，以下查询在子查询中包含一个连接性 EXISTS 谓语。该子查询可以匹配多个行。

```
SELECT p.*
FROM Products p
WHERE EXISTS
    ( SELECT *
      FROM SalesOrderItems s
      WHERE s.ProductID = p.ID
         AND s.ID = 2001);
```

SQL Anywhere 在 SELECT 列表中使用 DISTINCT，将该查询转换为内连接。

```
SELECT DISTINCT p.*
FROM Products p JOIN SalesOrderItems s
   ON p.ID = s.ProductID
WHERE s.ID = 2001;
```

```
Work[ DistH[ s<FK_ID_ID> JNL p<Products> ] ]
```

对于外部块中的每一行，如果该子查询最多可以匹配一行，则 SQL Anywhere 也可以在比较中排除子查询。下面的查询中就存在这种情况。

```
SELECT *
FROM Products p
WHERE p.ID =
    ( SELECT s.ProductID
      FROM SalesOrderItems s
      WHERE s.ID = 2001
         AND s.LineID = 1 );
```

SQL Anywhere 以如下方式重写该查询：

```
SELECT p.*
FROM Products p, SalesOrderItems s
WHERE p.ID = s.ProductID
   AND s.ID = 2001
   AND s.LineID = 1;
```

```
s<SalesOrderItems> JNL p<Products>
```

在执行取消子查询嵌套的重写优化时，会将 DUMMY 表视为一个特殊的表。即使子查询不相关，也始终会在形式为 SELECT expression FROM DUMMY 的子查询上执行子查询展平。

将谓语推入执行了 UNION 或 GROUP 的视图和派生表中

查询经常会限制视图的结果，使其只返回少数几个记录。在视图包含 GROUP BY 或 UNION 的情况下，最好让数据库服务器只计算所需行的结果。当且仅当谓语独占引用单个视图或派生表中的列时，才对其执行谓语推入。例如，不会将连接谓语推入视图中。

示例

假定您按如下方式定义了 ProductSummary 视图：

```
CREATE VIEW ProductSummary( ID,
        NumberOfOrders,
        TotalQuantity) AS
SELECT ProductID, COUNT( * ), sum( Quantity )
FROM SalesOrderItems
GROUP BY ProductID;
```

对于所订购的每一件产品，ProductSummary 视图都将返回包含该产品的订单数以及所有订单订购的产品总数。现在假定此视图上有以下查询：

```
SELECT *
FROM ProductSummary
WHERE ID = 300;
```

该查询将输出限制为只包括其 ID 列中的值为 300 的行。该查询以及视图定义中的查询可以合并为以下在语义上等效的 SELECT 语句：

```
SELECT ProductID, COUNT( * ), SUM( Quantity )
FROM SalesOrderItems
GROUP BY ProductID
HAVING ProductID = 300;
```

此查询不复杂的一项执行计划涉及计算每一件产品的集合，然后将结果限制为产品 ID 为 300 的一行。但是，由于 ProductID 列是分组列，可以将该列上的 HAVING 谓语推入查询的 WHERE 子句，从而生成以下语句：

```
SELECT ProductID, COUNT( * ), SUM( Quantity )
FROM SalesOrderItems
WHERE ProductID = 300
GROUP BY ProductID;
```

此 SELECT 语句会显著减少所需的计算。如果该谓语具有足够的选择性，则优化程序现在可以使用 ProductID 上的索引只检索产品 ID 为 300 的那些行，而不用按顺序扫描 SalesOrderItems 表。

相同的优化也用于包含 UNION 或 UNION ALL 的视图。

优化 OR 和 IN 列表谓语

为了利用索引列上的 IN 谓语，优化程序支持一种特殊的优化。这种优化也同样适用于相同索引列上用 OR 连在一起的多个谓语，因为这两种查询在语义上是等效的。若要能够进行这种优化，IN 列表必须只包含常量或在查询块的一次执行中为常量的值（例如外部引用）。

当优化程序遇到限定的 IN 列表谓语，并且该 IN 列表谓语所具有的选择性足以考虑索引检索时，优化程序就会将该 IN 列表谓语转换为嵌套循环连接。以下示例将说明这种优化的工作原理。

假定有以下查询，它列出了两个销售代表的所有订单：

```
SELECT *
FROM SalesOrders
WHERE SalesRepresentative = 902 OR SalesRepresentative = 195;
```

该查询在语义上等效于：

```
SELECT *
FROM SalesOrders
WHERE SalesRepresentative IN (195, 902);
```

优化程序估计到该 IN 列表谓语的组合选择性足够低，以便保证索引检索。因此，优化程序会将该 IN 列表视为虚拟表，并基于 SalesRepresentative 属性将该虚拟表连接到 SalesOrders 表。虽然这种优化的实际效果是在访问计划中包含附加的连接，但查询的连接度不会提高，因此优化时间不会受到影响。

这种优化具有两项主要的优点。首先，可以将 IN 列表当作 sargable 谓语并利用它来进行索引检索。其次，优化程序可以将 IN 列表排序，以匹配索引的排序序列，从而使检索更为有效。

以上查询的访问计划的简单形式为：

```
SalesOrders<FK_SalesRepresentative_EmployeeID>
```

另请参见

- “InList 算法 (IN)”一节第 567 页

优化 LIKE 谓语

常见的 LIKE 谓语通常包含作为文字常量或主机变量的模式。根据 LIKE 谓语包含的模式，优化程序可能会完全重写该 LIKE 谓语，或者用附加的条件（它们可用来对相应表执行索引检索）扩大该 LIKE 谓语。LIKE 谓语的附加条件使用 LIKE_PREFIX 谓语，这不能直接在查询中指定，但当查询优化程序可以应用优化时会出现在详细计划和图形式计划中。

示例

在下面的每一个示例中，假定 LIKE 谓语中的模式是文字常量或主机变量，X 是基表中的列：

- `X LIKE '%'` 被重写为 `X IS NOT NULL`。
- `X LIKE 'abc%'` 使用属于可优化搜索谓语的 LIKE_PREFIX 谓语（其可用于索引检索）来扩大并强制实施任何 X 值必须以字符 abc 开头这一条件。LIKE_PREFIX 强制执行具有多字节字符集和填补空白的数据库的正确语义。

将外部连接转换为内部连接

优化程序为其访问计划生成左深 (left-deep) 处理树。这一规则的唯一例外是存在右深 (right-deep) 嵌套外部连接表达式的情况。查询执行引擎的用于计算 LEFT OUTER JOIN 或 RIGHT OUTER JOIN 的算法要求，保留的表必须在任何连接策略中先于提供空值的表。因此，只要可能，优化程序就会寻找机会将 LEFT 或 RIGHT 外连接转换为内连接，这是因为内连接是可交换的，它使优化程序在执行连接枚举时具有更大的自由度。

如果以下条件之一为真，则 LEFT 或 RIGHT 外连接会转换为内连接：

- 不允许使用空值的谓语出现在查询的 WHERE 子句中，该谓语引用提供空值的表的列。由于该谓语不允许空值，因此将从结果中排除外连接生成的所有值为 NULL 的行，从而使该查询在语义上等效于内连接。
- 外连接的提供空值的一侧为保留一侧的每行只返回一行。如果该条件为真，则不存在提供空值的行，并且外连接等效于内连接。

当查询引用使用外连接编写的一个或多个视图时，此重写优化可应用于外连接查询。该查询的 WHERE 子句可以包括限制输出的条件，以便从一个或多个表的表达式中排除所有提供空值的行，从而使这种优化适用。

示例 1

对于下面的查询，针对 SalesOrderItems 表的每一行，均恰好存在与 Products 表匹配的一行，因为 ProductID 列声明为非 NULL，并且 SalesOrderItems 表具有以下外键：
`"FK_ProductID_ID" ("ProductID") REFERENCING "Products" ("ID").`

以下 SELECT 语句显示执行重写优化后如何重写查询：

```
SELECT * FROM SalesOrderItems s LEFT OUTER JOIN Products p ON (p.ID =
s.ProductID);
SELECT * FROM SalesOrderItems s JOIN Products p ON (p.ID = s.ProductID);
```

示例 2

以下查询列出产品及其数量较大的相应订单；LEFT OUTER JOIN 确保列出所有产品，即使产品没有订单：

```
SELECT *
FROM Products p KEY LEFT OUTER JOIN SalesOrderItems s
WHERE s.Quantity > 15;
```

该查询的问题在于，由于谓语 `s.Quantity > 15` 将在 `s.Quantity` 为 NULL 的情况下被解释为 FALSE，因此 WHERE 子句中的谓语将从结果中排除所有没有订单的产品。该查询在语义上等效于：

```
SELECT *
FROM Products p KEY JOIN SalesOrderItems s
WHERE s.Quantity > 15;
```

这种重写形式即为数据库服务器优化的查询。

在此示例中，该查询的编写几乎毫无疑问地存在错误；它应该改为：

```
SELECT *
FROM Products p
```



```
    KEY LEFT OUTER JOIN SalesOrderItems s
          ON s.Quantity > 15;
```

这样，数量的测试就是外连接条件的一部分。您可以将一些没有订单的新产品插入 Products 表中，然后再次执行这两个查询，这样就可以发现它们的区别。

```
INSERT INTO Products
SELECT ID + 10, Name, Description,
        'Extra large', Color, 50, UnitPrice, Photo
FROM Products
WHERE Name = 'Tee Shirt';
```

排除不必要的内连接和外连接

连接排除重写优化会通过从查询中排除表（如果这样做足够安全）来降低查询的连接度。通常，此优化应用于定义为主键到外键连接或主键到主键连接的内连接。连接排除优化也可以应用于外连接中使用的表（虽然使这种优化有效的条件要复杂得多）。

此优化不排除可使用 UPDATE 或 DELETE WHERE CURRENT 更新的表（即使这样做是正确的）。这会对查询的性能造成负面影响。然而，如果查询为只读，则可以在 SELECT 语句中指定 FOR READ ONLY，以确保执行连接排除。注意，尽管主查询块中的表是可更新的，但在子查询或嵌套派生表中出现的表是固有不可更新的。

简而言之，该重写优化适用于以下三种主要连接：

- 连接属于主键到外键连接，且在查询中只引用主表中的主键列。在这种情况下，如果主键表是不可更新的，则它会被排除。
- 连接属于同一表的两个实例之间的主键到主键连接。在这种情况下，如果其中一个表是不可更新的，则它会被排除。
- 连接属于外连接，且提供空值的表表达式具有以下属性：
  - 对于外连接的保留一侧的每一行，提供空值的表表达式最多返回一行。
  - 在外连接以外的其余查询中，不需要任何由提供空值的表表达式所生成的表达式。

示例

例如，在下面的查询中，连接属于主键到外键的连接，主键表、Products 表可以被排除：

```
SELECT s.ID, s.LineID, p.ID
FROM SalesOrderItems s KEY JOIN Products p
FOR READ ONLY;
```

该查询将被重写为：

```
SELECT s.ID, s.LineID, s.ProductID
FROM SalesOrderItems s
WHERE s.ProductID IS NOT NULL
FOR READ ONLY;
```

由于 SalesOrderItems 表中具有引用 Products 表的 NULL 外键的任何行都不会出现在结果中，因此第二个查询在语义上与第一个查询等效。

在以下查询中，如果提供空值的表表达式不能为保留一侧的任何行生成多个行，且在 LEFT OUTER JOIN 上没有使用 Products 表中的任何列，则 OUTER JOIN 可以被排除。

```
SELECT s.ID, s.LineID
FROM SalesOrderItems s LEFT OUTER JOIN Products p ON p.ID = s.ProductID
WHERE s.Quantity > 5
FOR READ ONLY;
```

该查询将被重写为：

```
SELECT s.ID, s.LineID
FROM SalesOrderItems s
WHERE s.Quantity > 5
FOR READ ONLY;
```

通过谓语推导发现可利用的条件

任何查询的有效访问策略实际上都依赖于 WHERE、ON 和 HAVING 子句中是否存在可优化搜索条件。只有通过将 sargable 条件作为匹配谓语利用，才有可能进行索引检索。此外，只有当存在等值连接条件时，才能使用散列、合并和块嵌套循环连接。由于上述原因，SQL Anywhere 会对初始查询文本中的搜索条件进行详细的分析，以发现可供优化程序利用的简化或隐含条件。

作为一个预处理步骤，视图扩展和合并一旦发生，就会对初始语句中的谓语进行一些简化。例如：

- 如果 X 可为空，`X = X` 会被重写为 `X IS NOT NULL`；否则将排除该谓语。
- `ISNULL(X,X)` 被重写为 `X`。
- 如果 X 为数字列，`X+0` 会被重写为 `X`。
- 将排除 `AND 1=1`。
- 将排除 `OR 1=0`。
- 由单个元素组成的 IN 列表谓语会被转换为简单的相等条件。

在这一预处理步骤之后，SQL Anywhere 会尝试将初始搜索条件规范化为连接性规范形式 (CNF)。若要使表达式成为 CNF 的形式，该表达式中的每一项都必须用 AND 连起来。每一项都由单个原子条件或一组用 OR 连起来的条件组成。

如果将任意条件转换为 CNF，所生成的表达式可能具有相似的复杂性，但会包含大得多的条件集。SQL Anywhere 将发现这种情况，并避免单纯地将条件转换为 CNF。相反，SQL Anywhere 会分析初始表达式以查找初始搜索条件所隐含的可利用谓语，并且用 AND 将这些推导出的条件与查询连起来。如果完全的规范化需要复制占用大量资源的谓语（例如限定子查询谓语），则还将避免完全的规范化。但是，只要可行，该算法就会将 IN 列表谓语合并起来。

将搜索条件完全规范化或找到可利用的条件后，优化程序会执行传递性分析，以发现传递等同条件（主要是传递连接条件和包含常量的条件）。这样，在基于开销的优化阶段中执行连接枚举时，优化程序将具有更大的自由度，因为这些传递条件允许附加的替代连接顺序。

示例

假定初始查询如下：


```
SELECT e.Surname, s.ID, s.OrderDate
FROM SalesOrders s, Employees e
WHERE
  ( e.EmployeeID = s.SalesRepresentative AND
    ( s.SalesRepresentative = 142 OR
      s.SalesRepresentative = 1596 )
  ) OR (
     e.EmployeeID = s.SalesRepresentative AND
     s.CustomerID = 667 );
```

该查询不包含连接性等值连接条件；如果不进行详细的谓语分析，优化程序将无法发现有效的访问计划。幸运的是，SQL Anywhere 能够将整个表达式转换为 CNF，从而生成等效的查询：

```
SELECT e.Surname, s.ID, s.OrderDate
FROM SalesOrders s, Employees e
WHERE
   e.EmployeeID = s.SalesRepresentative AND
   ( s.SalesRepresentative = 142 OR
     s.SalesRepresentative = 1596 OR
     s.CustomerID = 667 );
```

该查询现在可以作为内连接查询进行有效的优化。

排除不必要的大小写转换

缺省情况下，SQL Anywhere 数据库支持不区分大小写的字符串比较。有时，优化程序可能会遇到这样的查询：用户使用 UPPER、UCASE、LOWER 或 LCASE 内置函数在这些查询中显式强制进行不必要的文本转换。如果数据库的归类序列允许，SQL Anywhere 将自动排除这种不必要的转换。在谓语中排除这些大小写转换还有一个好处，就是将这些谓语的其中一部分转换为可优化搜索谓语，这种谓语可用于相应表的索引检索。

示例

请看以下查询：

```
SELECT *
FROM Customers
WHERE UPPER(Surname) = 'SMITH';
```

对于不区分大小写的数据库，该查询将在内部重写为如下查询，以便优化程序可以考虑使用 Customers.Surname 上的索引：

```
SELECT *
FROM Customers
WHERE Surname = 'SMITH';
```

将子查询重写为 EXISTS 谓语

作为 SQL Anywhere 设计基础的假定要求其节约内存并且在缺省情况下尽可能快地返回游标的前几项结果。为了与这些目标一致，SQL Anywhere 会在重写后语义正确的情况下，将所有集合操作子查询（如 IN、ANY 或 SOME 谓语）重写为 EXISTS 或 NOT EXISTS 谓语。这样，SQL Anywhere 就可以避免创建不必要的工作表，并且可以更容易地找到用于对表进行访问的合适索引。

不相关和相关子查询

不相关子查询是这样的子查询：其不包含对查询中其它更高级别部分中的表的显式引用。

下面是一个包含不相关子查询的普通查询。它选择在 2001 年 1 月 1 日没有下订单的所有客户的相关信息。

```
SELECT *
FROM Customers c
WHERE c.ID NOT IN
   (  SELECT o.CustomerID
      FROM SalesOrders o
      WHERE o.OrderDate = '2001-01-01' );
```

计算此查询的一种可能方式是先为 SalesOrder 表中所有在 2001 年 1 月 1 日下订单的客户创建一个工作表，然后查询 Customers 表并为该工作表列出的每位客户抽取一行。

但是，SQL Anywhere 会避免将结果实现为工作表。它还将优先考虑最快返回前几行结果的计划。因此，优化程序会使用 NOT EXISTS 谓语重写此类查询。这样，子查询就变成**相关**子查询了：现在，子查询包含对 Customers 表的 ID 列的显式外部引用。

```
SELECT *
FROM Customers c
WHERE NOT EXISTS
   (  SELECT *
      FROM SalesOrders o
      WHERE o.OrderDate = '2000-01-01'
         AND o.CustomerID = c.ID );
```

该查询在语义上等效于上面的查询，但用这种新语法表示时，会显示出多个优点：

1. 优化程序可以选择使用 SalesOrders 表的 CustomerID 属性或 OrderDate 属性上的索引。然而，在 SQL Anywhere 示例数据库中，只有 ID 和 CustomerID 列为索引列。
2. 优化程序可以选择计算子查询，而不将中间结果实现为工作表。
3. 数据库服务器可以在执行过程中高速缓存相关子查询的结果。这样就可以对外部引用 c.ID 的相同值重新使用此谓语的先前计算值。对于上述查询，高速缓存不起作用，因为客户标识号在 Customers 表中是唯一的。因此，始终使用外部引用 c.ID 的不同值来计算子查询。

有关子查询高速缓存的详细信息，请参见“子查询和函数高速缓存”一节第 564 页。

另请参见

- “相关和不相关子查询”一节第 475 页

内置用户定义的函数

当作为查询的一部分调用简单用户定义的函数时，有时会内置这些函数。即，查询会被重写以等同于初始查询，但却没有函数定义。决不会内置临时函数、递归函数和带有 NOT DETERMINISTIC 子句的函数。另外，如果通过子查询将某个函数作为参数调用，或从临时过程内部调用该函数时，不会内置该函数。

如果用户定义的函数采用以下形式之一，则可以内置这些函数：

- 包含一个 RETURN 语句的函数。例如：

```
CREATE FUNCTION F1( arg1 INT, arg2 INT )
RETURNS INT
BEGIN
 RETURN arg1 * arg2
END;
```

- 声明一个变量、指派该变量并返回一个值的函数。例如：

```
CREATE FUNCTION F2( arg1 INT )
RETURNS INT
BEGIN
 DECLARE result INT;
 SET result = ( SELECT ManagerID FROM Employees WHERE EmployeeID=arg1 );
 RETURN result;
END;
```

- 声明一个变量、选入该变量并返回一个值的函数。例如：

```
CREATE FUNCTION F3( arg1 INT )
RETURNS INT
BEGIN
 DECLARE result INT;
 SELECT ManagerID INTO result FROM Employees e1 WHERE EmployeeID=arg1;
 RETURN result;
END;
```

通过以下步骤内置用户定义的函数：复制用户定义的函数主体，通过调用插入参数，然后插入适当的 CAST 函数以确保查询的重写形式等效于原始形式。例如，假设您创建了一个函数（类似于先前定义的函数 F1），然后在查询的 FROM 子句中调用该过程，如下所示：

```
SELECT F1( e.EmployeeID, 2.5 ) FROM Employees e;
```

数据库服务器可能按如下方式重写该查询：

```
SELECT CAST( e.EmployeeID AS INT ) * CAST( 2.5 AS INT ) FROM Employees e;
```

另请参见

- “用户定义的函数简介”一节第 786 页
- “CREATE FUNCTION 语句（Web 服务）”一节 《SQL Anywhere 服务器 - SQL 参考》
- “CAST 函数 [Data type conversion]”一节 《SQL Anywhere 服务器 - SQL 参考》
- “SELECT 语句”一节 《SQL Anywhere 服务器 - SQL 参考》

内置简单的系统过程

在查询的 FROM 子句中调用系统过程时，如果该过程仅定义为主体中的一条 SELECT 语句，则有时会内置该过程。即，查询会被重写以等同于初始查询，但却没有过程定义。内置该过程时，它会被重写为派生表。如果过程使用缺省参数，或它在主体中包含一条 SELECT 语句之外的任何内容，则不会内置该过程。

例如，假定您创建了以下过程：

```
CREATE PROCEDURE Test1( arg1 INT )
 BEGIN
```

```
  SELECT * FROM Employees WHERE EmployeeID=arg1
END;
```

现在假定您在查询的 FROM 子句中调用该过程，如下所示：

```
SELECT * FROM Test1( 200 );
```

数据库服务器可能按如下方式重写该查询：

```
SELECT * FROM ( SELECT * FROM Employees WHERE EmployeeID=CAST( 200 AS INT ) )
AS Test1;
```

另请参见

- “用户定义的函数简介”一节第 786 页
- “CREATE FUNCTION 语句（Web 服务）”一节 《SQL Anywhere 服务器 - SQL 参考》
- “SELECT 语句”一节 《SQL Anywhere 服务器 - SQL 参考》

优化程序的工作原理

优化程序的作用是设计一种有效的方式来执行 SQL 语句。为此，优化程序必须为查询确定一个执行计划。这包括确定查询中引用的表的访问顺序、确定用于每个表的连接运算符和访问方法以及确定查询中未引用的实例化视图是否可以用于部分查询的计算。当为查询生成了可能的访问计划并预估了这些计划的开销后，优化程序将在连接枚举阶段尝试选取最佳计划来执行查询。最佳访问计划是指优化程序估计其将在最短的时间内以最低的开销返回所需结果集的访问计划。优化程序将通过估计所需的磁盘读写次数来确定每个枚举策略的开销。

在 Interactive SQL 中，可以通过单击 [**结果**] 窗格中的 [**计划**] 选项卡来查看用于执行查询的最佳访问计划。要更改所显示的细节级别，请更改 [**选项**] 窗口（通过 [**工具**] 菜单访问）的 [**计划**] 选项卡上的设置。请参见“读取图形式计划”一节第 571 页和“读取执行计划”一节第 569 页。

使返回第一行的开销最小化

优化程序使用通用磁盘访问开销模型来区分对数据库文件的随机检索和顺序检索之间存在的相对性能差异。可以利用 ALTER DATABASE 语句为特定的硬件配置校准数据库。请参见“ALTER DATABASE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

缺省情况下，查询处理的优化目标是返回完整的结果集。您可以使用 optimization_goal 选项对此缺省行为进行更改，以最大程度地降低快速返回第一行所需的开销。注意，如果将选项设置为 First-row，优化程序会选择旨在减少提取查询结果的第一行所需时间的访问计划，而且很可能以牺牲总检索时间为代价。请参见“optimization_goal 选项 [数据库]”一节 《SQL Anywhere 服务器 - 数据库管理》。

使用语义上等效的语法

大多数命令都可以使用 SQL 语言以多种不同的形式来表示。这些表达式在语义上等效（因为它们执行相同的任务），但在语法上却具有明显的差异。优化程序仅根据每个语句的语义设计出合适的访问计划，几乎毫无例外。

语法差异虽然看起来可能很明显，但通常不会造成任何影响。例如，谓语、表和属性在查询语法中的顺序差异并不会影响访问计划的选择。而优化程序也不会因为查询是否包含非实例化视图而受到影响。

减少优化程序查询的开销

在理想情况下，优化程序将尽可能地找出最高效的访问计划，但这一目标通常是不切实际的。对于复杂的查询，可能会存在多种可能性。

无论优化程序具有多高的效率，如果对每一种选择都进行分析，则会消耗一定的时间和资源。优化程序会将进一步优化所需的开销和执行到当前为止所找到的最佳计划所需的开销进行比较。如果优化程序已设计出了开销较低的计划，就会停止下来执行该计划。与执行已找到的访问计划相比，进一步的优化可能会消耗更多的资源。可以将 optimization_level 选项设置为一个高值来控制优化程序所消耗的资源量。请参见“optimization_level 选项 [数据库]”一节 《SQL Anywhere 服务器 - 数据库管理》。

当查询开销大并且非常复杂，或者优化级别设置的高时，优化程序就会运行较长的时间。如果查询开销庞大，则优化程序可能会运行很长的时间，从而导致出现明显的延迟。

优化程序估计值和列统计信息

优化程序会根据存储在数据库中的**列统计信息**和**启发式算法**（有根据的推测）来选择语句处理策略。对于优化程序所考虑的每个访问计划，都必须计算出一个估计的结果大小（行数）。例如，对基于查询中所用谓语的选择性估计的每个连接方法或索引访问，都会计算出一个估计的结果大小。这些估计的结果大小用于计算在计划中所使用的每个运算符（例如，JOIN 方法、GROUP BY 方法或顺序扫描）的预计磁盘访问开销和 CPU 开销。列统计信息是优化程序用于计算谓语的选择性估计值的主要数据。因此，它们对于正确估计访问计划的开销至关重要。

如果列统计信息失效或丢失，则性能会降低，因为不准确的统计信息可能导致执行计划效率低下。如果怀疑性能低下是由于不准确的列统计信息造成的，则应重新创建这些统计信息。请参见“更新列统计信息以提高优化程序性能”一节第 527 页。

优化程序使用列统计信息的方式

优化程序所使用的列统计信息的最重要的组成部分是**直方图**。直方图存储有关值在列中的分布情况的信息。在 SQL Anywhere 中，直方图表示列的数据分布的方式为：将列的域划分为一组连续的值域（也称为**桶**）；对于每个值域（即桶），记下表中有多少行的列值在桶内。

SQL Anywhere 会特别注意出现在表的大量行中的单一列值。重要的单值选择性保留在单个的直方图桶（例如，包括列域中的单个值的桶）中。SQL Anywhere 会尝试在每个直方图中保留最少数量的单个桶，通常是在 10 与 100 之间，具体取决于表的大小。此外，所有选择性大于 1% 的单个值均以单个桶的形式保留。因此，给定列的直方图会记下该列的前 *N* 个单值选择性，而值 *N* 取决于表的大小和大于 1% 的单值选择性的数量。

如果满足值域的最低数量，则低选择性频率一旦出现就会被高选择性频率替换。直方图在看到了足够多的选择性大于 1% 的值后，将只拥有多于最低数量的单值域。

有关列统计信息的详细信息，请参见“SYSCOLSTAT 系统视图”一节 《SQL Anywhere 服务器 - SQL 参考》。

优化程序使用启发式算法的方式

对于潜在执行计划中的每个表，优化程序估计到将形成部分结果的行数。该行数取决于表的大小以及查询的 WHERE 子句或 ON 子句中的限制。

给定某一列的直方图后，SQL Anywhere 会估计满足该列上的给定查询谓语的行数，其方法是累加与满足指定谓语的值相重叠的所有值域中的行数。对于直方图中部分包含在查询结果集中的值域，SQL Anywhere 将使用该值域内的内插值。

通常，优化程序会使用更为复杂的启发式算法。例如，只有在没有更合适的统计信息时，优化程序才使用缺省估计值。另外，优化程序还会利用索引和键来改进它对行数的推测。下面是一些单列的示例：

- 使某列等于某个值：当该列具有唯一索引或者为主键时估计一行。
- 索引列与常量的比较：探测索引以估计满足比较条件的行的百分比。

- 使某个外键等于某个主键（键连接）：使用关系表大小确定估计值。例如，如果一个有 5000 行的表具有一个外键，该外键引用一个有 1000 行的表，则优化程序会推测：对于每个主键行存在五个外键行。

另请参见

有关列值分布的信息，请参见：

- “ESTIMATE 函数 [Miscellaneous]”一节 《SQL Anywhere 服务器 - SQL 参考》
- “ESTIMATE_SOURCE 函数 [杂类]”一节 《SQL Anywhere 服务器 - SQL 参考》

优化程序使用过程统计信息的方式

与基表不同，在 FROM 子句中执行的过程调用没有列统计信息。因此，优化程序为来自过程调用的数据使用所有选择性估计值的缺省值或推测值。过程调用的执行时间及其结果集中的总行数将使用从先前调用中收集的统计信息来估计。这些统计信息通过 ProCall 算法在 ISYSPROCEDURE 系统表的统计信息列中进行维护。请参见“SYSPROCEDURE 系统视图”一节 《SQL Anywhere 服务器 - SQL 参考》和“ProcCall 算法 (PC)”一节第 567 页。

另请参见

有关获取谓语选择性的信息，请参见：

- “sa_get_histogram 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》
- “直方图实用程序 (dbhist)”一节 《SQL Anywhere 服务器 - 数据库管理》

更新列统计信息以提高优化程序性能

列统计信息永久地存储在数据库的系统表 ISYSCOLSTAT 中。为持续提高优化程序的性能，数据库服务器会在处理任何 SELECT、INSERT、UPDATE 或 DELETE 语句时自动更新列统计信息。方法是：监控满足引用表或列的任何谓语的行数，将此行数与估计的行数进行比较，然后在必要时对现有统计信息进行更新。

如果可以使用更准确的列统计信息，优化程序就可以计算出更适合的估计值，并提高后继查询的性能。

可以使用数据库选项设置是否更新列统计信息。update_statistics 数据库选项控制是否在执行查询时更新列统计信息，而 collect_statistics_on_dml_updates 数据库选项控制是否在执行修改数据的 DML 语句（例如，LOAD、INSERT、DELETE 和 UPDATE）时更新统计信息。

如果您怀疑性能不佳是由于统计信息没有准确反映当前列值所致，则最好执行 CREATE STATISTICS 或 DROP STATISTICS 语句。CREATE STATISTICS 删除旧的统计信息并创建新的统计信息，而 DROP STATISTICS 只删除旧的统计信息。

执行 CREATE INDEX 语句时，将自动为索引创建统计信息。

执行 LOAD TABLE 语句时，将自动为表创建统计信息。

另请参见

- “SYSCOLSTAT 系统视图”一节 《SQL Anywhere 服务器 - SQL 参考》
- “DROP STATISTICS 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “CREATE STATISTICS 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “update_statistics 选项 [数据库]”一节 《SQL Anywhere 服务器 - 数据库管理》
- “collect_statistics_on_dml_updates 选项 [数据库]”一节 《SQL Anywhere 服务器 - 数据库管理》

自动性能调优

查询中最常见的约束之一是与列值相等。下面的示例测试 Sex 列的等同性。

```
SELECT *
FROM Employees
WHERE Sex = 'f';
```

查询经常会在第二次执行时以不同的方式进行优化。对于上述类型的约束，SQL Anywhere 将根据经验自动考虑值分布异常的列。数据库会永久地存储这一信息，直到您使用 DROP STATISTICS 命令将其显式删除为止。请注意，在该列上有谓语的后继查询可能会使数据库服务器在该列上重新创建一个直方图。请参见“更新列统计信息以提高优化程序性能”一节第 527 页。

优化程序的基础假定

SQL Anywhere 查询优化程序的设计方向和理念建立在多个假定之上。通过理解优化程序的决定，您可以提高您自己的应用程序的质量或性能。这些假定提供了将帮助您了解其余章节所含信息的上下文。

最少的管理工作

过去，要想使数据库服务器发挥出很高的性能，主要需要依赖于知识渊博、尽职尽责的数据库管理员。为了取得很好的数据库性能，数据库管理员需要花费大量的时间来调整各种各样的数据存储和性能控制参数。随着数据库数据的变化，这些控制参数经常需要不断地加以调整。

随着数据库的发展和变化，SQL Anywhere 不断积累经验并做出调整。每个查询都会更好地了解数据库中的数据分布。SQL Anywhere 将自动存储此信息并利用它优化将来的查询。

每个查询既参与提供这种内部知识，同时又受益于这种内部知识。每个用户都可以受益于 SQL Anywhere 通过执行其他用户的查询而获取的知识。

统计信息收集机制是数据库服务器中一个必不可少的部分，不需要任何外部机制。如果您发现它在某种情况下可能会有帮助，则可以为数据库服务器提供索引提示。这些提示可确保在优化过程中使用某些索引，从而替换优化程序基于选择性估计值做出的决定。如果您将这些提示编写成触发器或过程代码，则需要在任何适当的时候对这些提示进行更新。请参见“更新列统计信息以提高优化程序性能”一节第 527 页和“使用索引”一节第 66 页。

对第一行或整个结果集进行优化

使用 optimization_goal 选项，可以指定查询处理的优化目标是快速返回第一行还是尽可能降低返回整个结果集所耗费的开销（缺省行为）。请参见“optimization_goal 选项 [数据库]”一节 《SQL Anywhere 服务器 - 数据库管理》。

优化混合或 OLAP 负载

使用 optimization_workload 选项，可以指定是否应该针对通常将更新、删除或插入与查询同时执行（混合负载）的数据库来优化查询处理，或者指定数据库中的更新活动的主要形式是否是很少与查询同时执行的批处理式更新。

有关详细信息，请参见“optimization_workload 选项 [数据库]”一节 《SQL Anywhere 服务器 - 数据库管理》。

统计信息存在并正确

优化程序是自行调优的，它在内部存储所需的全部信息。ISYSCOLSTAT 系统表是数据分布和谓语选择性估计值的持久存储库。在完成每个查询时，SQL Anywhere 将使用在查询执行过程中收集到的统计信息更新 ISYSCOLSTAT。因此，所有后续查询将可以使用更准确的估计值。

优化程序非常依赖这些统计信息，因此它所生成的访问计划的质量在很大程度上取决于这些统计信息。如果您最近插入了大量的新行，这些统计信息可能会不再准确地描述相应的数据。您可能会发现后续查询的执行速度非常慢。

如果对数据做出了重大变更并且发现查询执行速度很慢，则最好执行 DROP STATISTICS 和/或 CREATE STATISTICS。请参见“更新列统计信息以提高优化程序性能”一节第 527 页。

索引可用于满足谓语的要求

通常，SQL Anywhere 可以借助于索引来计算搜索条件。利用索引可以加快优化程序的数据访问速度并减少从基表中读取和处理的信息量。例如，如果查询包含搜索条件 `WHERE column-name=value`，并且该列存在索引，则可以使用索引扫描以仅读取该表中满足搜索条件的那些行。

索引还可以在连接表时显著地提高性能。

只要有可能，优化程序便会尝试仅索引检索以满足查询。通过仅索引检索，数据库服务器仅使用索引中的数据以满足查询，而无需访问表中的行。

当没有索引可供优化程序使用时，将会执行开销很大的顺序表扫描。

优化程序会自动选择使用其确定的、将产生最佳性能的索引。但是，您也可以在查询中使用索引提示以指定希望优化程序使用的索引。如果不能使用任何指定的索引，则会返回错误。请注意，索引提示可导致性能降低，只能由经验丰富的用户进行尝试。请参见“FROM 子句”一节 《SQL Anywhere 服务器 - SQL 参考》。

使用 [索引顾问] 决定是否建议数据库使用附加索引。请参见“索引顾问”一节第 167 页。

另请参见

- “在查询中使用谓语”一节第 531 页

虚拟内存是稀有资源

操作系统和多个应用程序会频繁地共享计算机的内存。SQL Anywhere 将内存视为稀有资源。由于 SQL Anywhere 非常节约地使用内存，因此可以在小型计算机上运行。如果要在便携式计算机或旧式计算机上运行数据库，这种节约性就显得非常重要。

保留额外内存（例如用于保存游标的内容）的开销可能很大。如果高速缓存已满，则可能需要将一个或多个页写入磁盘，以便为新页留出空间。为了完成后续操作，可能需要重新读取某些页。

鉴于这种情况，SQL Anywhere 将较高的开销与需要附加高速缓存开销的执行计划相关联。这种开销会促使优化程序放弃选择使用工作表的计划。

另一方面，优化程序会在提高性能时非常谨慎地使用内存。例如，如果在处理查询的过程中需要重复使用子查询的结果，它就会高速缓存这些结果。

内存调控器

SQL Anywhere 数据库服务器使用高速缓存（也称为**缓冲池**）以临时在内存中存储（缓冲）数据库页的映像。这些页通常是表页和索引页，尽管还有若干其它类型的物理页存储在 SQL Anywhere 数据库中。除了这些页之外，数据库服务器还使用两个其它内存池的高速缓存。其中一个池是用于数据库服务器数据结构（例如表示连接、语句和游标的那些数据结构）的虚拟内存。第二个池由用作**查询内存**的虚拟存储的高速缓存页组成。

查询执行算法（例如散列连接和排序）需要内存来高效运行。SQL Anywhere 使用**内存调控器**来决定每个语句可用于查询执行的查询内存量。内存调控器负责为语句分配查询内存池以保证负载的高效执行。QueryMemPages 数据库服务器属性显示可用于分配的查询内存池中的页数。池容量设置为服务器的*最大*高速缓存大小（即，高速缓存大小上限，它可以由 -ch 服务器选项控制）的某一比例。QueryMemPercentOfCache 数据库服务器属性提供可以作为查询内存的最大高速缓存大小的比例，该比例为 50%。

内存调控器授予各个语句所选页数，该语句随后可将这些页数用于内存密集型查询处理算法。查询内存池中的内存仍然可以用于其它目的（例如缓冲表或索引页），直到查询处理算法使用这些页。使用查询内存的内存密集型查询处理算法包括所有基于散列的运算符（例如非重复散列、散列分组依据、散列连接）以及排序和窗口运算符。

当语句开始执行时，内存调控器使用优化程序的估计值来确定对该语句有用的内存是多少。该估计值会作为 QueryMemMaxUseful 出现在图形式计划中。语句的查询内存在该请求的访问计划中使用的特定内存密集型运算符之间进行分配。交换运算符之下的并行内存密集型运算符各自接收为其分配的查询内存。简单请求不会从大量内存中受益，但是如果有足够内存来保存全部所需行，则使用基于散列的运算符或排序的请求可更高效地运行。

要提高数据库服务器进程并发水平，需要该数据库服务器为每个附加并发任务或请求保留一些查询内存量，同时减少用于任意特定请求的内存量。此外，内存调控器会限制可以并发执行的内存密集型请求的数量。基于运行数据库服务器的计算机的性能特性选择该最大值，该限值通过服务器属性 QueryMemActiveMax 进行显示。内存调控器还会保留正在运行的并发内存密集型请求数的估计

值，该估计值以数据库服务器属性和性能监控器统计信息 QueryMemActiveEst 的形式提供。内存调控器使用该运行平均值决定如何指派查询内存池的内存。如果正在执行的内存密集型请求很少，则会为每个请求指派更多的内存。如果执行的请求很多，会为每个请求指派较少的内存以更加均匀地共享查询内存，并考虑估计的对每个请求有用的查询内存页数。

如果某个内存密集型语句开始执行，而正在执行的并发内存密集型请求的数量已达到最大，则进来的语句会等待现有请求之一释放其指派的内存。query_mem_timeout 数据库选项控制进来的请求等待内存授予的时间长短。缺省设置为 -1，此时请求等待数据库服务器定义的时间段。如果经等待后没有内存授予可用，则可能通过内存不足执行策略（如果针对该计划的内存密集型物理运算符存在该策略），使用少量内存执行该语句的访问计划，这可能会导致执行速度很慢。数据库服务器属性和性能监控器统计信息 QueryMemGrantWaiting 显示正在等待要授予的内存请求的当前请求数，QueryMemGrantWaited 显示请求在授予内存请求之前等待的总次数。

在图形式计划中，值 QueryMemNeedsGrant 显示内存调控器认为这是简单请求（无需内存授予）还是内存密集型请求（需要内存授予）。如果内存调控器将某个请求分类为不需要内存授予的请求，则该请求会立即开始执行。否则，该请求会请求使用一定比例的查询内存池中的内存。图形式计划值 QueryMemLikelyGrant 显示为执行请求可能授予该请求的页数的估计值。

另请参见

- QueryMemActiveMax 属性： “数据库服务器属性”一节 《SQL Anywhere 服务器 - 数据库管理》
- QueryMemPages 属性： “数据库服务器属性”一节 《SQL Anywhere 服务器 - 数据库管理》
- QueryMemPercentOfCache 属性： “数据库服务器属性”一节 《SQL Anywhere 服务器 - 数据库管理》
- “query_mem_timeout 选项 [数据库]”一节 《SQL Anywhere 服务器 - 数据库管理》
- “设置数据库服务器的进程并发水平”一节 《SQL Anywhere 服务器 - 数据库管理》
- “含统计信息的图形式计划”一节第 572 页
- “-ch 服务器选项”一节 《SQL Anywhere 服务器 - 数据库管理》

在查询中使用谓语

谓语是用逻辑运算符 AND 和 OR 连接在一起的条件表达式，可在 WHERE、HAVING 或 ON 子句中组成条件集合。在 SQL 中，求值结果为 UNKNOWN 的谓语将解释为 FALSE。

可以利用索引从表中检索行的谓语被称作**可优化搜索**谓语。此名称来自于短语 *search argument-able*。涉及将列与常量、其它列或表达式进行比较的谓语可能是可优化搜索谓语。

以下语句中的谓语是可优化搜索谓语。SQL Anywhere 可以使用 Employees 表的主索引有效地计算该谓语的值。

```
SELECT *
FROM Employees
WHERE Employees.EmployeeID = 102;
```

在计划中，这会显示为： `Employees<Employees>`

对比之下，以下谓语就不是可优化搜索谓语。虽然 EmployeeID 列被编入主索引，但由于结果中包含所有行（或除一行之外的所有行），因此使用此索引并不会加快计算速度。

```
SELECT *
FROM Employees
where Employees.EmployeeID <> 102;
```

在计划中，这会显示为： `Employees<seq>`

类似地，没有任何索引可以帮助搜索名字以字母 k 结尾的所有雇员。同样，计算此结果的唯一方法是分别检查每一行。

函数

通常，在列名上包含函数的谓语不是可优化搜索谓语。例如，在以下查询上将不使用索引：

```
SELECT *
FROM SalesOrders
WHERE YEAR ( OrderDate ) ='2000';
```

要避免使用函数，可重写查询以使其成为可优化搜索查询。例如，可以改写上述查询：

```
SELECT *
FROM SalesOrders
WHERE OrderDate > '1999-12-31'
AND OrderDate < '2001-01-01';
```

如果将函数值存储在计算列中并在该列上建立索引，则使用函数的查询就会成为可优化搜索查询。**计算列**是从表上其它列中获得值的列。例如，如果有一个名为 OrderDate 的列，它保存订单的日期，则可以创建一个名为 OrderYear 的计算列，该列保存从 OrderDate 列中抽取的年份值。

```
ALTER TABLE SalesOrders
ADD OrderYear INTEGER
COMPUTE ( YEAR( OrderDate ) );
```

然后，可以按普通方式添加列 OrderYear 的索引：

```
CREATE INDEX IDX_year
ON SalesOrders ( OrderYear );
```

如果随即执行以下语句，则数据库服务器会发现存在一个包含此信息的索引列，并使用该索引来回应查询。

```
SELECT * FROM SalesOrders
WHERE YEAR( OrderDate ) = '2000';
```

为了进行列替代，计算列的域必须与 COMPUTE 表达式的域相同。在上面的示例中，如果 `YEAR( OrderDate )` 返回的是字符串而不是整数，优化程序就不会替代表达式的计算列，并且索引 IDX_year 也不能用于检索所需的行。

有关计算列的详细信息，请参见“使用计算列”一节第 28 页。

示例

在下面的每个示例中，属性 *x* 和 *y* 分别是一个表中的两个列。属性 *z* 包含在一个单独的表中。假定这些属性中的每一个都有索引。

可优化搜索	非可优化搜索
x = 10	*x* < > 10
x IS NULL	*x* IS NOT NULL
x > 25	*x* = 4 OR *y* = 5
x = *z*	*x* = *y*
x IN (4, 5, 6)	*x* NOT IN (4, 5, 6)
x LIKE 'pat%'	*x* LIKE '%tern'
x = 20 - 2	*x* + 2 = 20

有时，谓语是否是可优化搜索谓语可能不是很明显。在这些情况下，您可能能够重写该谓语，使其成为可优化搜索谓语。对于每一个示例，都可以基于字母表中 u 紧接在 t 后这一事实来重写谓语 *x* LIKE 'pat%'：*x* >= 'pat' and *x* < 'pau'。采用这种形式时，属性 *x* 上的索引对于查找有限范围内的值非常有用。幸运的是，SQL Anywhere 会自动进行此特定转换。

用于在表上进行索引检索的可优化搜索谓语属于**匹配**谓语。WHERE 子句可以包含多个匹配谓语。最合适的谓语取决于连接策略。在考虑替代连接策略时，优化程序会重新计算它对匹配谓语做出的选择。请参见“通过谓语推导发现可利用的条件”一节第 520 页。

MIN 和 MAX 函数基于开销的优化

MIN/MAX 基于开销的优化旨在利用现有索引有效地计算包括 MAX 或 MIN 集合函数的简单集合查询的结果。这种优化的目标是，能够以利用索引只检索出少数几行的方式计算出结果。若要成为这种优化的候选对象，查询必须满足以下条件：

- 不能包含 GROUP BY 子句
- 必须位于单个表上
- 查询的 SELECT 列表中只能包含一个集合函数（MAX 或 MIN）

示例

为了说明此优化，假定 SalesOrderItems 表中存在名为 prod_qty (ShipDate ASC, Quantity ASC) 的索引。因此，以下查询

```
SELECT MIN( Quantity )
 FROM SalesOrderItems
 WHERE ShipDate = '2000-03-25';
```

会被在内部重写为

```
SELECT MAX( Quantity )
 FROM ( SELECT FIRST Quantity
        FROM SalesOrderItems
        WHERE ShipDate = '2000-03-25'
```

```
            AND Quantity IS NOT NULL
  ORDER BY ShipDate ASC, Quantity ASC ) AS s(Quantity);
```

在应用这种优化时，可能不会为集合查询生成 NULL_VALUE_ELIMINATED 警告。

重写后的查询的执行计划（简单形式）为：

```
GrByS[ RL[ SalesOrderItems<prod_qty> ] ]
```

计划高速缓存

通常，优化程序会在每次执行查询时为查询选择执行计划。通过在执行时进行优化，使得优化程序可以根据当前系统状态以及当前选择性估计值和基于主机变量值的估计值来选择计划。对于频繁执行的查询，查询优化的开销可能会超过执行时进行优化所带来的好处。为减少重复优化这些语句的开销，SQL Anywhere 服务器会为以下各项考虑高速缓存计划：

- 在存储过程、用户定义函数和触发器中执行的所有语句。
- 满足跳过优化条件的 SELECT、INSERT、UPDATE 或 DELETE 语句。请参见“查询处理阶段”一节第 510 页。

  对于 INSERT 语句，只有 INSERT...VALUES 语句能够高速缓存；INSERT...ON EXISTING 语句则不可高速缓存。

  对于 UPDATE 和 DELETE 语句，WHERE 子句必须存在，且必须包含使用主键标识行的搜索条件。如希望使用计划高速缓存，则不允许有其它搜索条件存在。另外，对于 UPDATE 语句，包含变量赋值的 SET 子句将使该语句不能够高速缓存。

当这些语句之一由某个连接执行多次后，优化程序将在不知道主机变量值的情况下为该语句构建一个可重用的计划。可重用的计划可能具有较高的开销，因为主机变量值不能用于选择性估计或语义查询转换。如果可重用的计划的结构与先前执行语句时所构建的计划的结构相同，则数据库服务器会将可重用的计划添加到计划高速缓存中。如果对每次执行进行优化的好处会超过不进行优化所节省的开销，则执行计划不会被高速缓存。

如果执行计划使用未被语句引用的实例化视图，且 materialized_view_optimization 选项设置为非 Stale 值，则不会高速缓存该执行计划，且会在下次调用存储过程、用户定义函数或触发器时再次优化该语句。

计划高速缓存是对用于执行访问计划的数据结构进行逐个连接的高速缓存。要重用高速缓存的计划，其步骤包括在高速缓存中查找计划并将其重置为初始状态。通常，这比在所有查询处理阶段中处理语句要快得多。如果高速缓存的计划未被频繁使用，并且不会提高高速缓存使用率，它们就可能被存储到磁盘中。优化程序会定期重新优化查询，以检验高速缓存的计划是否仍然具有较高的效率。

要高速缓存的最大计划数目由 max_plans_cached 选项指定。缺省值为 20。若要禁用计划高速缓存，请将此选项设置为 0。请参见“max_plans_cached 选项 [数据库]”一节 《SQL Anywhere 服务器 - 数据库管理》。

可以使用 QueryCachedPlans 统计信息显示当前对多少个查询执行计划进行了高速缓存。可以使用 CONNECTION_PROPERTY 函数检索此属性，以显示为给定连接高速缓存了多少个查询执行计划，或者可以使用 DB_PROPERTY 函数来计算为所有连接高速缓存的执行计划的数目。此属性可以与 QueryCachePages、QueryOptimized、QueryBypassed 和 QueryReused 组合使用，以帮助确定

max_plans_cached 选项的最佳设置。请参见“连接属性”一节 《SQL Anywhere 服务器 - 数据库管理》。

可以使用数据库或连接属性 QueryCachePages 来确定用于高速缓存执行计划的页数。这些页占用临时文件中的空间，但不必驻留在内存中。

另请参见

- “跳过查询处理阶段的资格”一节第 511 页
- “使用实例化视图提高性能”一节第 536 页
- “materialized_view_optimization 选项 [数据库]”一节 《SQL Anywhere 服务器 - 数据库管理》
- “DB_PROPERTY 函数 [System]”一节 《SQL Anywhere 服务器 - SQL 参考》
- “CONNECTION_PROPERTY 函数 [System]”一节 《SQL Anywhere 服务器 - SQL 参考》

使用实例化视图提高性能

实例化视图是指这样一种视图：其结果集存储在磁盘上（这一点很像基表），但却是经过计算得出的（这一点很像视图）。从概念上讲，实例化视图既是视图（有查询说明）又是表（有已永久实现的行）。因此，许多对表执行的操作也可以对实例化视图执行。例如，可以在实例化视图上建立索引，也可以从实例化视图卸载。

定义实例化视图

在设计应用程序时，可考虑为频繁执行的开销巨大的查询或查询中开销巨大的部分（如涉及密集集合和连接操作的查询）定义实例化视图。实例化视图专用于在以下环境中提高性能：

- 数据库很大
- 频繁的查询导致对大量数据执行重复性集合和连接操作
- 对基础数据的更改相对很少
- 不严格要求访问即时数据

您不是一定要更改查询以从实例化视图中受益。例如，实例化视图非常适合于在其中的基础数据不经常更改的数据仓库应用程序中使用。

进行优化时，优化程序将保留实例化视图的列表，将这些视图视为部分或完全满足某个已提交查询的候选视图。如果优化程序发现实例化视图的某个候选视图可以满足全部或部分查询，则它会将该视图包含在为优化的枚举阶段生成的建议中，此阶段将基于开销确定最佳计划。优化程序用来将实例化视图与查询进行匹配的过程称为**视图匹配**。实例化视图必须满足一定条件，优化程序才能将其考虑在内。这意味着除非查询中显式引用了实例化视图，否则无法确保它会被优化程序使用。但是，您可以确保要考虑的视图均满足某些条件。

如果优化程序确定允许使用实例化视图，则会对每个候选实例化视图进行检查。如果实例化视图满足以下条件，则视图匹配算法会考虑使用该视图：

- 已启用实例化视图供数据库服务器使用。请参见“启用和禁用实例化视图”一节第 60 页。
- 已启用实例化视图以在优化中使用。请参见“允许和禁止优化程序使用实例化视图”一节第 62 页。
- 已经初始化实例化视图。请参见“初始化实例化视图”一节第 54 页。
- 实例化视图满足所有待考虑的优化程序要求。请参见“实例化视图和视图匹配算法”一节第 537 页。
- 用于创建实例化视图的一些重要选项的值与执行查询的连接选项的值匹配。请参见“实例化视图的限制”一节第 51 页。
- 实例化视图的上次刷新没有超过为数据库选项 materialized_view_optimization 设置的失效阈值。请参见“设置优化程序的实例化视图失效程度阈值”一节第 63 页。

如果实例化视图满足上述条件，并且也满足全部或部分查询，则视图匹配算法会将该实例化视图包括在为优化的枚举阶段生成的建议中，此阶段将基于开销找到最佳计划。然而，这并不表示最后会

在最终的执行计划中使用该实例化视图。例如，当估计另一访问计划（该计划未使用实例化视图）所需开销更低时，仍有可能不使用适用于计算查询结果的实例化视图。

确定候选实例化视图的列表

随时都可以通过执行以下命令来获取优化程序所考虑的候选实例化视图的列表：

```
SELECT * FROM sa_materialized_view_info( ) WHERE AvailForOptimization='Y';
```

返回的列表特定于发出请求的连接，因为优化程序在生成列表时已将帐户选项设置考虑在内。如果为连接指定的选项与创建实例化视图时使用的选项不匹配，则不会考虑将这一创建的实例化视图作为候选实例化视图。有关必须匹配的选项的列表，请参见“实例化视图的限制”一节第 51 页。

要获取因选项设置不匹配而没有被连接视为候选的所有实例化视图的列表，可从将要执行查询的连接执行以下命令：

```
SELECT * FROM sa_materialized_view_info( ) WHERE AvailForOptimization='O';
```

确定是否考虑实例化视图

通过在 Interactive SQL 中查看查询的图形式计划的 [**高级详细信息**] 窗口，可以看到用于特定查询的实例化视图的列表。请参见“读取执行计划”一节第 569 页。

通过查看由优化程序枚举的访问计划，也可以在 Sybase Central 中使用 [应用程序分析] 模式来确定查询的枚举阶段是否将实例化视图考虑在内。要查看优化程序所枚举的访问计划，跟踪功能必须打开且必须配置为包括 OPTIMIZATION_LOGGING 跟踪类型。有关此跟踪类型的详细信息，请参见“应用程序分析”一节第 161 页和“选择诊断跟踪级别”一节第 173 页。

有关优化的枚举阶段的详细信息，请参见“查询处理阶段”一节第 510 页。

注意
使用快照隔离时，优化程序不会将已在当前事务快照开始后刷新的实例化视图考虑在内。

实例化视图和视图匹配算法

视图匹配算法可确定实例化视图是否可以用于满足某个查询。该确定过程包括两个步骤：一个查询评估步骤和一个实例化视图评估步骤。

如果实例化视图的定义满足以下条件，则优化程序会将其包括在视图匹配算法将要检查的一组实例化视图中。

- 只包含一个查询块
- 只包含一个 FROM 子句

- 不包含任何以下构造或规范：
  - GROUPING SETS
  - CUBE
  - ROLLUP
  - 子查询
  - 派生表
  - UNION
  - EXCEPT
  - INTERSECT
  - 实例化视图
  - DISTINCT
  - TOP
  - FIRST
  - 自连接
  - 递归连接
  - FULL OUTER JOIN

如果 HAVING 子句不包含子选择或子查询，则实例化视图定义可以包含 GROUP BY 子句和 HAVING 子句。

注意
这些限制只适用于视图匹配算法所考虑的实例化视图。如果在查询中显式引用了某个实例化视图，则优化程序会像使用基表一样使用该视图。

另请参见

- “读取执行计划”一节第 569 页
- “查询处理阶段”一节第 510 页
- “应用程序分析”一节第 161 页

查询评估

在查询评估过程中，视图匹配算法将对查询进行检查。如果符合以下任一条件，则不使用实例化视图来处理查询。

- 查询所引用的所有表均可更新。

  优化程序不会为固有可更新或在可更新游标中显式声明的 SELECT 语句考虑实例化视图。如果使用 Interactive SQL（缺省情况下会为 SELECT 语句使用可更新的游标），就会发生这种情况。

- 语句属于使用优化程序跳过的简单 DML 语句，且经过启发式优化。但可以使用 OPTION 子句的 FORCE OPTIMIZATION 选项来强制对任何 SELECT 语句执行基于开销的优化。请参见“SELECT 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

- 查询的执行计划已经被高速缓存，包括在存储过程和用户定义函数中的查询就属于这种情况。数据库服务器可能会高速缓存这些查询的执行计划，以使它们可以被重新使用。对于这类查询，

查询执行计划将在执行之后进行高速缓存。下次执行此查询时，会对计划进行检索，执行阶段之前的所有阶段都会被跳过。请参见“计划高速缓存”一节第 534 页。

实例化视图评估

进行实例化视图评估时，会确定哪些现有实例化视图可用于计算全部或部分查询。

一旦某个实例化视图已经与部分查询匹配，则会决定是否在最终查询执行计划中使用该视图，该决定是基于开销做出的。枚举阶段的作用是生成其中包含由视图匹配算法建议的视图，并基于估计的计划开销选择一个可能包含也可能不包含某些实例化视图的最佳访问计划。

如果实例化视图被定义为分组选择项目连接查询（也称为**分组查询**或包含 GROUP BY 子句的查询），则视图匹配算法可以将其与分组的查询块匹配。如果实例化视图被定义为选择项目连接查询（即不是分组查询），则视图匹配算法可将其与任何类型的查询块匹配。

下文列出了一些条件，视图匹配算法在决定视图 V 是否与属于查询 Q 的查询块 QB 的一部分匹配时，需要利用这些条件。通常，视图 V 必须包含查询 QB 的表的子集。只有 V 的扩展表例外。V 的扩展表是指仅用一行与 V 的其余表连接的表。例如，如果主键表的唯一谓语是一个非空外键列与其主键列的等值连接，则该主键表即为一个扩展表。有关包含扩展表的实例化视图的示例，请参见“示例 2：匹配分组选择项目连接视图”一节第 543 页。

- 用于创建实例化视图 V 的选项值与执行查询的连接的选项值匹配。有关必须匹配的选项的列表，请参见“实例化视图的限制”一节第 51 页。
- 实例化视图 V 的上次刷新未超过 materialized_view_optimization 数据库选项或 SELECT 语句中 MATERIALIZED VIEW OPTIMIZATION 子句（如果指定）指定的失效阈值。请参见“设置优化程序的实例化视图失效程度阈值”一节第 63 页。
- V 中使用的所有表（V 的一些扩展表可能例外）均存在于 QB 中。QB 中的这组常用表在下文中将以 CT 表示。
- 在查询 Q 中，CT 中没有可更新的表。
- CT 中的所有表均属于 QB 中外连接的同一方（即，它们均属于外连接的保留一方或 QB 的外连接的空值提供方）。
- 可以确定 V 中的谓语包含 QB 中只引用 CT 的谓语的子集。换言之，V 中的谓语比 QB 中的谓语限制要少。QB 中与 V 中的谓语完全匹配的谓语称为**被匹配的谓语**。
- 任何没有在被匹配的谓语中使用的、引用 CT 中的表的 QB 表达式，必须出现在 V 的选择列表中。
- 如果 V 和 QB 均已进行分组，则除 CT 中的表之外 QB 不包含其它表。此外，V 的 GROUP BY 子句中的表达式集必须等于 QB 的 GROUP BY 子句中的表达式集或是其超集。
- 如果 V 和 QB 是在同一表达式集合上进行分组，则 QB 中的所有集合函数还必须在 V 中进行计算，或可以通过 V 的集合函数进行计算。例如，如果 QB 包含 AVG(x)，则 V 必须包含 AVG(x)，或其必须同时包含 SUM(x) 和 COUNT(x)。

- 如果 QB 的 GROUP BY 子句是 V 的 GROUP BY 子句的子集，则 V 的集合函数中必须包括 QB 的简单集合函数，而其组合集合函数必须通过 V 的简单集合函数进行计算。简单集合函数包括：

  - BIT_AND
  - BIT_OR
  - BIT_XOR
  - COUNT
  - LIST
  - MAX
  - MIN
  - SET_BITS
  - SUM
  - XMLAGG

  可通过简单集合函数计算的组合集合函数有：

  - SUM(x)
  - COUNT(x)
  - SUM(CAST(x AS DOUBLE))
  - SUM(CAST(x AS DOUBLE) * CAST(x AS DOUBLE))
  - VAR_SAMP(x)
  - VAR_POP(x)
  - VARIANCE(x)
  - STDDEV_SAMP(x)
  - STDDEV_POP(x)
  - STDDEV(x)

  下列统计集合函数：

  - COVAR_SAMP(y,x)
  - COVAR_POP(y,x)
  - CORR(y,x)
  - REGR_AVGX(y,x)
  - REGR_AVGY(y,x)
  - REGR_SLOPE(y,x)
  - REGR_INTERCEPT(y,x)
  - REGR_R2(y,x)
  - REGR_COUNT(y,x)
  - REGR_SXX(y,x)
  - REGR_SYY(y,x)
  - REGR_SXY(y,x)

可通过以下简单集合函数计算：

- SUM(y1)
- SUM(x1)
- COUNT(x1)
- COUNT(y1)
- SUM(x1*y1)
- SUM(y1*x1)
- SUM(x1*x1)
- SUM(y1*y1)

其中 x1 = CAST(IFNULL(x, x,y) AS DOUBLE))，y1 = CAST(IFNULL(y,y,x) AS DOUBLE)。

视图匹配算法示例

示例 1：匹配选择项目连接视图

如果查询频繁访问某个基表的某一部分，则最好定义一个实例化视图来存储此部分。例如，以下定义了一个实例化视图 V_Canada，该视图存储 Customer 表中居住在加拿大的所有客户。当在 State 列限定于某些值的情况下使用该实例化视图时，最好在实例化视图 V_Canada 的 State 列上创建索引 V_Canada_State。

```
CREATE MATERIALIZED VIEW V_Canada AS
 SELECT c.ID, c.City, c.State, c.CompanyName
  FROM Customers c
  WHERE c.State IN ( 'AB', 'BC', 'MB', 'NB', 'NL',
   'NT', 'NS', 'NU', 'ON', 'PE', 'QC', 'SK', 'YT' );
REFRESH MATERIALIZED VIEW V_Canada;
CREATE INDEX V_Canada_State on V_Canada( State );
```

任何仅需要居住在加拿大的客户的子集的查询块都可以从该实例化视图中受益。例如，下面的查询 1 将为每个公司计算在安大略的所有客户的产品总价，可以在其访问计划中使用实例化视图 V_Canada。查询 1 的使用实例化视图 V_Canada 的访问计划将提供一个有效的计划，就像查询 1 已被重写为查询 1_v 一样，这在语义上是等效的。注意，优化程序并不使用实例化视图重写该查询，而生成的、使用实例化视图的访问计划在理论上可以视为与重写的查询对应。

查询 1 的执行计划使用实例化视图 V_Canada，如下所示：

```
Work[ GrByH[ V_Canada<V_Canada_State> JNLO
SalesOrders<FK_CustomerID_ID> JNLO SalesOrderItems<FK_ID_ID> JNLO
Products<ProductsKey> ] ]
```

查询 1：

```
SELECT SUM( SalesOrderItems.Quantity
   * Products.UnitPrice ) AS Value
 FROM Customers c
  LEFT OUTER JOIN SalesOrders
   ON( SalesOrders.CustomerID = c.ID )
  LEFT OUTER JOIN SalesOrderItems
   ON( SalesOrderItems.ID = SalesOrders.ID )
  LEFT OUTER JOIN Products
   ON( Products.ID = SalesOrderItems.ProductID )
```

```
 WHERE c.State = 'ON'
 GROUP BY c.CompanyName;
```

查询 1_v：

```
SELECT SUM( SalesOrderItems.Quantity
   * Products.UnitPrice ) AS Value
  FROM V_Canada
   LEFT OUTER JOIN SalesOrders ON( SalesOrders.CustomerID = V_Canada.ID )
   LEFT OUTER JOIN SalesOrderItems ON( SalesOrderItems.ID = SalesOrders.ID )
   LEFT OUTER JOIN Products ON( Products.ID = SalesOrderItems.ProductID )
  WHERE V_Canada.State = 'ON'
  GROUP BY V_Canada.CompanyName;
```

查询 2 可以在主查询块和 HAVING 子查询中使用该视图。优化程序枚举的某些使用实例化视图 V_Canada 的访问计划会提供查询 2_v，其与查询 2 在语义上是等效的，其中用 V_Canada 视图替代了 Customer 表。

执行计划为：

```
Work[ GrByH[ V_Canada<V_Canada_State> JNLO
SalesOrders<FK_CustomerID_ID> JNLO SalesOrderItems<FK_ID_ID> JNLO
Products<ProductsKey> ] ] : GrByS[ V_Canada<seq> JNLO
SalesOrders<FK_CustomerID_ID> JNLO SalesOrderItems<FK_ID_ID> JNLO
Products<ProductsKey>
```

查询 2：

```
SELECT  SUM( SalesOrderItems.Quantity
  * Products.UnitPrice ) AS Value
 FROM Customers c
  LEFT OUTER JOIN SalesOrders
   ON( SalesOrders.CustomerID = c.ID )
  LEFT OUTER JOIN SalesOrderItems
   ON( SalesOrderItems.ID = SalesOrders.ID )
  LEFT OUTER JOIN Products
   ON( Products.ID = SalesOrderItems.ProductID )
 WHERE c.State = 'ON'
 GROUP BY CompanyName
 HAVING Value >
  ( SELECT AVG( SalesOrderItems.Quantity
        * Products.UnitPrice ) AS Value
      FROM Customers c1
       LEFT OUTER JOIN SalesOrders
       ON( SalesOrders.CustomerID = c1.ID )
       LEFT OUTER JOIN SalesOrderItems
       ON( SalesOrderItems.ID = SalesOrders.ID )
        LEFT OUTER JOIN Products
       ON( Products.ID = SalesOrderItems.ProductID )
       WHERE c1.State IN ('AB', 'BC', 'MB', 'NB', 'NL', 'NT', 'NS',
      'NU', 'ON', 'PE', 'QC', 'SK', 'YT' ) );
```

查询 2_v：

```
SELECT  SUM( SalesOrderItems.Quantity
         * Products.UnitPrice ) AS Value
 FROM V_Canada
 LEFT OUTER JOIN SalesOrders
  ON( SalesOrders.CustomerID=V_Canada.ID )
 LEFT OUTER JOIN SalesOrderItems
  ON( SalesOrderItems.ID=SalesOrders.ID )
 LEFT OUTER JOIN Products
  ON( Products.ID=SalesOrderItems.ProductID )
```



```
WHERE V_Canada.State = 'ON'
GROUP BY V_Canada.CompanyName
HAVING Value >
   ( SELECT AVG( SalesOrderItems.Quantity
         * Products.UnitPrice ) AS Value
       FROM V_Canada
        LEFT OUTER JOIN SalesOrders
        ON( SalesOrders.CustomerID = V_Canada.ID )
        LEFT OUTER JOIN SalesOrderItems
        ON( SalesOrderItems.ID = SalesOrders.ID )
         LEFT OUTER JOIN Products
        ON( Products.ID = SalesOrderItems.ProductID )
        WHERE V_Canada.State IN ('AB', 'BC', 'MB',
       'NB', 'NL', 'NT', 'NS', 'NU', 'ON', 'PE', 'QC',
       'SK', 'YT' ) );
```

示例 2：匹配分组选择项目连接视图

分组实例化视图可以使分组查询性能达到最佳。如果频繁执行的查询中使用了相似的集合，则应在这些查询中使用的 GROUP BY 子句的超集上定义实例化视图，以预先集合数据。查询的组合集合函数可从视图中使用的简单集合函数计算得出。因此，建议您在实例化视图中只存储简单集合函数。

下面的实例化视图 V_quantity 预先计算每年和每月每种产品的总和和计数。下面的查询 3 可使用该视图只选择 2000 年中的月份（简要计划为 `Work[ GrByH[ V_quantity<seq> ] ]`，相当于查询 3_v）。

下面的查询 4（未引用扩展表 SalesOrders）仍可使用 V_quantity，因为视图包含计算查询 4 所需的全部数据（简要计划为 `Work[ GrByH[ V_quantity<seq> ] ]`，对应于查询 4_v）。

```
CREATE MATERIALIZED VIEW V_Quantity AS
 SELECT s.ProductID,
  Month( o.OrderDate ) AS month,
  Year( o.OrderDate ) AS year,
  SUM( s.Quantity ) AS q_sum,
  COUNT( s.Quantity  ) AS q_count
 FROM SalesOrderItems s KEY JOIN SalesOrders o
 GROUP BY s.ProductID, Month( o.OrderDate ),
  Year( o.OrderDate );
REFRESH MATERIALIZED VIEW V_Quantity;
```

查询 3：

```
SELECT s.ProductID,
  Month( o.OrderDate ) AS month,
  AVG( s.Quantity) AS avg,
  SUM( s.Quantity ) AS q_sum,
  COUNT( s.Quantity ) AS q_count
 FROM SalesOrderItems s KEY JOIN SalesOrders o
 WHERE year( o.OrderDate ) = 2000
 GROUP BY s.ProductID, Month( o.OrderDate );
```

查询 3_v：

```
SELECT V_Quantity.ProductID,
  V_Quantity.month AS month,
  SUM( V_Quantity.q_sum ) / SUM( V_Quantity.q_count )
   AS avg,
  SUM( V_Quantity.q_sum ) AS q_sum,
  SUM( V_Quantity.q_count ) AS q_count
 FROM V_Quantity
```

```
 WHERE V_Quantity.year  = 2000
 GROUP BY V_Quantity.ProductID, V_Quantity.month;
```

查询 4：

```
SELECT s.ProductID,
  AVG( s.Quantity ) AS avg,
  SUM( s.Quantity ) AS sum
 FROM SalesOrderItems s
 WHERE s.ProductID IS NOT NULL
 GROUP BY s.ProductID;
```

查询 4_v

```
SELECT V_Quantity.ProductID,
  SUM( V_Quantity.q_sum ) / SUM( V_Quantity.q_count )
   AS avg,
  SUM( V_Quantity.q_sum ) AS sum
 FROM V_Quantity
 WHERE V_Quantity.ProductID IS NOT NULL
 GROUP BY V_Quantity.ProductID;
```

示例 3：匹配复杂查询

视图匹配算法会应用于每个查询块，因此每个查询块可以使用多个实例化视图，而且整个查询也可以使用多个实例化视图。下面的查询 5 可以使用以下三个实例化视图：V_Canada，用于 LEFT OUTER JOIN 的提供空值方；V_ship_date（下面进行了定义），用于主查询块的保留方；V_quantity，用于子查询块。查询 5_v 的执行计划为：

```
Work[ Window[ Sort[ V_ship_date<V_Ship_date_date> JNLO
( so<SalesOrdersKey> JH V_Canada<V_Canada_state> ) ] ] ] :
GrByS[V_quantity<seq> ].
```

```
CREATE MATERIALIZED VIEW V_ship_date AS
 SELECT s.ProductID, p.Description,
  s.Quantity, s.ShipDate, s.ID
 FROM SalesOrderItems s KEY JOIN Products p ON ( s.ProductId = p.ID )
 WHERE s.ShipDate >= '2000-01-01'
  AND s.ShipDate <= '2001-01-01';
REFRESH MATERIALIZED VIEW V_ship_date;
CREATE INDEX V_ship_date_date ON V_ship_date( ShipDate );
```

查询 5：

```
SELECT p.ID, p.Description, s.Quantity,
  s.ShipDate, so.CustomerID, c.CompanyName,
  SUM( s.Quantity ) OVER ( PARTITION BY s.ProductID
                      ORDER BY s.ShipDate
                      ROWS BETWEEN UNBOUNDED PRECEDING
                      AND CURRENT ROW ) AS cumulative_qty
 FROM SalesOrderItems s JOIN Products p
  ON (s.ProductID = p.ID) LEFT OUTER JOIN (
   SalesOrders so JOIN Customers c
    ON ( c.ID = so.CustomerID AND c.State = 'ON' ) )
  ON (s.ID = so.ID )
 WHERE s.ShipDate >= '2000-01-01'
   AND s.ShipDate <= '2000-12-31'
   AND s.Quantity  > ( SELECT  AVG( s.Quantity ) AS avg
                       FROM SalesOrderItems s KEY JOIN SalesOrders o
                       WHERE year( o.OrderDate) = 2000 )
 FOR READ ONLY;
```


查询 5_v：

```
SELECT  V_ship_date.ID, V_ship_date.Description,
  V_ship_date.Quantity, V_ship_date.ShipDate,
  so.CustomerID, V_Canada.CompanyName,
  SUM( V_ship_date.Quantity ) OVER ( PARTITION BY V_ship_date.ProductID
                                     ORDER BY V_ship_date.ShipDate
                                     ROWS BETWEEN UNBOUNDED PRECEDING
                                     AND CURRENT ROW ) AS cumulative_qty
 FROM V_ship_date
  LEFT OUTER JOIN ( SalesOrders so JOIN V_Canada
   ON ( V_Canada.ID = so.CustomerID AND V_Canada.State = 'ON' ) )
  ON ( V_ship_date.ID = so.ID )
 WHERE V_ship_date.ShipDate >= '2000-01-01'
  AND V_ship_date.ShipDate <= '2000-12-31'
  AND V_ship_date.Quantity >
    ( SELECT SUM( V_quantity.q_sum ) / SUM( V_quantity.q_count )
      FROM V_Quantity
      WHERE V_Quantity.year = 2000 )
 FOR READ ONLY;
```

示例 4：匹配带有 OUTER JOIN 的实例化视图

使用与仅带有内连接的视图类似的规则，视图匹配算法可匹配带有 OUTER JOIN 的视图和查询。只要提供空值方的所有表都是扩展表，则实例化视图的 OUTER JOIN 空值提供方可不在查询中出现。查询可包含内连接，以匹配视图的外连接。下面的查询 6_v、7_v、8_v 和 9_v 说明了如何使用在其定义中包含 OUTER JOIN 的实例化视图来回应查询。

下面的查询 6 完全匹配实例化视图 V_SalesOrderItems_2000，并可作为查询 6_v 来计算。

查询 7 包含一些额外的外连接保留方的谓语，并且仍然可以使用 V_SalesOrderItems_2000 进行计算。请注意，在视图 V_SalesOrderItems_2000 中，提供空值的表 Products 是扩展表。这意味着该视图也与查询 8 匹配，查询 8 不包含表 Products。

查询 9 只包含表 SalesOrderItems 和表 Products 的内连接，且通过仅选择视图 V_SalesOrderItems_2000 中表 Products 的非空值提供行，与该视图匹配。查询 9_v 中的额外谓语 V.Description IS NOT NULL 用于仅选择那些非提供空值的行。

```
CREATE MATERIALIZED VIEW V_SalesOrderItems_2000 AS
 SELECT s.ProductID, p.Description,
  s.Quantity, s.ShipDate, s.ID
 FROM SalesOrderItems s LEFT OUTER JOIN Products p
    ON ( s.ProductId = p.ID )
 WHERE s.ShipDate >= '2000-01-01'
  AND s.ShipDate <= '2001-01-01';
REFRESH MATERIALIZED VIEW V_SalesOrderItems_2000;
CREATE INDEX V_SalesOrderItems_shipdate ON V_SalesOrderItems_2000( ShipDate );
```

查询 6：

```
SELECT s.ProductID, p.Description,
  s.Quantity, s.ShipDate, s.ID
 FROM SalesOrderItems s LEFT OUTER JOIN Products p
    ON ( s.ProductId = p.ID )
 WHERE s.ShipDate >= '2000-01-01'
  AND s.ShipDate <= '2001-01-01'
 FOR READ ONLY;
```

查询 6_v：

```
SELECT V.ProductID,  V.Description, V.Quantity, V.ShipDate,
  SUM( V.Quantity ) OVER ( PARTITION BY V.ProductID
                       ORDER BY V.ShipDate
                       ROWS BETWEEN UNBOUNDED PRECEDING
                       AND CURRENT ROW ) AS cumulative_qty
 FROM V_SalesOrderItems_2000 as V
 FOR READ ONLY;
```

查询 7：

```
SELECT s.ProductID,  p.Description, s.Quantity, s.ShipDate,
  SUM( s.Quantity ) OVER ( PARTITION BY s.ProductID
                       ORDER BY s.ShipDate
                       ROWS BETWEEN UNBOUNDED PRECEDING
                       AND CURRENT ROW ) AS cumulative_qty
 FROM SalesOrderItems s LEFT OUTER JOIN Products p
    ON (s.ProductID = p.ID )

 WHERE s.ShipDate >= '2000-01-01'
   AND s.ShipDate <= '2001-01-01'
   AND s.Quantity >= 50
 FOR READ ONLY;
```

查询 7_v：

```
SELECT V.ProductID,  V.Description, V.Quantity, V.ShipDate,
  SUM( V.Quantity ) OVER ( PARTITION BY V.ProductID
                       ORDER BY V.ShipDate
                       ROWS BETWEEN UNBOUNDED PRECEDING
                       AND CURRENT ROW ) AS cumulative_qty
 FROM V_SalesOrderItems_2000 as V
 WHERE V.Quantity >= 50
 FOR READ ONLY;
```

查询 8：

```
SELECT s.ProductID, s.Quantity, s.ShipDate,
  SUM( s.Quantity ) OVER ( PARTITION BY s.ProductID
                       ORDER BY s.ShipDate
                       ROWS BETWEEN UNBOUNDED PRECEDING
                       AND CURRENT ROW ) AS cumulative_qty
 FROM SalesOrderItems s
 WHERE s.ShipDate >= '2000-01-01'
   AND s.ShipDate <= '2001-01-01'
   AND s.Quantity >= 50
 FOR READ ONLY;
```

查询 8_v：

```
SELECT V.ProductID, V.Quantity, V.ShipDate,
  SUM( V.Quantity ) OVER ( PARTITION BY V.ProductID
                       ORDER BY V.ShipDate
                       ROWS BETWEEN UNBOUNDED PRECEDING
                       AND CURRENT ROW ) AS cumulative_qty
 FROM V_SalesOrderItems_2000 as V
 WHERE V.Quantity >= 50
 FOR READ ONLY;
```

查询 9：

```
SELECT s.ProductID,  p.Description, s.Quantity, s.ShipDate,
  SUM( s.Quantity ) OVER ( PARTITION BY s.ProductID
```



```
                        ORDER BY s.ShipDate
                        ROWS BETWEEN UNBOUNDED PRECEDING
                        AND CURRENT ROW ) AS cumulative_qty
  FROM SalesOrderItems s JOIN Products p
       ON (s.ProductID = p.ID )

  WHERE s.ShipDate >= '2000-01-01'
    AND s.ShipDate <= '2001-01-01'
  FOR READ ONLY;
```

查询 9_v：

```
  SELECT V.ProductID,  V.Description, V.Quantity, V.ShipDate,
    SUM( V.Quantity ) OVER ( PARTITION BY V.ProductID
                        ORDER BY V.ShipDate
                        ROWS BETWEEN UNBOUNDED PRECEDING
                        AND CURRENT ROW ) AS cumulative_qty
  FROM V_SalesOrderItems_2000 as V
  WHERE V.Description IS NOT NULL
  FOR READ ONLY;
```

[查询执行] 算法

优化程序的功能是将某些 SQL 语句（SELECT、INSERT、UPDATE 或 DELETE）转换成由各种关系代数运算符（连接、重复排除、联合等）组成的有效访问计划。访问计划中的运算符在结构上可能与初始 SQL 语句不同，但访问计划的各种运算符计算出的结果在语义上与 SQL 所请求的结果等效。

访问计划中的关系代数运算符

访问计划包括一个由关系代数运算符所组成的树，这些运算符从树叶开始，它们使用查询的基本输入（通常为表中的各行）并自下而上对各行进行处理，以便树根生成最终结果。为便于理解，可查看图形形式的访问计划。请参见“读取执行计划”一节第 569 页和“读取图形式计划”一节第 571 页。

SQL Anywhere 支持这些不同关系代数运算符的多种实现。例如，SQL Anywhere 支持三种不同的内连接实现：嵌套循环连接、合并连接和散列连接。其中的每个运算符在特定环境中都有其各自的优点。查询优化程序会对一些参数进行分析以做出选择，这些参数包括高速缓存中表数据的数量、连接谓语的特征及选择性、输入到连接的顺序以及从中输出的顺序、可用于执行连接的内存量以及各种其它因素。

SQL Anywhere 在执行时可动态地从优化程序所选择的物理代数运算符切换到在逻辑上与初始运算符等效的另一不同物理算法。通常，此替代访问计划用于以下两种环境之一：

- 当用于执行语句的总内存量接近于内存调控器阈值时，则切换到一个可能执行得更慢、但却可以释放大量内存以供其它运算符（或其它请求）使用的策略。当这种情况发生时，QueryLowMemoryStrategy 属性将递增。此信息还出现在语句的图形式计划中。有关 QueryLowMemoryStrategy 属性的信息，请参见“连接属性”一节 《SQL Anywhere 服务器 - 数据库管理》。

  运算符可以使用的内存量取决于服务器的进程并发水平和活动连接数。

  有关内存调控器和进程并发水平的详细信息，请参见：

  - “SQL Anywhere 中的线程”一节 《SQL Anywhere 服务器 - 数据库管理》
  - “设置数据库服务器的进程并发水平”一节 《SQL Anywhere 服务器 - 数据库管理》
  - “内存调控器”一节第 530 页
- 如果在其执行开始时，特定运算符（例如，散列内连接）确定其输入与优化程序在优化时计算出的预期的基数不同。在这种情况下，运算符可能会切换到执行开销较小的不同策略。通常，此替代策略会使用索引嵌套循环处理。对于散列连接，QueryJHToJNLOptUsed 属性会在此切换发生时递增。在语句的图形式计划中也会出现连接方法切换。有关 QueryJHToJNLOptUsed 属性的信息，请参见“连接属性”一节 《SQL Anywhere 服务器 - 数据库管理》。

查询执行期间的并行

SQL Anywhere 支持两种不同类型的并行查询。查询间并行和查询内并行。查询间并行是指在多个单独的 CPU 上同时执行多个不同请求。每个请求（任务）都在单个线程上运行，并在单个处理器上执行。

查询内并行是指多个 CPU 同时处理一个请求，这样便可以在多处理器硬件上并行计算查询的各个部分。这些部分的处理由交换算法进行（请参见“交换算法（交换）”一节第 565 页）。若同时执行的查询数目通常少于可用处理器数目，则在这样的负载情况下，采用查询内并行会带来帮助。最大并行度由 max_query_tasks 选项的设置控制（请参见“max_query_tasks 选项 [数据库]”一节《SQL Anywhere 服务器 - 数据库管理》）。

仅当预期并行操作可提高性能时，优化程序才会估算并行操作的额外开销（额外复制行的开销以及协调工作的额外开销）并选择并行计划。

查询内并行机制不用于优先级选项设置为后台的连接。请参见“priority 选项 [数据库]”一节《SQL Anywhere 服务器 - 数据库管理》。

如果当前正在处理请求的服务器线程数（服务器属性 ActiveReq），最近超出了授权该数据库服务器使用的计算机上的 CPU 内核的数量，则不使用查询内并行机制。确切的时间段由服务器决定，通常为几秒钟。请参见“数据库服务器属性”一节 《SQL Anywhere 服务器 - 数据库管理》。

并行执行

某个查询是否可以利用并行执行，这取决于多种因素：

- 优化时系统中的可用资源（例如，内存、高速缓存中的数据量等）
- 计算机上逻辑处理器的数目
- 用于数据库存储的磁盘设备的数目，与处理器的相对速度以及计算机的 I/O 体系结构。
- 请求所需的特定代数运算符。SQL Anywhere 支持五种可以并行方式执行的代数运算符：
  - 并行顺序扫描（表扫描）
  - 并行索引扫描
  - 并行散列连接以及散列半连接和反半连接的并行版本
  - 并行嵌套循环连接以及嵌套循环半连接和反半连接的并行版本
  - 并行散列过滤
  - 并行散列分组依据

使用了不支持的运算符的查询在某些情况下仍然能够并行执行，但是就像在 dbisql 中看到的一样，计划中支持的运算符必须在不支持的运算符之下出现。在一个查询中，如果大部分不支持的运算符可在顶部附近出现，则该查询更有可能使用并行机制。例如，虽然排序运算符不能并行，但是如果将排序运算符放在计划的顶部且所有的并行运算符都在其之下，则在最外层块中使用了 ORDER BY 的查询就可以并行。相比之下，在派生表中使用了 TOP *n* 和 ORDER BY 的查询使用并行机制的可能性则比较小，因为排序必须在计划顶部之外的地方出现。

缺省情况下，SQL Anywhere 将假定任何 dbspace 均驻留在具有单个盘片的磁盘子系统上。尽管在此类环境下并行执行查询会有很多优点，但是，除非表数据全部驻留在高速缓存中，否则优化程序用于单个设备的 I/O 开销模型使得优化程序很难选择并行表或索引扫描。但若使用 ALTER DATABASE CALIBRATE PARALLEL READ 语句校准 I/O 子系统，优化程序即可通过更准确地估

算开销来判断并行执行的好处，并且在具有多个转轴的情况下，优化程序很可能会选择具有一定并行度的执行计划。

当查询内并行用于访问计划时，该计划将包含一个交换运算符，其作用是合并（联合）每个子树的并行计算的结果。交换运算符下的子树的数目就是并行度。每个子树或访问计划组件都是一个数据库服务器任务（请参见“-gn 服务器选项”一节 《SQL Anywhere 服务器 - 数据库管理》）。数据库服务器内核会基于执行线程（或纤程）的可用性以对待单个 SQL 请求的方式调度这些要执行的任务。这种体系结构意味着任何访问计划的并行计算主要是自行调优的，因为并行执行任务的工作在服务器内核允许的线程（纤程）上进行调度，并平均地执行计划组件任务。

另请参见

- “max_query_tasks 选项 [数据库]”一节 《SQL Anywhere 服务器 - 数据库管理》
- “SQL Anywhere 中的线程”一节 《SQL Anywhere 服务器 - 数据库管理》
- “-gtc 服务器选项”一节 《SQL Anywhere 服务器 - 数据库管理》
- “设置数据库服务器的进程并发水平”一节 《SQL Anywhere 服务器 - 数据库管理》
- “交换算法（交换）”一节第 565 页
- “读取执行计划”一节第 569 页
- “ALTER DATABASE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》

查询中的并行机制

如果一个查询处理的行数比其返回的行数多很多，则该查询更有可能使用并行机制。在这种情况下，所处理的行数包括所有扫描的行的大小加上所有中间结果的大小。因为将使用索引跳过表的大部分，所以不包括不会被扫描的行。理想的情况是对大型表使用单行 GROUP BY，它将扫描很多行但只返回一行。如果组的规模很大，多组查询也可以作为备选方案。删除许多行的任何谓语或连接条件，也是并行处理的很好选择。

下表列出了无论在优化还是在执行时间上，查询都无法使用并行机制的情况：

- 服务器计算机没有多处理器
- 服务器计算机未授权使用多处理器。可查看服务器属性 NumLogicalProcessorsUsed 检查此项。但是，请注意，超线程处理器不能计入查询内并行机制，因此，如果计算机是超线程的，必须将 NumLogicalProcessorsUsed 的值除以二。
- max_query_tasks 选项设置为 1
- 优先级选项设置为后台
- 包含查询的语句不是 SELECT 语句
- 最近一段时间以来，ActiveReq 的值一直大于或等于 NumLogicalProcessorsUsed 的值（如果计算机是超线程的，请将处理器数除以二）
- 没有足够的任务可用。

另请参见

- “查询执行期间的并行”一节第 549 页
- “SQL Anywhere 中的线程”一节 《SQL Anywhere 服务器 - 数据库管理》
- “max_query_tasks 选项 [数据库]”一节 《SQL Anywhere 服务器 - 数据库管理》
- “priority 选项 [数据库]”一节 《SQL Anywhere 服务器 - 数据库管理》
- max_query_tasks、priority、NumLogicalProcessorsUsed 和 ActiveReq 属性： “数据库服务器属性”一节 《SQL Anywhere 服务器 - 数据库管理》
- “CREATE DATABASE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “ALTER TABLE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》

表访问方法

本节介绍用于访问表的各种方法，其中表扫描和索引扫描是最常用的方法。

IndexScan 方法

IndexScan 使用索引来确定哪些行满足搜索条件。索引扫描在访问表之前帮助减少符合条件的行集。索引扫描还可以将行按排序的顺序返回。

IndexScan 在简要计划中显示为 `correlation-name<index-name>`，其中 *correlation-name* 是在 FROM 子句中指定的相关名（如果未指定相关名，则为表名），*index-name* 是索引的名称。

索引为读取大表中的少数行提供了高效的机制。但是，在从表读取许多行时，索引扫描比顺序扫描开销更大。索引扫描从数据库中读取页的顺序是随机的，这种方法所占用的资源比按顺序读取所占用的资源多。如果页上有多行满足搜索条件，索引扫描还可以多次引用同一表页。如果索引扫描仅匹配少数几页，这些页很可能会留在高速缓存中，而多次访问不会导致额外的 I/O。但是，如果有多页匹配搜索条件，它们可能无法全部装入高速缓存。这会使索引扫描从磁盘中多次读取同一页。

如果搜索条件具有 sargable 特性，并且优化程序对搜索条件选择性的估计值较小，使索引扫描的开销低于顺序表扫描的开销，则优化程序将使用索引扫描来满足搜索条件。

索引扫描也可以在从索引读取行之后计算非可优化搜索条件。在索引扫描中计算条件比在索引扫描之后在过滤器中进行计算要略为有效。

即使没有要满足的搜索条件，索引也可用于满足在 ORDER BY 子句中显式定义或者在 GROUP BY 或 DISTINCT 子句中隐式需要的排序要求。与基于散列的分组和非重复方法相比，带 GROUP BY 和 DISTINCT 的排序方法可以更快地返回初始行，但它们在返回整个结果集时的速度可能较慢。

如果 optimization_goal 设置为 first-row，则优化程序往往会优先使用索引扫描而非顺序表扫描。这是因为索引扫描会比表扫描更快地返回查询的前几行。

在编写查询时，可以指定索引提示来告知优化程序要使用哪些索引及如何使用它们。但是，索引提示会替换查询优化程序的决策逻辑，因此只应由经验丰富的用户使用。使用索引提示可能会导致访问计划达不到最优，并会导致性能较差。请参见“FROM 子句”一节 《SQL Anywhere 服务器 - SQL 参考》。

IndexOnlyScan 方法 (IO)

当优化程序使用的索引包含满足查询所需的基础表的所有数据时，可以完全避免从基础表读取值，从而可以直接从索引检索数据。这称为**仅索引检索**。仅索引检索可以减少满足查询所需的 I/O 和高速缓存量并提高性能。只要有可能，优化程序就会执行仅索引检索。

另请参见

- “FROM 子句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “在查询中使用谓语”一节第 531 页
- “optimization_goal 选项 [数据库]”一节 《SQL Anywhere 服务器 - 数据库管理》
- “读取执行计划”一节第 569 页

MultipleIndexScan 方法 (MultIdx)

当可以使用或必须使用多个索引以满足查询（该查询包含一组通过逻辑运算符 AND 或 OR 组合的搜索条件）时，使用 MultipleIndexScan。MultipleIndexScan 将多个 IndexScan 方法与其它运算符组合在一起，以满足搜索条件。

当使用多个索引计算使用 AND 运算符组合的谓语时，MultipleIndexScan 会执行索引交集操作。当使用多个索引计算使用 OR 运算符组合的谓语时，MultipleIndexScan 会执行索引并集操作。但是，请注意，MultipleIndexScan 并不只限于并集或交集操作，例如，MultipleIndexScan 可以使用外连接执行和索引并集。

可以通过检查执行计划来决定是否将多索引扫描用于特定查询。在简要计划中，多索引扫描方法显示为 `table-name<MultIdx...`，后面跟有使用的索引列表。

在详细计划和图形式计划中，通过 MultipleIndexScan 节点指示多索引扫描的使用，该节点下的条目提供有关使用了哪些索引以及如何组合其结果的详细信息。

另请参见

- “FROM 子句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “在查询中使用谓语”一节第 531 页
- “读取执行计划”一节第 569 页

ParallelIndexScan 方法

使用 ParallelIndexScan 时，各个 IndexScan 运算符借助一个交换运算符协同工作，从而以并行方式执行索引扫描。当每个 IndexScan 运算符需要行时，它会读取下一未处理的叶页并返回该页中的行，一次返回一行。这样，这些页便在各个 IndexScan 运算符之间分配，从而实现并行处理。无论这些页在 IndexScan 运算符之间如何分配，所有行都会被访问到。

TableScan 方法 (seq)

TableScan 按照行在数据库中的存储顺序来读取表上所有页中的所有行。这称为顺序表扫描。

顺序表扫描在简要文本计划和详细文本计划中显示为 *correlation_name<seq>*，其中 *correlation_name* 是在 FROM 子句中指定的相关名（如果未指定相关名，则为表名）。

当大多数表页可能具有匹配查询搜索条件的行或者尚未定义合适的索引时，将使用顺序表扫描。

虽然顺序表扫描可能会比索引扫描读取更多的页，但由于这些页是以连续块的形式从磁盘中读取的，因此磁盘 I/O 的开销会非常低（如果数据库文件在磁盘上没有碎片，这种性能改善将最为明显）。顺序 I/O 会减少磁盘头移动和旋转的等待时间。对于大表，顺序表扫描还会一次读取包含多个页的组。这将进一步降低顺序表扫描相对于索引扫描的开销。

虽然顺序表扫描可能会比匹配许多行的索引扫描花费的时间少，但如果顺序表扫描被执行多次，它们将无法像索引扫描那样有效地利用高速缓存。由于索引扫描访问的表页很可能较少，因此这些页很可能会在高速缓存中，这将加快访问的速度。所以，对于重复的表访问（如嵌套循环连接的右侧），最好采用索引扫描。

对于以隔离级别 3 执行的事务，SQL Anywhere 会在所访问的每一行上获取锁—即使该行不满足搜索条件。在这一隔离级别，顺序表扫描会在表中的所有行上获取锁，而索引扫描只在匹配搜索条件的行上获取锁。这意味着顺序表扫描在多用户环境中可能会显著地降低吞吐量。因此，优化程序在隔离级别 3 会非常倾向于优先使用索引访问，而不是顺序访问。顺序扫描可在扫描过程中有效地计算表列和常数之间的简单比较谓语。其它只引用要扫描的表的搜索条件在这些简单比较之后进行计算，此方法比顺序扫描后在过滤器中计算条件要略为高效。

ParallelTableScan 方法

使用 ParallelTableScan 时，各个 TableScan 运算符借助一个交换运算符协同工作，从而以并行方式执行顺序表扫描。当每个 TableScan 运算符需要行时，它会读取下一未处理的表页并返回该页中的行，一次返回一行。这样，这些页便在各个 TableScan 运算符之间分配，从而实现并行处理。无论这些页在并行 TableScan 运算符之间如何分配，表中所有行都会被访问到。

HashTableScan 方法 (HTS)

HashTableScan 扫描散列连接的生成端，就像它是内存表一样，从而将具有下面第一种结构的计划转换成具有下面第二种结构的计划，其中 `idx` 是可用于探查存储在散列表中的连接键值的索引：

```
table1<seq>*JH ( <operator>... ( table2<seq> ) )
```

```
table1<seq>*JF ( <operator>... ( HTS JNB table2<idx> ) )
```

当散列连接和扫描之间具有干预运算符时，散列表扫描会减少必须由其它运算符处理的所需行数。当索引探测的选择性很高时（例如，当生成端的行数与索引的基数相比较少时），此策略最有用。

注意
如果散列连接的生成端很大，则执行常规顺序扫描会更有效。

优化程序会计算域值生成大小，方式与它为散列连接替代执行计算阈值相似。如果生成端中的行数超过此阈值，则会放弃 HashTableScan，执行期间 (`HTS JNB table<idx>`) 将被视为顺序扫描 (`table<seq>`)。

注意
如果散列表的生成端不得不溢出到磁盘，则会使用顺序策略。

RowIdScan 方法 (ROWID)

RowIdScan 用于定位基表或临时表中的某一行，这基于使用 ROWID 函数的相等比较谓语来实现。比较谓语可以指常量文字，但通常 ROWID 函数是与由系统函数或过程调用（如 sa_locks）返回的行标识符值一起使用。

RowId 扫描在简要和详细文本计划中显示为 `correlation-name<ROWID>`，其中 *correlation-name* 是在 FROM 子句中指定的相关名（如果未指定相关名，则为表名）。

RowIdScan 不能辨别 ROWID 函数所引用的给定表中的无效行标识符，也不能辨别给定行标识符不再存在这一情况。因此，如果无法在表中找到比较谓语中指定的行标识符，则 RowIdScan 将返回空集。

另请参见

- “ROWID 函数 [Miscellaneous]”一节 《SQL Anywhere 服务器 - SQL 参考》
- “sa_locks 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》

算法类型

连接算法

SQL Anywhere 提供了多种不同的连接实现方法供查询优化程序进行选择。每个连接算法都有其具体特征，因此对于给定的查询和给定的执行环境而言，都会存在有的算法会比较适用而有的算法则不太适用的情况。

访问计划中连接的顺序与初始 SQL 语句中连接的顺序可能相符也可能不符；查询优化程序将负责以最低执行开销为前提为每个查询选择最佳连接策略。某些情况下，可能会为一些复杂语句使用查询重写优化，以增加或减少任何特定语句的计算连接数量。

以下是 SQL Anywhere 支持的三种连接算法，不过每种算法都具有其它变体：

- **嵌套循环连接**　最直接的算法为嵌套循环连接。对于左侧的每一行，都会基于连接条件对右侧进行扫描以查找匹配项。通常会通过索引来访问右侧的各行，以减少总执行开销。这种情况常常称为**索引嵌套循环连接**。

  嵌套循环连接具有支持 LEFT OUTER 和 FULL OUTER 连接的变体。还可以将嵌套循环实现用于半连接（最常用于处理 EXISTS 子查询）。

  尽管计算不等式条件上的连接效率非常低，但无论连接条件的特征为何，都可以使用嵌套循环连接。

  使用嵌套循环 FULL OUTER 连接来计算任何大小的输入需要的开销都非常大，只有其它连接算法都不能使用时，查询优化程序才会选择使用这种算法。

- **合并连接** 合并连接依赖于将其两个输入按连接属性进行排序。连接条件必须至少包含一个相等谓语，查询优化程序才会选择此方法。基本算法就是直接合并两个输入：当两个连接属性的值不同时，算法会滚动到左侧或右侧（取决于哪一侧的值小）的下一行。当具有多个匹配项时，可能需要进行回溯。

  提供了合并连接变体以支持 LEFT OUTER 和 FULL OUTER 连接。FULL OUTER 连接的合并连接明显比嵌套循环连接更为高效。

  基本的合并连接算法还用于支持 SQL 集合运算符 EXCEPT 和 INTERSECT，尽管这些变体已在访问计划中显式命名为 EXCEPT 或 INTERSECT 算法。

- **散列连接** 散列连接是 SQL Anywhere 数据库服务器所支持的最通用的连接方法。简言之，散列连接算法会用两个输入中的较小输入建立内存中的散列表，然后读取较大的输入并探查内存中的散列表以查找匹配项。

  提供了散列连接变体以支持 LEFT OUTER 连接、FULL OUTER 连接、半连接和反半连接。此外，当使用递归 UNION 查询表达式时，SQL Anywhere 还支持用于递归 INNER 和 LEFT OUTER 连接的散列连接变体。

  可以并行执行散列内连接、左外连接、半连接和反半连接算法。

  如果由算法构建的内存中散列表不适合可用内存，则散列连接算法会将输入拆分成多个部分（对非常大的输入可能递归），然后单独对每个部分执行连接。如果没有足够的高速缓存来保存连接属性为特定值的所有行，则在可能的情况下，每个散列连接都会在第一次放弃中间结果后动态地切换到基于索引的嵌套循环策略，以避免用尽语句的内存消耗限额。

  还使用散列连接的变体来支持 SQL 查询表达式 EXCEPT 和 INTERSECT，尽管这些变体已在访问计划中显式命名为 EXCEPT 或 INTERSECT 算法。

HashJoin 算法（JH、JHSP、JHFO、JHAP、JHO、JHPO）

HashJoin 会用两个输入中的较小输入建立内存中的散列表，然后读取较大的输入并探查内存中的散列表以查找匹配项，这些匹配项会被写入工作表中。如果内存无法容纳较小的输入，HashJoin 就会将这两个输入分为较小的工作表。将以递归的方式对这些较小的工作表进行处理，直到较小的输入能够装入内存为止。

HashJoin 还会执行以下操作：

- 在返回第一行之前计算其结果中的所有行
- 使用一个工作表，该工作表提供不敏感的语义（除非已请求对值敏感的游标）
- 以并行方式执行
- 在输入中的行被复制到内存之前，将这些行锁定

如果较小的输入能够装入内存，则无论较大输入的大小如何，HashJoin 的性能都将达到最佳。通常，如果其中一个输入将明显小于另一个输入，则优化程序就会选择散列连接。

如果 HashJoin 的运行环境中没有足够的高速缓存来保存其连接属性为特定值的所有行，HashJoin 将无法完成。在这种情况下，HashJoin 会放弃中间结果，而改用基于索引的 NestedLoopsJoin。较小表的所有行都将被读取并用来探查工作表以查找匹配项。这种基于索引的策略明显比其它连接方

法慢得多，如果优化程序检测到在查询执行过程中可能会出现内存不足的情况，则它将避免使用散列连接生成访问计划。

HashJoin 运算符可以使用的内存量取决于服务器的进程并发水平和活动连接数。请参见“SQL Anywhere 中的线程”一节 《SQL Anywhere 服务器 - 数据库管理》和“设置数据库服务器的进程并发水平”一节 《SQL Anywhere 服务器 - 数据库管理》。

当由于内存不足而需要嵌套循环策略时，性能计数器将递增。您可以使用 QueryLowMemoryStrategy 数据库属性或连接属性读取此监控器，或者在 Windows 性能监控器的 [查询：或者在 Windows 性能监控器的 [查询：内存不足策略] 计数器中读取此监控器。

当内存不足时，将在 Windows Mobile 上禁用 HashJoin。

注意
Windows Mobile 上可能未提供 Windows 性能监控器。

有关详细信息，请参见“连接属性”一节 《SQL Anywhere 服务器 - 数据库管理》中的 QueryLowMemoryStrategy 和“设置数据库服务器的进程并发水平”一节 《SQL Anywhere 服务器 - 数据库管理》。

RecursiveHashJoin 算法 (JHR)

RecursiveHashJoin 是 HashJoin 的变体，用于递归联合查询。有关详细信息，请参见“HashJoin 算法（JH、JHSP、JHFO、JHAP、JHO、JHPO）”一节第 555 页和“递归公用表表达式”一节第 411 页。

RecursiveLeftOuterHashJoin 算法 (JHRO)

RecursiveLeftOuterHashJoin 是 HashJoin 的变体，用于某些递归联合查询。有关详细信息，请参见“HashJoin 算法（JH、JHSP、JHFO、JHAP、JHO、JHPO）”一节第 555 页和“递归公用表表达式”一节第 411 页。

HashSemijoin 算法 (JHS)

HashSemijoin 在左侧和右侧之间执行半连接。右侧只用于确定左侧的哪些行将显示在结果中。使用 HashSemijoin 时，将读取右侧，以形成内存中的散列表，然后将按左侧的每一行探查该表。如果找到匹配项，左侧的行被输出到结果中，匹配过程将重新开始以查找左侧的下一行。必须至少有一个等值连接条件，查询优化程序才会考虑使用 HashSemijoin。与 NestedLoopsSemijoin 相同，若连接的输入包括来自已被重写为连接的存在限定（IN、SOME、ANY、EXISTS）嵌套查询的表表达式，则可使用 HashSemijoin。如果连接条件包括非等值条件，或不存在合适的索引使得对右侧的索引检索开销足够低，则使用 HashSemijoin 将比使用 NestedLoopsSemijoin 的效果更好。

与 HashJoin 一样，如果没有足够的高速缓存来完成操作，HashSemijoin 可能会恢复为嵌套循环半连接策略。如果是这样，性能计数器将递增。您可以使用 QueryLowMemoryStrategy 数据库属性或连接属性读取此监控器，或者在 Windows 性能监控器的 [查询：或者在 Windows 性能监控器的 [查询：内存不足策略] 计数器中读取此监控器。

HashSemijoin 运算符可以使用的内存量取决于服务器的进程并发水平和活动连接数。请参见“SQL Anywhere 中的线程”一节 《SQL Anywhere 服务器 - 数据库管理》和“设置数据库服务器的进程并发水平”一节 《SQL Anywhere 服务器 - 数据库管理》。

注意
Windows Mobile 上可能未提供 Windows 性能监控器。

有关详细信息，请参见“连接属性”一节 《SQL Anywhere 服务器 - 数据库管理》中的 QueryLowMemoryStrategy。

HashAntisemijoin 算法 (JHA)

HashAntisemijoin 在左侧和右侧之间执行反半连接。右侧只用于确定左侧的哪些行将显示在结果中。使用 HashAntisemijoin 时，将读取右侧，以形成内存中的散列表，然后将按左侧的每一行探查该表。仅当没有从右侧找到任何匹配的行时，才会输出左侧的每行。当连接的输入包括来自可被重写为反连接的限定（NOT IN、ALL、NOT EXISTS）嵌套查询的表表达式时，可以使用 HashAntisemijoin。如果不存在合适的索引使得对右侧的索引检索开销足够低，则使用 HashAntisemijoin 比对引用限定查询的搜索条件进行计算的效果更好。

与 HashJoin 一样，如果没有足够的高速缓存来完成操作，HashAntisemijoin 可能会恢复为嵌套循环策略。如果是这样，性能计数器将递增。您可以使用 QueryLowMemoryStrategy 数据库属性或连接属性读取此监控器，或者在 Windows 性能监控器的 [查询：或者在 Windows 性能监控器的 [查询：内存不足策略] 计数器中读取此监控器。

HashAntisemijoin 运算符可以使用的内存量取决于服务器的进程并发水平和活动连接数。请参见“SQL Anywhere 中的线程”一节 《SQL Anywhere 服务器 - 数据库管理》和“设置数据库服务器的进程并发水平”一节 《SQL Anywhere 服务器 - 数据库管理》。

注意
Windows Mobile 上可能未提供 Windows 性能监控器。

有关详细信息，请参见“连接属性”一节 《SQL Anywhere 服务器 - 数据库管理》中的 QueryLowMemoryStrategy。

MergeJoin 算法（JM、JMFO、JMO）

MergeJoin 读取均已按照连接属性进行排序的两个输入。对于左侧输入中的每个行，该算法通过按排序顺序对行进行访问来读取右侧输入中的所有匹配行。

如果输入尚未按连接属性排序（可能是因为先前的合并连接或因为曾使用索引来满足搜索条件），优化程序将添加一个排序，以生成正确的行顺序。这一排序会增加合并连接的开销。

与 HashJoin 相比，MergeJoin 的优点之一是排序开销可以由若干个连接分摊（只要这些合并连接涉及相同的属性）。如果输入的大小可能比较接近或者排序开销可以由若干个操作分摊，优化程序将优先选择 MergeJoin 而不是 HashJoin。

NestedLoopsJoin 算法（JNL、JNLFO、JNLO）

NestedLoopsJoin 通过为左侧的每行完全读取右侧的内容来计算左右两侧的连接。（由于优化程序将为请求中的每个块选择正确的连接顺序，因此表在查询中的语法顺序无关紧要。）

如果连接条件不包含相等条件或要使用第一行优化目标对语句进行优化（即，optimization_goal 选项设置为 First-Row，或在 FROM 子句中将 FASTFIRSTROW 指定为表提示），则优化程序可能会选择 NestedLoopsJoin。

由于 NestedLoopsJoin 会多次读取右侧，因此它对右侧的开销非常敏感。如果右侧是索引扫描或小表，则可以使用先前迭代中的高速缓存页计算右侧。但是，如果右侧是顺序表扫描或是匹配许多行的索引扫描，则需要从磁盘中多次读取右侧。通常，NestedLoopsJoin 的效率比其它连接方法的效率低。但是，与必须在返回之前计算整个结果的连接方法相比，NestedLoopsJoin 可以快速地返回第一个匹配行。

NestedLoopsJoin 是唯一可以为包含连接的查询提供敏感语义的连接算法。这意味着连接上的敏感游标只能用 NestedLoopsJoin 来执行。

另请参见

- “optimization_goal 选项 [数据库]”一节 《SQL Anywhere 服务器 - 数据库管理》
- “FROM 子句”一节 《SQL Anywhere 服务器 - SQL 参考》

NestedLoopsSemijoin 算法 (JNLS)

与 NestedLoopsJoin 类似，NestedLoopsSemijoin 通过为左侧的每行扫描右侧的内容来连接其输入。由于使用了 NestedLoopsJoin，可能会多次读取右侧，因此对于较大的输入，最好进行索引扫描。

NestedLoopsSemijoin 与 NestedLoopsJoin 在两方面有所不同。第一，NestedLoopsSemijoin 只输出左侧的值；右侧只用来限制左侧的哪些行将显示在结果中。第二，在遇到第一个匹配项后，NestedLoopsSemijoin 将停止右侧的每个搜索。当连接的输入包括来自已被重写为连接的存在限定（IN、SOME、ANY、EXISTS）嵌套查询的表表达式时，可以使用 NestedLoopsSemijoin。

NestedLoopsAntisemijoin 算法 (JNLA)

与 NestedLoopsJoin 算法类似，NestedLoopsAntisemijoin 通过为左侧的每行扫描右侧的内容来连接其输入。由于使用了 NestedLoopsJoin，可能会多次读取右侧，因此对于较大的输入，最好进行索引扫描。NestedLoopsAntisemijoin 与 NestedLoopsJoin 的不同之外在于，它只输出左侧的值；右侧只用来限制左侧的哪些行将显示在结果中。具体来说，仅当在右侧没有对应值时，才会包括左侧的值。

重复排除算法

重复排除运算符会产生不含重复行的输出。重复排除节点可以由优化程序引入（例如在将嵌套查询转换为连接时）。

有关详细信息，请参见“HashDistinct 算法 (DistH)”一节第 559 页和“OrderedDistinct 算法 (DistO)”一节第 559 页。

HashDistinct 算法 (DistH)

HashDistinct 采用单个输入并返回所有非重复行。HashDistinct 通过读取其输入并建立内存中的散列表来实现此目的。如果在散列表中找到某一输入行，将忽略该输入行；否则，该输入行将被写入工作表中。如果输入不能完全装入内存中的散列表，则会将其分割为多个较小的工作表，并以递归方式进行处理。

HashDistinct 还会执行以下操作：

- 如果非重复行能够装入内存中的表，则该算法会非常顺利地执行，而与输入中的总行数无关。
- 使用一个工作表，因此可以提供不敏感的语义或对值敏感的语义。
- 找到以前没有返回的行时，返回该行。但是，必须完全实现非重复散列的结果后，才能从查询返回。如果必要，优化程序将为执行计划添加一个工作表以确保实现此目标。
- 会锁定其输入的行。

如果优化程序检测到在查询执行过程中可能会出现内存不足的情况，它将避免使用非重复散列算法生成访问计划。如果 HashDistinct 的运行环境中的可用高速缓存极少，该算法将无法完成。在这种情况下，HashDistinct 会放弃中间结果，而改用内部内存不足方法。

HashDistinct 运算符可以使用的内存量取决于服务器的进程并发水平和活动连接数。请参见“SQL Anywhere 中的线程”一节 《SQL Anywhere 服务器 - 数据库管理》和“设置数据库服务器的进程并发水平”一节 《SQL Anywhere 服务器 - 数据库管理》。

OrderedDistinct 算法 (DistO)

如果输入按所有列进行排序，则可以使用 OrderedDistinct。该算法读取每一行并将之与前一行进行比较。如果相同，则忽略该行；否则，将输出该行。如果行已经排序（可能是由于索引或合并连接），OrderedDistinct 将非常有效；如果输入未排序，优化程序将插入一个排序。OrderedDistinct 本身不使用任何工作表，但是所插入的任何排序都会使用一个工作表。

分组算法

分组算法计算输入的汇总。只有当查询包含 GROUP BY 子句或集合函数（如 `SELECT COUNT(*) FROM T`）时，分组算法才适用。

有关详细信息，请参见“HashGroupBy 算法 (GrByH)”一节第 560 页、“OrderedGroupBy 算法 (GrByO)”一节第 561 页和“SingleRowGroupBy 算法 (GrByS)”一节第 561 页。

HashGroupBy 算法 (GrByH)

HashGroupBy 建立每组包含一行的内存中的散列表。当读取输入行时，将在工作表中查找关联组。它会更新集合函数并将组行重写到工作表中。如果没有找到组记录，则将初始化新的组记录并将其插入到工作表中。

HashGroupBy 可在返回第一行之前计算其所有结果行，并且可用于满足完全敏感的游标或对值敏感的游标。必须完全实现散列分组依据的结果后，才能从查询返回。如果必要，优化程序将为执行计划添加一个工作表以确保实现此目标。

可以按并行方式执行 HashGroupBy。

如果组能够装入内存，HashGroupBy 就会非常顺利地执行，与输入的大小无关。如果内存无法容纳散列表，将把输入分为多个较小的工作表，并以递归的方式对这些表进行分区，直到能够装入内存为止。如果优化程序检测到在查询执行过程中可能会出现内存不足的情况，它将避免使用 HashGroupBy 生成访问计划。如果没有足够的内存来保存分区，则优化程序就会放弃 HashGroupBy 的中间结果，而改用内部内存不足策略。

HashGroupBy 运算符可以使用的内存量取决于服务器的进程并发水平和活动连接数。请参见“SQL Anywhere 中的线程”一节 《SQL Anywhere 服务器 - 数据库管理》和“设置数据库服务器的进程并发水平”一节 《SQL Anywhere 服务器 - 数据库管理》。

ClusteredHashGroupBy 算法 (GrByHClust)

在某些情况下，输入表的分组列中的值会聚簇，这样，相似的值便可以靠近在一起显示。例如，如果表包含的列始终设置为当前日期，则具有一个日期的所有行在表中就会相对靠近。ClusteredHashGroupBy 利用这种聚簇。

当为明显大于可用内存的表分组时，优化程序可以使用 ClusteredHashGroupBy。特别是当 HAVING 谓语只返回一小部分行时，该方法十分有效。

如果在更新数据与执行查询同时进行的环境中选择 ClusteredHashGroupBy，则可能导致优化程序部分执行大量无用工作。因此，ClusteredHashGroupBy 最适合于偶尔进行批处理式更新和基于读取的查询的 OLAP 负载。将 optimization_workload 选项设置为 OLAP 以指示优化程序应在其调查的可能性中包括 ClusteredHashGroupBy。请参见“optimization_workload 选项 [数据库]”一节 《SQL Anywhere 服务器 - 数据库管理》。

当创建可以在 OLAP 负载中使用的索引或外键时，请指定 FOR OLAP WORKLOAD 子句。该子句可使数据库服务器保留一条由 ClusteredHashGroupBy 所使用的、有关相同键内两行之间的最大页面距离的统计信息。请参见“CREATE INDEX 语句”一节 《SQL Anywhere 服务器 - SQL 参考》、“CREATE TABLE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》和“ALTER TABLE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

有关 OLAP 负载的详细信息，请参见“OLAP 支持”第 423 页。

HashGroupBySets 算法 (GrByHSets)

HashGroupBy 的变体 HashGroupBySets 在执行 GROUPING SETS 查询时使用。

不能以并行方式执行 HashGroupBySets。

有关详细信息，请参见“HashGroupBy 算法 (GrByH)”一节第 560 页。

OrderedGroupBy 算法 (GrByO)

OrderedGroupBy 读取按分组列排序的输入。当读取每一行时，该算法会将该行与前一行进行比较。如果分组列匹配，则更新当前组；否则，将输出当前组并启动新组。

OrderedGroupBySets 算法 (GrByOSets)

OrderedGroupBy 的变体 OrderedGroupBySets 在执行 GROUPING SETS 查询时使用。此算法要求输入按分组列排序。请参见“OrderedGroupBy 算法 (GrByO)”一节第 561 页。

SingleRowGroupBy 算法 (GrByS)

当没有指定 GROUP BY 时，SingleRowGroupBy 用于生成单行集合。单个组行将保留在内存中，并针对每个输入行进行更新。

SortedGroupBySets 算法 (GrBySSets)

处理包含 GROUPING SETS 的 OLAP 查询时使用 SortedGroupBySets。

查询表达式算法

查询表达式算法可分为以下几类：

- 排除算法，包括 MergeExcept 和 HashExcept
- 交叉算法，包括 MergeIntersect 和 HashIntersect
- 联合算法，包括 Union、UnionAll 和 RecursiveUnion

排除算法（EAH、EAM、EH、EM）

SQL Anywhere 查询优化程序可以从集合差异 SQL 运算符 EXCEPT 的以下两个物理实现中进行选取：基于排序的变体 MergeExcept (EM) 和基于散列的变体 HashExcept (EH)。

MergeExcept 使用 MergeJoin 通过按排序顺序分析匹配行来计算两个输入之间的集合差异。通常，要求对两个输入进行显式排序。同样，HashExcept 使用 HashAntisemijoin 来计算两个输入之间的集合差异，使用左外散列连接来计算两个输入之间的差异 (EXCEPT ALL)。

当检测到内存不足时，HashExcept 可以动态地切换到嵌套循环策略。如果发生这种情况，性能计数器将递增。可以使用 QueryLowMemoryStrategy 数据库属性或连接属性读取此监控器，也可以在图

形式计划中的 QueryLowMemoryStrategy 统计信息中（如果运行时有统计信息）或者在 Windows 性能监控器的 [查询：内存不足策略] 计数器中读取此监控器。

当内存不足时，将在 Windows Mobile 上禁用 HashExcept。

对于 EXCEPT，MergeExcept 和 HashExcept 与 DISTINCT 算法之一结合使用，以确保结果中不包含重复项。对于 EXCEPT ALL，HashExceptAll 和 MergeExceptAll 与 RowReplicate 结合使用，RowReplicate 用于计算结果中包含的重复行的正确数目。

另请参见

- “EXCEPT 子句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “使用 UNION、INTERSECT 和 EXCEPT 对查询结果执行集合运算”一节第 354 页
- QueryLowMemoryStrategy 连接属性： “连接属性”一节 《SQL Anywhere 服务器 - 数据库管理》
- QueryLowMemoryStrategy 数据库属性： “数据库属性”一节 《SQL Anywhere 服务器 - 数据库管理》
- 查询：内存不足策略的统计信息： “性能监控器统计”一节第 197 页

交叉算法（IH、IM、IAH、IAM）

SQL Anywhere 查询优化程序可以从集合交叉 SQL 运算符 INTERSECT 的以下两个物理实现中进行选取：基于排序的变体 MergeIntersect (IM) 和基于散列的变体 HashIntersect (IH)。

MergeIntersect 使用 MergeJoin 通过按排序顺序分析行匹配来计算两个输入之间的集合交叉。通常，要求对两个输入进行显式排序。同样，HashIntersect 使用 HashJoin 来计算两个输入之间的集合和包交集（INTERSECT 和 INTERSECT ALL）。

如有必要，当检测到内存不足时，HashIntersect 可以动态地切换到嵌套循环策略。如果发生这种情况，性能计数器将递增。可以使用 QueryLowMemoryStrategy 数据库属性或连接属性读取此监控器，也可以在图形式计划中的 QueryLowMemoryStrategy 统计信息中（如果运行时有统计信息）或者在 Windows 性能监控器的 [查询：内存不足策略] 计数器中读取此监控器。

当内存不足时，将在 Windows Mobile 上禁用 HashIntersect。

对于 INTERSECT，MergeIntersect 或 HashIntersect 与 DISTINCT 算法之一结合使用，以确保结果中不包含重复项。对于 INTERSECT ALL 运算符，MergeIntersectAll 和 HashIntersectAll 与 RowReplicate 结合使用，RowReplicate 用于计算结果中包含的重复行的正确数目。

另请参见

- “INTERSECT 子句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “使用 UNION、INTERSECT 和 EXCEPT 对查询结果执行集合运算”一节第 354 页
- QueryLowMemoryStrategy 连接属性： “连接属性”一节 《SQL Anywhere 服务器 - 数据库管理》
- QueryLowMemoryStrategy 数据库属性： “数据库属性”一节 《SQL Anywhere 服务器 - 数据库管理》
- 查询:内存不足策略的统计信息： “性能监控器统计”一节第 197 页

RecursiveTable 算法 (RT)

递归表是作为查询中的 WITH 子句的结果构建的公用表表达式，其中 WITH 子句用于递归联合查询。公用表表达式是临时视图，它们只在某一条 SELECT 语句的范围内是已知的。请参见“公用表表达式”第 403 页。

RecursiveUnion 算法 (RU)

RecursiveUnion 在执行递归联合查询期间应用。请参见“递归公用表表达式”一节第 411 页。

RowReplicate 算法 (RR)

RowReplicate 在执行 EXCEPT ALL 和 INTERSECT ALL 等集合操作期间使用。它是以下这样的运算的功能：结果集中的行数与作为运算对象的两个集合中的行数显式相关。RowReplicate 可确保结果集中的行数正确。请参见“使用 UNION、INTERSECT 和 EXCEPT 对查询结果执行集合运算”一节第 354 页。

UnionAll 算法 (UA)

UnionAll 从其每一项输入中读取行并将这些行输出，而不管是否有重复项。该算法用于实现 UNION 和 UNION ALL 子句。在 UNION 子句中，需要使用重复排除算法（如 HashDistinct 或 OrderedDistinct）来删除 UnionAll 所生成的任何重复项。

请参见“HashDistinct 算法 (DistH)”一节第 559 页和“OrderedDistinct 算法 (DistO)”一节第 559 页。

排序算法

排序算法在查询包括 ORDER BY 子句或查询的执行策略需要对其输入进行总体排序时适用。

有关详细信息，请参见“排序算法（排序）”一节第 563 页和“UnionAll 算法 (UA)”一节第 563 页。

排序算法（排序）

排序将其输入信息读入内存中，在内存中对输入信息进行排序，然后输出结果。如果输入的信息不能完全装入到内存中，则将按顺序分多个步骤运行，然后将它们合并在一起。排序不返回任何行，直到它读取了所有的输入行。排序会锁定其输入行。

如果排序的运行环境中的可用高速缓存极少，该算法可能会无法完成。在这种情况下，排序将使用基于索引的排序方法对其余输入进行排序。它会读取输入行并将其插入到工作表中，而且会在工作表的排序列上建立索引。在这种情况下，将使用复杂的索引扫描从工作表中读取行。这种基于索引的策略要慢很多。如果优化程序检测到在查询执行过程中可能会出现内存不足的情况，它将避免使用排序生成访问计划。当由于内存不足而需要使用基于索引的策略时，性能计数器将递增；可以使

用 QueryLowMemoryStrategy 属性或者在 Windows 性能监控器的 [查询：内存不足策略] 计数器中读取此监控器。

排序运算符可以使用的内存量取决于服务器的进程并发水平和活动连接数。请参见“SQL Anywhere 中的线程”一节 《SQL Anywhere 服务器 - 数据库管理》和“设置数据库服务器的进程并发水平”一节 《SQL Anywhere 服务器 - 数据库管理》。

排序关键字的大小、行大小和输入信息的总大小都会影响排序性能。对于较大的行，使用 VALUES SENSITIVE 游标的开销可能会低一些。在这种情况下，SELECT 列表中的行不会被复制到排序所使用的工作表中。虽然排序不会将输出行写到工作表中，但是必须在实现排序的结果后，才能将行返回到应用程序。如果必要，优化程序将添加一个工作表以确保实现该目标。

SortTopN 算法 (SrtN)

SortTopN 用于包含 TOP N 子句和 ORDER BY 子句的查询。对于只为结果集中需要的那些行进行排序，这是一种有效的算法。

子查询和函数高速缓存

SQL Anywhere 在处理子查询时会高速缓存结果。这种高速缓存是一个请求一个请求地执行的；并发请求或多个连接决不会共享已高速缓存的结果。如果 SQL Anywhere 需要为同一组相关值重新计算子查询，它就可以直接从高速缓存中检索结果。如此一来，SQL Anywhere 就可以避免许多重复和多余的计算。当请求完成（查询的游标关闭）时，SQL Anywhere 就会释放被高速缓存的值。

在进行查询处理时，SQL Anywhere 会监视被高速缓存的子查询值的重用频率。如果相关变量的值极少重复，SQL Anywhere 就只需要计算一次大多数的值。在这种情况下，SQL Anywhere 会发现重新计算偶尔出现的重复值比高速缓存大量只出现一次的条目更为有效。因此，对于语句的其余部分，数据库服务器会暂停对该子查询进行高速缓存，而为外部查询块中的每个行重新计算该子查询。

如果相关列的大小大于 255 字节，SQL Anywhere 也不会进行高速缓存。在这种情况下，您可能需要重写查询或者将另一列添加到表中，以提高这些操作的效率。

函数高速缓存

一些内置函数和用户定义函数的高速缓存方式与子查询结果的高速缓存方式相同。这样就可以大大改进那些在查询处理过程中用相同参数调用的且占用大量资源的函数。但是，这可能意味着函数的调用次数少于其它情况下的预期调用次数。

对于要高速缓存的函数，它必须满足两个条件：

- 对于一组给定的参数，它必须始终返回相同的结果。
- 它必须对基础数据没有副面影响。

满足这些条件的函数称为**确定型**或**等幂**函数。SQL Anywhere 将所有用户定义的函数视为确定型（除非在创建时将它们声明为 NOT DETERMINISTIC）。即，数据库服务器假定对具有相同参数的同一函数连续进行两次调用将返回相同的结果，并且不会对查询的语义产生任何不良的副作用。

除了少数几个例外，内置函数均被视为确定型函数。RAND、NEW_ID 和 GET_IDENTITY 函数被视为非确定型函数，它们的结果不进行高速缓存。

有关用户定义函数的详细信息，请参见“CREATE FUNCTION 语句（Web 服务）”一节 《SQL Anywhere 服务器 - SQL 参考》。

其它算法

下面是可以在访问计划中使用的其它算法。

DecodePostings (DP)

文本索引存储在表的压缩块中。DecodePostings 解码文本索引中术语的位置信息。

另请参见“全文搜索”一节第 284 页。

DerivedTable 算法 (DT)

派生表是包含在查询的 FROM 子句中的 SELECT 语句。SELECT 语句的结果集在逻辑上被当作表来处理。查询优化程序也可能在查询重写过程中生成派生表，例如在包含基于集合的运算 UNION、INTERSECT、或 EXCEPT 的查询中。图形式计划显示派生表的名称和计算列的列表。

派生表体现了在不更改查询结果的情况下，访问计划中不能合并或展平到语句访问计划的其它部分中的部分。派生表用于实施在初始语句中指定的派生表语义，且由于查询重写优化和各种其它原因，尤其是当查询涉及一个或多个外连接时，其可能会出现在计划中。

有关派生表的详细信息，请参见“FROM 子句：指定表”一节第 264 页和“FROM 子句”一节《SQL Anywhere 服务器 - SQL 参考》。

示例

下面的查询在其图形式计划里有派生表：

```
SELECT EmployeeID FROM Employees
UNION ALL
SELECT DepartmentID FROM (
    SELECT TOP 5 DepartmentID
    FROM Departments
    ORDER BY DepartmentName DESC ) MyDerivedTable;
```

交换算法（交换）

交换用于在处理 SELECT 语句时实现查询内并行。交换运算符具有两个或多个子子树，每个都以并行方式执行。在每个子树执行时，其会填充随后由交换的父运算符使用的行缓冲区。交换的结果就是其子项的结果的联合。交换的每个子项都使用一个任务，父项也一样。因此，使用一个具有两个子项的交换的计划需要执行三个任务。

交换仅在处理 SELECT 语句且启用了查询内并行机制时使用。

有关并行机制的详细信息，请参见“SQL Anywhere 中的线程”一节 《SQL Anywhere 服务器 - 数据库管理》。

过滤器算法（过滤器、PreFilter）

过滤器应用包括任何类型谓语、涉及子选择的比较、EXISTS 和 NOT EXISTS 子查询（以及其它形式的限定子查询）的搜索条件。搜索条件出现在语句的 WHERE 子句和 HAVING 子句中，还出现在 FROM 子句内 JOINS 的 ON 条件中。

优化程序可在其认为合适的情况下任意简化和更改搜索条件中的谓语集，并可构建访问计划，该计划按照与初始语句中指定的顺序不同的顺序来应用条件。查询重写优化可能会对计划中计算的谓语集进行大量的更改。

在许多情况下，查询中的谓语可能不会导致在访问计划中存在过滤器。例如，诸如 IndexScan 等各种算法，它们可以强制应用谓语，无需使用显式运算符。例如，请考虑涉及两个文字常量的 BETWEEN 谓语，且谓语中引用的列已建立索引。BETWEEN 谓语可以由索引扫描的上限和下限强制实施，而查询计划中将不会包含过滤器。属于连接条件的谓语通常也不会作为过滤器出现在访问计划中。

PreFilter 与过滤器基本相同，只是在 PreFilter 的谓语中所使用的表达式不依赖于查询中引用的任何表或视图。举一个简单的例子，`WHERE 1 = 2` 子句中的搜索条件即可作为一个预过滤器进行计算。

另请参见

- “WHERE 子句：指定行”一节第 266 页
- “EXISTS 搜索条件”一节 《SQL Anywhere 服务器 - SQL 参考》

散列过滤器算法（HF、HFP）

散列过滤器（有时称为 bloom 过滤器）是一种表示列或列集中值的分布的数据结构。散列过滤器可被视为（长）位字符串，其中 1 位指示存在某一特定行，而 0 位指示在此位的位置缺少任何行。通过将一组行中的值散列到过滤器的各个位的位置，数据库服务器可以确定是否存在与该值匹配的行（可能存在散列冲突）。

例如，计划为：

```
R<idx> *JH S<seq> JH* T<idx>
```

此处，将 R 连接到 S 和 T。数据库服务器先读取 R 的所有行，然后再从 T 读取任意行。如果散列过滤器是使用由索引扫描返回的 R 中的行构建的，则数据库服务器可以立即拒绝可能无法与 R 连接的 T 中的行。这将减少必须存储在第二个散列连接中的行数。

散列过滤器可以在同时满足以下条件的查询中使用：

- 查询中的操作在向以后的操作返回行之前会读取它的整个输入。例如，某一列上两个表的散列连接需要从其中的一个输入读取所有相关行，以构成连接的散列表。
- 查询访问计划中的后续操作会引用该操作结果中的行。例如，第一个连接所在列上的另一个连接只使用满足第一个连接的那些行。

在这种情况下，作为第一个连接的结果构建的散列过滤器可显著地提高第二个连接的性能。其实现方法是：在散列过滤器的位字符串中执行后备查询操作，以确定第一个连接是否先前已成功处理过某行—如果不存在这样的行，则可完全避免为第二个连接进行散列表探测，因为散列过滤器中不存在 1 位即表示探测将无法产生匹配项。

InList 算法 (IN)

当可以使用索引来满足 IN 列表谓语时，将使用 InList。例如，在以下查询中，优化程序发现它可以使用 Employees 表的主键索引来对该表进行访问。

```
SELECT *
FROM Employees
WHERE EmployeeID IN ( 102, 105, 129 );
```

为了完成这一任务，使用特殊的 IN 列表表在左侧建立了一个连接。行读取自 IN 列表表，并用于探查 Employees 表。

要使用 InList，IN 列表谓语中的每个元素必须是常量，或者是可在优化时经计算得到常量值的某个值（如 CURRENT DATE、CURRENT TIMESTAMP 以及非确定性系统函数和用户定义的函数），或者是在查询块的一次执行中为常量的某个值（外部引用）。例如，下面的查询满足 InList 的条件。

```
SELECT *, (
   SELECT FIRST GivenName
    FROM Employees e
    WHERE e.DepartmentID IN ( 500, d.DepartmentID )
    ORDER BY e.DepartmentID )
FROM Departments d;
```

多个 IN 列表谓语可以使用同一索引来得到满足。

OpenString 算法 (OpenString)

当 SELECT 语句的 FROM 子句包含 OPENSTRING 子句时，可以使用 OpenString。从 BLOB 或在 OPENSTRING 子句中指定的文件读取行。请参见“FROM 子句”一节 《SQL Anywhere 服务器 - SQL 参考》。

OpenString 运算符还出现在 LOAD TABLE 语句的计划中。

ProcCall 算法 (PC)

ProcCall（用于 FROM 子句中的过程）执行过程调用并返回其结果集中的行。它不能进行向后读取，因此如果游标类型要求进行向后读取，则它会出现在工作表之下。

每次调用 ProcCall 时，数据库服务器都会记下参数值、返回的行数以及用于读取所有行的总时间。优化程序会使用此信息来估计开销和后续过程调用的基数。数据库服务器为每个过程维护一个返回行数的移动平均值和一个总执行时间的移动平均值。它还为特定参数值维护有限数量的单独移动平均值。此信息以二进制格式永久地存储在 SYSPROCEDURE 系统表的统计信息列中，只供内部使用。

有关多个结果集的限制及模式匹配要求的信息，请参见 FROM 子句的过程子句，“FROM 子句”一节 《SQL Anywhere 服务器 - SQL 参考》。

另请参见

- “SYSPROCEDURE 系统视图”一节 《SQL Anywhere 服务器 - SQL 参考》
- “优化程序使用过程统计信息的方式”一节第 527 页

RowConstructor 算法 (ROWS)

RowConstructor 是一种专用运算符，可以创建用作其它算法的输入的虚拟行。RowConstructor 有以下两种使用方式：

- 对于 INSERT ...VALUES 语句，在 VALUES 子句中引用的表达式（通常为文字常量和/或主机变量）构成了要插入的虚拟行。在这种情况下，行构造函数将出现在图形式计划中 INSERT 之下。
- 对系统表 SYS.DUMMY 的直接或间接引用将被自动转换以使用 RowConstructor，从而取代 SYS.DUMMY 的表扫描的需要并无需锁定 DUMMY 表的单个页。

对于简单或详细文本计划，计划字符串继续包含对表 SYS.DUMMY 的引用，即使已经使用了 RowConstructor，而不是执行 SYS.DUMMY 的表扫描。

另请参见

- “DUMMY 系统表”一节 《SQL Anywhere 服务器 - SQL 参考》
- “INSERT 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “读取执行计划”一节第 569 页

RowLimit 算法 (RL)

RowLimit 返回其输入的前 *n* 行并忽略其余的行。行限制是通过 SELECT 语句的 TOP n 或 FIRST 子句来设置的。请参见“SELECT 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

Termbreaker 算法 (TermBreak)

术语断开算法用于全文搜索。请参见“文本配置对象”一节第 285 页中有关如何使用术语断开器的说明。

窗口算法 (Window)

计算使用窗口函数的 OLAP 查询时会使用窗口。请参见“窗口函数”一节第 436 页。

读取执行计划

执行计划是数据库服务器用来访问数据库中与语句相关的信息的步骤集。无论语句的执行计划是否刚经过优化、是否跳过了优化程序，也无论其计划是否是从先前的执行进行的高速缓存，均可进行保存和查看。查询执行计划可能不会与初始语句中所使用的语法完全对应，查询执行计划可使用实例化视图而非查询中明确指定的基表。但执行计划中描述的操作在语义上等同于初始查询。

您可以在 Interactive SQL 中查看执行计划，也可以使用 SQL 函数查看执行计划。可选择检索以下几种不同格式的执行计划：

- 简要文本计划
- 详细文本计划
- 图形式计划
- 含基本统计信息的图形式计划
- 含完整统计信息的图形式计划
- UltraLite（简单、详细或图形）

还可以通过使用 GRAPHICAL_PLAN 和 EXPLANATION 函数并利用特定的游标类型来获取 SQL 查询的计划。请参见“GRAPHICAL_PLAN 函数 [Miscellaneous]”一节 《SQL Anywhere 服务器 - SQL 参考》和“EXPLANATION 函数 [Miscellaneous]”一节 《SQL Anywhere 服务器 - SQL 参考》。

补充阅读内容

有关语句在其执行之前要经历的阶段的详细信息，请参见“查询处理阶段”一节第 510 页。

有关数据库服务器在重写查询时所遵循的规则的详细信息，请参见：

- “语义查询转换”一节第 513 页
- “将子查询重写为 EXISTS 谓语”一节第 521 页
- “使用实例化视图提高性能”一节第 536 页

有关优化程序用来实现查询的算法和方法的信息，请参见“[查询执行] 算法”一节第 548 页。

有关如何访问图形式计划的详细信息，请参见“查看图形式计划”一节第 579 页。

有关如何读取执行计划的信息，请参见“读取文本计划”一节第 569 页和“读取图形式计划”一节第 571 页。

读取文本计划

查询执行计划的文本表示形式有两种：简要计划和详细计划。使用 SQL 函数访问文本计划。请参见“查看简要文本计划和详细文本计划”一节第 571 页。

计划也有图形式版本。请参见“读取图形式计划”一节第 571 页。

简要文本计划

当您要快速比较计划时，简要文本计划将非常有用。在所有访问计划格式中，它所提供的信息量是最少的，但它在一行上提供这些信息。

在下面的示例中，由于 ORDER BY 子句使整个结果集进行排序，因此计划以 `Work[Sort` 开头。对 Customers 表的访问是通过它的主键索引 CustomersKey 进行的。因为列 Customers.ID 是主键，所以将使用索引扫描来满足搜索条件。缩写 JNL 指示优化程序选择合并连接来处理 Customers 和 SalesOrders 之间的连接。最后，使用外键索引 FK_CustomerID_ID 对 SalesOrders 表进行访问，以在 Customers 表中查找 CustomerID 小于 100 的行。

```
SELECT EXPLANATION ('SELECT GivenName, Surname, OrderDate
FROM Customers JOIN SalesOrders
WHERE CustomerID < 100
ORDER BY OrderDate');
```

```
Work[ Sort[ Customers<CustomersKey> JNL
SalesOrders<FK_CustomerID_ID> ] ]
```

有关计划中使用的代码字的详细信息，请参见“执行计划缩写”一节第 583 页。

用冒号来分隔连接策略

下面的命令包含两个**查询块**：引用 SalesOrders 和 SalesOrderItems 表的外部选择块和从 Products 表选择的子查询块。

```
SELECT EXPLANATION ('SELECT *
FROM SalesOrders AS o
   KEY JOIN SalesOrderItems AS I
WHERE EXISTS
   (  SELECT *
      FROM Products p
      WHERE p.ID = 300 )');
```

```
o<seq> JNL i<FK_ID_ID> : p<ProductsKey>
```

用冒号将不同查询块的连接策略分开。简要计划始终会首先列出主块的连接策略，然后才列出其它查询块的连接策略。其它查询块的连接策略的顺序可能与语句中查询块的顺序或者查询块的执行顺序不对应。

有关计划中使用的缩写的详细信息，请参见“执行计划缩写”一节第 583 页。

详细文本计划

详细文本计划提供的信息略多于简要文本计划，这些信息以便于打印和查看（无需滚动）的方式提供。

在下面的示例中，详细文本计划的第一行为 `Plan[ Total Cost Estimate: 6.46e-005 ]`，其中 Plan 一词表示查询块的开始。Total Cost Estimate 为优化程序估计的计划执行时间（以毫秒为单位）。Estimated Cache Pages 是预计的可用于处理语句的当前高速缓存大小。

该计划指出，将对结果进行排序，并且会使用嵌套循环连接。在连接运算符所在的行上，还包含 TRUE 一词或其它搜索条件及其选择性估计值（其为对连接运算符所生成的所有行的计算）。

IndexScan 行指示分别通过索引 CustomersKey 和 FK_CustomerId_ID 访问 Customers 和 SalesOrders 表。

```
SELECT PLAN ('SELECT GivenName, Surname, OrderDate, Region, Country
FROM Customers JOIN SalesOrders ON ( SalesOrders.CustomerID = Customers.ID )
WHERE CustomerID < 100 AND ( Region LIKE ''Eastern''
      OR Country LIKE ''Canada'' )
ORDER BY OrderDate');

( Plan [ Total Cost Estimate: 6.46e-005, Costed Best Plans: 1, Costed Plans:
10, Optimization Time: 0.0011462,
Estimated Cache Pages: 348 ]
  ( WorkTable
    ( Sort
      ( NestedLoopsJoin
       ( IndexScan Customers CustomersKey[ Customers.ID < 100 : 0.0001% Index
| Bounded ] )
        ( IndexScan SalesOrders FK_CustomerID_ID[ Customers.ID =
SalesOrders.CustomerID : 0.79365% Statistics ]
          [  ( SalesOrders.CustomerID < 100 : 0.0001% Index | Bounded )
          AND  ( ( ((Customers.Country LIKE 'Canada' : 100% Computed)
          AND (Customers.Country = 'Canada' : 5% Guess))
          OR ((SalesOrders.Region LIKE 'Eastern' : 100% Computed)
          AND (SalesOrders.Region = 'Eastern' : 5% Guess)) ) : 100%
Guess )  ] )
      )
    )
  )
)
```

有关计划中使用的缩写的详细信息，请参见“执行计划缩写”一节第 583 页。

查看简要文本计划和详细文本计划

♦ 查看简要文本计划 (SQL)

1. 以具有 DBA 权限的用户身份连接到数据库。
2. 执行 EXPLANATION 函数。请参见“EXPLANATION 函数 [Miscellaneous]”一节 《SQL Anywhere 服务器 - SQL 参考》。

♦ 查看详细文本计划 (SQL)

1. 以具有 DBA 权限的用户身份连接到数据库。
2. 执行 PLAN 函数。请参见“PLAN 函数 [Miscellaneous]”一节 《SQL Anywhere 服务器 - SQL 参考》。

读取图形式计划

Interactive SQL 中的图形式计划功能会在 [**计划查看器**] 窗口中显示查询的执行计划。执行计划包括一个由关系代数运算符所组成的树，这些运算符从树叶开始，它们使用查询的基本输入（通常为表中的各行）并自下而上对各个行进行处理，以便树根生成最终结果。树中节点对应特定的代数运算

符，但并不是服务器所执行的所有查询运算都由节点表示。例如，在图形式计划中不直接显示子查询和函数高速缓存的影响。

在图形式计划中显示的节点具有不同的形状，表示所执行的操作的类型：

- 六角形表示实例化数据的操作。
- 梯形表示索引扫描。
- 直角矩形表示表扫描。
- 圆角矩形表示未在上面列出的操作。

可以使用图形式计划来诊断特定查询的性能问题。例如，计划中的信息可以帮助您确定表是否需要索引来提高此特定查询的性能。按下 [**计划查看器**] 中的 [**保存**] 按钮，可保存查询的图形式计划以备将来引用。SQL Anywhere 图形式计划以扩展名 *.saplan* 保存。

在图形式计划中，用粗线条和红色边框来标识可能出现的性能问题。例如：

- 计划中节点间线条加粗表明已处理行的数量相应增加。如果在表扫描上出现粗线条，则可能表示需要创建索引。
- 节点周围的红色边框表示与执行计划中其它操作相比，此操作开销较大。

可在 Interactive SQL 中自定义计划的节点形状和其它图形组件。请参见“自定义图形式计划的外观”一节第 578 页。

可查看图形式计划、含汇总信息的图形式计划或含详细统计信息的图形式计划。所有这三种计划都允许查看计划中估计开销最大的部分。生成含统计信息的图形式计划开销更大，因为它提供了执行查询时数据库服务器监控的实际查询执行统计信息。在构建访问计划时查询优化程序会使用一些估计值，含统计信息的图形式计划允许将这些估计值与执行时实际监控到的统计信息直接进行比较。但要注意，优化程序通常无法精确估计查询的开销，因此估计值和实际值之间会有差异。

要查看图形式计划，请参见“查看图形式计划”一节第 579 页。还可使用 Sybase Central 中的 [应用程序分析] 模式来获得图形式计划。有关 Sybase Central 的应用程序分析功能的详细信息，请参见“应用程序分析”一节第 161 页。

有关文本计划的详细信息，请参见“读取文本计划”一节第 569 页。

含统计信息的图形式计划

图形式计划比简要文本计划或详细文本计划能提供更多信息。尽管含统计信息的图形式计划的生成会占用较多的资源，但它能提供在查询执行过程中数据库服务器监测到的实际查询执行统计信息，并且允许优化程序将构建访问计划时所使用的预计值和在执行过程中监测到的实际统计信息加以比较。若实际统计信息和估计统计信息之间差异明显，则表示优化程序可能没有获得足够的信息来正确估计查询的开销，从而生成了一个低效的执行计划。

要生成含统计信息的图形式计划，数据库服务器必须执行语句。对于运行时间长的语句，生成图形式计划可能会占用大量的时间。如果语句是 UPDATE、INSERT 或 DELETE，只执行语句的只读部分；不执行表的修改。但是，如果语句包含用户定义的函数，这些函数会作为查询的一部分执行。如果用户定义的函数有副作用（例如，修改行、创建表、向控制台发送消息等等），在生成含统计信息的图形式计划时会进行这些更改。在某些情况下，可以在获得含统计信息的图形式计划后发

出 ROLLBACK 语句来撤消这些副作用。请参见“ROLLBACK 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

使用含统计信息的图形式计划分析性能

可使用含统计信息的图形式计划来识别数据库性能问题。有关含统计信息的图形式计划的详细字段说明，请参见“节点统计字段说明”一节第 579 页和“优化程序统计字段说明”一节第 581 页。

识别查询执行问题

可为根运算符节点显示影响查询执行的数据库选项和其它全局设置。

查看选择性性能

谓语的选择性（条件表达式）是指满足条件的行的百分比。估计的谓语的选择性为优化程序估计其开销提供了依据信息。准确的选择性估计值对于优化程序的正确运行至关重要。例如，如果优化程序错误地将某一谓语的选择性估计得很高（例如 5% 的选择性），但实际该谓语的选择性要低很多（例如 50%），那么性能就可能会受到损害。尽管选择性估计值可能不精确，但如果有较大偏差，则表明存在问题。

如果您确定查询主要部分的选择性信息不准确，则可以使用 CREATE STATISTICS 为列生成一组新的统计信息。在极少数情况下，您可能希望提供显式选择性估计，尽管这种方法会在以后更新统计信息时引发问题。

如果确定查询是跳过查询，则不会显示选择性统计信息。有关跳过查询的详细信息，请参见“优化程序的工作原理”一节第 525 页和“显式选择性估计”一节 《SQL Anywhere 服务器 - SQL 参考》。

选择性低的指示符在以下位置出现：

- **RowsReturned，实际值和估计值** **RowsReturned** 是结果集中的行数。**RowsReturned** 统计信息显示在树顶部根节点的表中。如果估计行数和实际行数之间存在很大的差异，则附加在此节点或子树上的谓语选择性可能不正确。
- **谓语选择性，实际值和估计值** 查找 [**谓语**] 小标题以查看谓语选择性。有关读取谓语信息的信息，请参见“在图形式计划中查看选择性”一节第 576 页。

  如果谓语位于没有直方图的基列之上，则执行 CREATE STATISTICS 语句创建直方图可能会更正问题。请参见“CREATE STATISTICS 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

  如果选择性偏差仍然没有得到更正，则可以考虑随查询文本中的谓语一起指定选择性的用户估计值。
- **估计值来源** [**统计信息**] 窗格中的 [谓语] 小标题下还列出了选择性估计值的来源。

  当谓语的选择性估计值来源是 [**推测**] 时，优化程序将没有可用的信息来确定谓语的过滤特性，这可能表示存在问题（比如缺少直方图）。如果估计值来源是 [**索引**] 并且选择性估计值不正确，则问题可能是索引有偏差；使用 REORGANIZE TABLE 语句整理索引碎片可能会有用。请参见“REORGANIZE TABLE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

查看高速缓存性能

如果高速缓存读取数（[**CacheRead**] 字段）和高速缓存命中数（[**CacheHits**] 字段）相同，则说明针对此 SQL 语句处理的所有对象都驻留在高速缓存中。若高速缓存读取数大于高速缓存命中数，则表明数据库正从磁盘读取表或索引页，因为这些表或索引页没有驻留在服务器的高速缓存中。在某些情况下（如散列连接），这正是您所需要的结果。而在另外一些情况下（如嵌套循环连接），较低的高速缓存命中率可能说明没有足够的高速缓存（缓冲池）来保证查询高效地执行。在这种情况下，提高服务器的高速缓存大小可能会有用。

有关高速缓存管理的详细信息，请参见“增加高速缓存大小”一节第 210 页。

识别无效的索引

通常很难从查询执行计划中看出索引能否帮助提高性能。SQL Anywhere 中使用的一些基于扫描的算法可在不使用索引的情况下为许多查询提供极佳的性能。

有关索引和性能的详细信息，请参见“有效使用索引”一节第 220 页和“索引顾问”一节第 167 页。

识别数据碎片问题

[**运行时**] 和 [**FirstRowRunTime**] 的实际值与估计值在根节点统计信息中提供。如果节点存在 [**运行时**] 值，则只有这个值会显示在 [**子树统计**] 部分。

对 [**运行时**] 的解释取决于它出现在的统计信息部分。在 [**节点统计**] 中，[**运行时**] 是在*只对此节点*执行期间对应运算符所花费的累积时间。在 [**子树统计**] 中，[**运行时**] 表示针对此节点下紧邻的整个运算符子树所花费的总执行时间。因此，对大多数运算符而言，[**运行时**] 和 [**FirstRowRunTime**] 是相互独立的测量值，应单独分析。

[**FirstRowRunTime**] 值是生成此节点中间结果的第一行所需要的时间。

如果表扫描或索引扫描的 [**运行时**] 值高于预期值，可通过执行 REORGANIZE TABLE 语句来提高性能。可使用 sa_table_fragmentation() 和 sa_index_density() 系统过程来确定表或索引是否产生了碎片。

有关详细信息，请参见“REORGANIZE TABLE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》和“减少表碎片”一节第 214 页。

有关计划中使用的代码字的详细信息，请参见“执行计划缩写”一节第 583 页。

查看详细的图形式计划节点信息

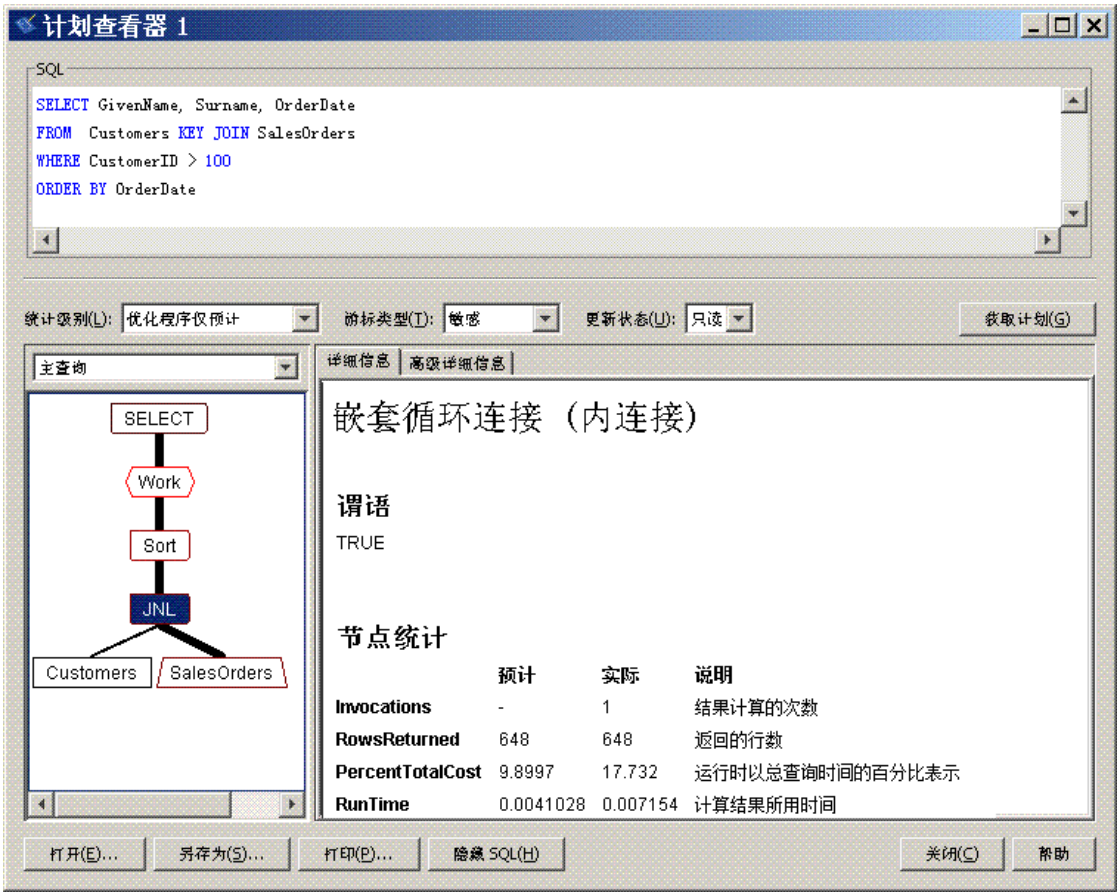
如果想查看图形式计划中的详细节点信息，请在左窗格中单击图形式图表中的节点。关于节点的详细信息会显示在 [**详细信息**] 和 [**高级详细信息**] 窗格的右侧。在 [**详细信息**] 窗格中，关于节点的统计信息分三部分显示：

- **节点统计**
- **子树统计**
- **优化程序统计**

节点统计是指与特定节点的执行有关的统计信息。叶节点有一个 [**详细信息**] 窗格，其中显示了某个运算符的估计统计信息和实际统计信息。当叶节点显示在父节点的右侧时，可多次从父运算符中提取行。例如，嵌套循环连接的叶节点（顺序扫描、索引扫描或 RowID 扫描节点）既包含每次调用时的运行时间统计（平均），也包含累积的实际运行时间统计。

如果节点不是叶节点，那么这个节点会使用其它节点的中间结果，[**详细信息**] 窗格在 [**子树统计**] 部分显示此节点整个子树的估计统计信息和实际的累积统计信息。代表整个 SQL 请求的优化程序统计信息只提供给根节点。优化程序统计值与语句的优化密切相关，其中包括的值有优化目标设置、优化级别设置、需要考虑的计划数等等。

在下面所示的例子中，选择了 [**嵌套循环连接 (JNL)**] 节点，右窗格中显示的信息只与该节点相关。例如，[**谓语**] 说明是 [TRUE]，这说明没有使用谓词。如果单击 [**Customers**] 节点，[**谓语**] 值则更改为 [Customers.ID > 100 : 100% Index; true 126/126 100%.]



在 [**高级详细信息**] 窗格中显示的信息与特定的运算符有关。对于根节点，[**高级详细信息**] 窗格包含优化查询时生效的所有连接选项设置。对于其它节点类型，[**高级详细信息**] 窗格可能包括关于处理特定节点时考虑哪个索引或实例化视图方面的信息。

若要获取有关图形式计划中每个节点的上下文相关帮助，请右击节点，然后选择 [**帮助**]。

有关计划中使用的缩写的详细信息，请参见“执行计划缩写”一节第 583 页。

注意
如果一个查询被识别为跳过查询，则会跳过某些优化步骤，并且 [**查询优化程序**] 部分和 [**谓语**] 部分都不会显示在图形式计划中。有关所跳过的查询的详细信息，请参见“优化程序的工作原理”一节第 525 页。

另请参见

- “读取图形式计划”一节第 571 页
- “查看图形式计划”一节第 579 页
- “读取执行计划”一节第 569 页
- “节点统计字段说明”一节第 579 页
- “优化程序统计字段说明”一节第 581 页

在图形式计划中查看选择性

在下面显示的示例中，所选择的节点代表 Departments 表的扫描，统计信息窗格显示**谓语**（作为搜索条件）、其选择性估计值及其实际选择性。

在 [**详细信息**] 窗格中，有关单个节点的统计信息分为三个部分：[**节点统计**]、[**子树统计**] 和 [**优化程序统计**]。

节点统计与此特定节点的执行有关。如果节点不是计划中的叶节点，并因此使用其它节点的中间结果，那么 [**详细信息**] 窗格将显示 [**子树统计**] 部分，它包含了此节点整个子树的估计和实际累积统计信息。优化程序统计信息只针对根节点出现，它代表整个 SQL 请求。

也许不会为跳过查询显示选择性信息。有关跳过查询的详细信息，请参见“优化程序的工作原理”一节第 525 页。

访问计划取决于数据库中可用的统计信息，而统计信息又取决于以前所执行的查询。在下面显示的内容中，您可以看到不同的统计信息和计划。

此谓语说明是

```
Departments.DepartmentName = 'Sales' : 20% Column; true 1/5 20%
```

它可以按如下方式理解：

- `Departments.DepartmentName = 'Sales'` 是谓语。
- `20%` 是优化程序的选择性估计值。也就是说，优化程序的查询访问选择将基于该估计值（20%的行满足该谓语）。

  这一输出与 ESTIMATE 函数提供的输出相同。有关详细信息，请参见“ESTIMATE 函数 [Miscellaneous]”一节 《SQL Anywhere 服务器 - SQL 参考》。
- `Column` 是估计值来源。这一输出与 ESTIMATE_SOURCE 函数提供的输出相同。有关选择性估计值的可能来源的完整列表，请参见“ESTIMATE_SOURCE 函数 [杂类]”一节 《SQL Anywhere 服务器 - SQL 参考》。
- `true 1/5 20%` 是执行期间谓语的实际选择性。对谓语进行了五次计算，其中有一次为真，因此其实际选择性是 20%。

如果实际选择性与估计值有很大差异，且对谓语进行了许多次计算，则不正确的估计值可能会导致查询性能出现重大问题。收集有关谓语的统计信息可为优化程序提供有关确定选择依据的更佳信息，从而可以提高性能。

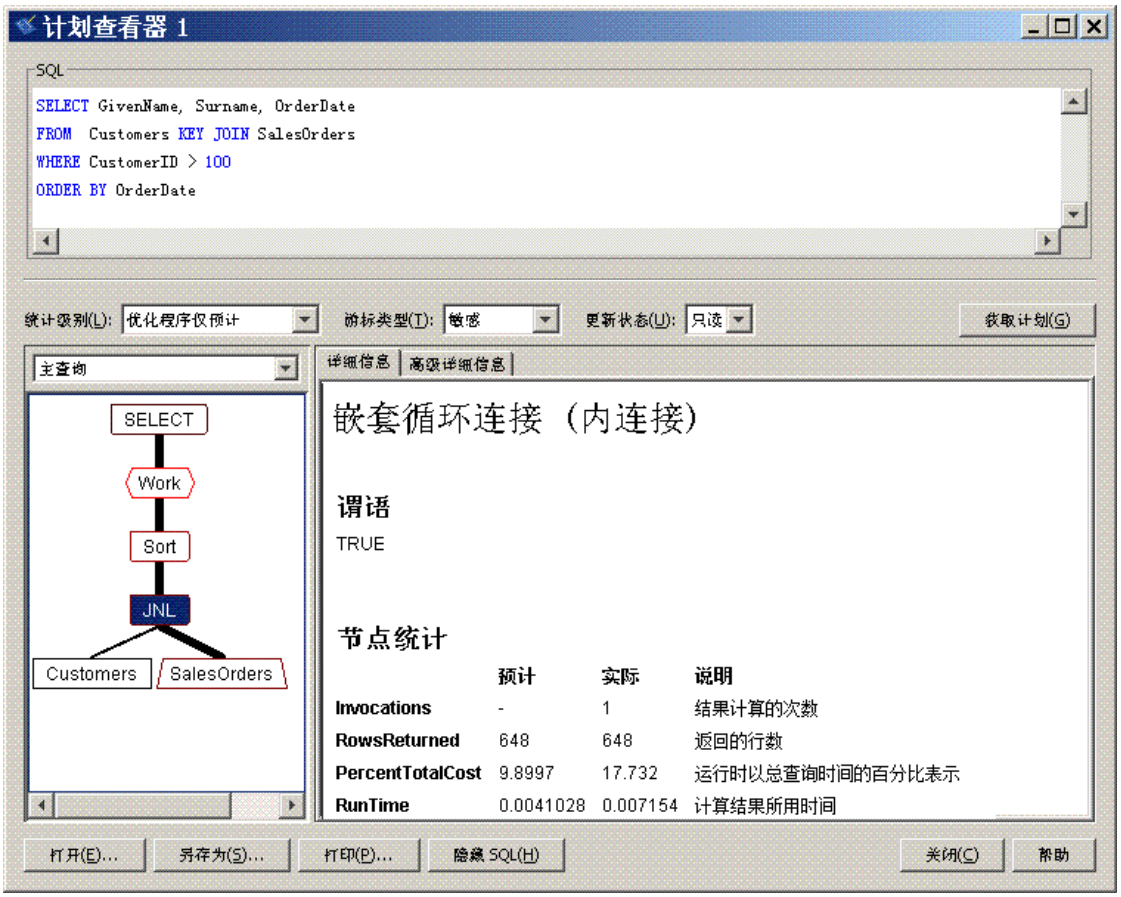
注意
如果您选择的是图形式计划而不是含统计信息的图形式计划，则最后两项统计信息不会显示出来。

自定义图形式计划的外观

在执行图形式计划后，可以自定义计划中各项的外观。要改变图形式计划的外观，请在 Interactive SQL 计划查看器左下角的窗格中右击该计划，选择 [**自定义**]，并更改设置。您的更改会应用到随后显示的图形式计划中。

要打印图形式计划，右击计划并选择 [**打印**]。

下面是提供的查询及其相应的图形式计划。以下图示采用了树的形式，表示每个节点从它下面的节点请求行。



查看图形式计划

可使用 Interactive SQL 或 GRAPHICAL_PLAN 函数查看图形式计划。要访问文本计划，请参见“读取文本计划”一节第 569 页。

查看图形式计划

- **查看图形式计划 (Interactive SQL)**

1. 启动 Interactive SQL 并连接到 SQL Anywhere 数据库。
2. 在 [**SQL 语句**] 窗格中键入语句。
3. 选择 [**工具**] » [**计划查看器**]。
4. 选择统计级别、游标类型和更新状态，然后单击 [**获取计划**]。

- **查看图形式计划 (SQL)**

使用 GRAPHICAL_PLAN 函数以字符串的形式查看 XML 格式的图形式计划。

1. 以具有 DBA 权限的用户身份连接到数据库。
2. 执行 GRAPHICAL_PLAN 函数。请参见“GRAPHICAL_PLAN 函数 [Miscellaneous]”一节《SQL Anywhere 服务器 - SQL 参考》。

另请参见：

- “执行计划缩写”一节第 583 页

节点统计字段说明

下面是在图形式计划的 [**节点统计**] 部分显示的字段的说明。

字段	说明
CacheHits	此运算符发出的高速缓存读取请求（不需要进行磁盘读取操作的缓冲池满足了这些请求）的总数。
CacheRead	为读取数据库文件的页（通常为表页和/或索引页），此运算符所进行尝试的总次数。
CPUTime	此节点代表的处理算法所需的 CPU 时间。
DiskRead	由于此节点的处理已从磁盘读取的累积页数。
DiskReadTime	执行数据库页（此节点处理时需要）的磁盘读取所需要的累积运行时间。
DiskWrite	由于此节点的处理已写入磁盘的累积页数。

字段	说明
DiskWriteTime	执行数据库页（此节点的处理算法需要）的磁盘写入所需要的累积运行时间。
FirstRowRunTime	**FirstRowRunTime** 值是生成此节点中间结果的第一行所需要的实际时间。
Invocations	调用节点来计算一个结果并将结果返回到父节点的次数。多数节点只被调用一次。但是，如果扫描节点的父节点是一个嵌套循环连接，那么该节点可能被执行多次，并且每次调用后可能返回不同的行集。
PercentTotalCost	计算此特定节点内结果所花费的 [**运行时**]，表示为语句总运行时的百分比。
QueryMemMaxUseful	预期用于此特定运算符的估计查询内存量。如果该值与实际使用的查询内存量（报告为 [**实际**] 统计信息）差别较大，则表明查询优化程序所估计的结果集大小可能有问题。此估计错误的一个可能原因是谓语选择性估计值不准确或缺失。
RowsReturned	作为处理请求的结果返回到父节点的行数。**RowsReturned** 经常（但不是必须）与该节点所代表的（可能是派生的）对象中的行数相同。考虑表示基表扫描的叶节点。**RowsReturned** 值可能小于或大于表中的行数。如果在计算最终结果时父节点无法请求表中的所有行，则 **RowsReturned** 值较小。在某些情况下 **RowsReturned** 值可能较大，比如在 GROUP BY GROUPING SETS 查询中，父 Group By Hash Grouping Sets 节点需要通过输入进行多次传递以计算不同的组。 如果估计的返回行数与实际返回的行数存在较大差异，则可能表明优化程序正在处理选择性很低的信息。
运行时	该值是对挂钟时间的测量，包括输入/输出等待、行锁、表锁、内部服务器并发控制机制和实际运行时处理。对 [**运行时**] 的解释取决于它出现在的统计信息部分。在 [节点统计] 中，[**运行时**] 是只对此节点执行期间该节点的相应运算符所花费的累积时间。此项统计的估计值和实际值都显示在 [节点统计] 部分中。 如果节点的 [**运行时**] 大于表扫描或索引扫描的预期值，则进一步分析可能有助于查明问题。查询可能争用共享资源并因此阻塞；可使用 sa_locks() 系统过程监视阻塞的连接。作为另一个示例，磁盘上的数据库页布局可能不是最优，或者表产生了内部页碎片。可以通过执行 REORGANIZE TABLE 语句来提高性能。可使用 sa_table_fragmentation() 和 sa_index_density() 系统过程来确定表或索引是否产生了碎片。

优化程序统计字段说明

下面是在图形式计划的 [**优化程序统计**] 部分显示的字段的说明。[**优化程序统计**] 提供了有关数据库服务器状态和所选语句优化的信息。

字段	说明
优化方法	用于选择执行策略的算法。将返回下列值： ● 开销旁路 ● 简单开销旁路 ● 试探法旁路 ● 跳过然后优化 ● 已优化 ● 重新使用的 ● 重新使用的（简单）
成本最佳计划	当查询优化程序枚举不同的查询执行策略时，查询优化程序会跟踪它查找某个策略的次数，该策略的估计成本比在当前策略之前找到的最佳策略的成本更低。针对特定查询很难预测其发生频率，但是较低的数值表示查询优化程序的算法明显修整了搜索空间，并且通常表示优化时间更快。因为优化程序为给定语句的每个查询块至少启动一次枚举进程，因此 [**成本最佳计划**] 表示的是累积次数。请参见“优化程序的工作原理”一节第 525 页。 如果 [**成本最佳计划**]、[**成本计划**] 和 [**优化时间**] 的值为 0，则说明该语句没有被 SQL Anywhere 优化程序优化。相反，数据库服务器跳过了该语句，并生成了不优化该语句的执行计划，或该语句的计划已被高速缓存。请参见“查询处理阶段”一节第 510 页。
成本计划	优化程序考虑用于此请求的不同访问计划数，已部分或全部估计了此请求的成本。与 [**成本最佳计划**] 一样，较小的值通常表示更快的优化时间，较大的值表示更复杂的 SQL 请求。 如果 [**成本最佳计划**]、[**成本计划**] 和 [**优化时间**] 的值为 0，则说明该语句没有被优化。相反，数据库服务器跳过了该语句，并生成不优化该语句的执行计划。请参见“查询处理阶段”一节第 510 页。
优化时间	优化语句所用的时间。 如果 [**成本最佳计划**]、[**成本计划**] 和 [**优化时间**] 的值为 0，则说明该语句没有被优化。相反，数据库服务器跳过了该语句，并生成不优化该语句的执行计划。请参见“查询处理阶段”一节第 510 页。
预计高速缓存页	预计的可用于处理语句的当前高速缓存大小。 为减少低效率的访问计划，优化程序假定当前高速缓存大小的一半可用于处理所选语句。
CurrentCacheSize	优化时数据库服务器的高速缓存大小，以千字节为单位。

字段	说明
QueryMemMaxUseful	对此请求有用的查询内存页数。如果页数为零，则语句的执行计划不包含内存密集型运算符，并且不受服务器内存调控器的控制。请参见“内存调控器”一节第 530 页。
QueryMemNeedsGrant	表示内存调控器是否必须向存在于此请求的执行策略中的一个或多个内存密集型查询执行运算符授予内存。请参见“内存调控器”一节第 530 页。
QueryMemLikelyGrant	预计的查询内存池的页数，如果语句立即执行，会将该页数授予这个语句。此估计值随以下因素而变化：计划中内存密集型运算符的数量、数据库服务器的进程并发水平和并发执行的内存密集型请求的数量。请参见“内存调控器”一节第 530 页。
QueryMemPages	查询内存池中可用于所有连接的内存密集型查询执行算法的内存总量，以页数表示。请参见“内存调控器”一节第 530 页。
QueryMemActiveMax	在任何特定时间可主动使用查询内存的最大任务数。请参见“内存调控器”一节第 530 页。
QueryMemActiveEst	数据库服务器对主动使用查询内存的任务数的稳态平均值的估计。请参见“内存调控器”一节第 530 页。
isolation_level	语句的隔离级别。某个语句的隔离级别可能与同一事务中其它语句的隔离级别不同，并且可能会针对特定基表被进一步替换（通过在 FROM 子句中使用提示）。请参见“isolation_level 选项 [数据库] [兼容性]”一节 《SQL Anywhere 服务器 - 数据库管理》。
optimization_goal	表示优化查询处理的目的是为迅速返回第一行，还是为最大程度地降低返回整个结果集的开销。请参见“optimization_goal 选项 [数据库]”一节 《SQL Anywhere 服务器 - 数据库管理》。
optimization_level	控制查询优化程序为查找访问计划而消耗的资源量。请参见“optimization_level 选项 [数据库]”一节 《SQL Anywhere 服务器 - 数据库管理》。
optimization_workload	optimization_workload 设置的 [**Mixed**] 或 [**OLAP**] 值。请参见“optimization_workload 选项 [数据库]”一节 《SQL Anywhere 服务器 - 数据库管理》。
max_query_tasks	可用于单个查询的并行执行计划的最大任务数。请参见“max_query_tasks 选项 [数据库]”一节 《SQL Anywhere 服务器 - 数据库管理》。
user_estimates	控制是考虑还是忽略查询文本的单独谓词中指定的用户估计。请参见“user_estimates 选项 [数据库]”一节 《SQL Anywhere 服务器 - 数据库管理》。

执行计划缩写

以下是在执行计划中可看到的缩写。

简要文本计划	详细文本计划	其它信息
	成本最佳计划	优化程序为给定查询生成访问计划并估算访问计划的开销。在此过程中，当前最佳计划可能被估计开销更低的新最佳计划所取代。最后一个最佳计划即是用于执行语句的执行计划。成本最佳计划指出优化程序发现比当前最佳计划更好的计划的次数。次数较少说明最佳计划在枚举过程早期就已确定。由于优化程序为给定语句的每个查询块至少启动一次枚举过程，因此成本最佳计划表示的是累积次数。请参见“优化程序的工作原理”一节第 525 页。
	成本计划	与目前已找到的最佳计划相比，优化程序生成的许多计划开销都十分庞大。成本计划表示在给定语句的枚举过程期间，优化程序所考虑的部分或完整计划的数目。
DELETE	Delete	DELETE 操作的根节点。请参见“DELETE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。
DistH	HashDistinct	请参见“HashDistinct 算法 (DistH)”一节第 559 页。
DistO	OrderedDistinct	请参见“OrderedDistinct 算法 (DistO)”一节第 559 页。
DP	DecodePostings	请参见“DecodePostings (DP)”一节第 565 页。
DT	DerivedTable	请参见“DerivedTable 算法 (DT)”一节第 565 页。
EAH	HashExceptAll	请参见“排除算法（EAH、EAM、EH、EM）”一节第 561 页。
EAM	MergeExceptAll	请参见“排除算法（EAH、EAM、EH、EM）”一节第 561 页。
EH	HashExcept	请参见“排除算法（EAH、EAM、EH、EM）”一节第 561 页。
EM	MergeAccept	请参见“排除算法（EAH、EAM、EH、EM）”一节第 561 页。

简要文本计划	详细文本计划	其它信息
Exchange	Exchange	请参见“交换算法（交换）”一节第 565 页。
Filter	Filter	请参见“过滤器算法（过滤器、PreFilter）”一节第 566 页。
GrByH	HashGroupBy	请参见“HashGroupBy 算法 (GrByH)”一节第 560 页。
GrByHClust	HashGroupByClustered	请参见“ClusteredHashGroupBy 算法 (GrByHClust)”一节第 560 页。
GrByHSets	HashGroupBySets	请参见“HashGroupBySets 算法 (GrByHSets)”一节第 560 页。
GrByO	OrderedGroupBy	请参见“OrderedGroupBy 算法 (GrByO)”一节第 561 页。
GrByOSets	OrderedGroupBySets	请参见“OrderedGroupBySets 算法 (GrByOSets)”一节第 561 页。
GrByS	SingleRowGroupBy	请参见“SingleRowGroupBy 算法 (GrByS)”一节第 561 页。
GrBySSets	SortedGroupBySets	请参见“SortedGroupBySets 算法 (GrBySSets)”一节第 561 页。
HF	HashFilter	请参见“散列过滤器算法（HF、HFP）”一节第 566 页。
HFP	ParallelHashFilter	请参见“散列过滤器算法（HF、HFP）”一节第 566 页。
HTS	HashTableScan	请参见“HashTableScan 方法 (HTS)”一节第 553 页。
IAH	HashIntersectAll	请参见“交叉算法（IH、IM、IAH、IAM）”一节第 562 页。
IAM	MergeIntersectAll	请参见“交叉算法（IH、IM、IAH、IAM）”一节第 562 页。
IH	HashIntersect	请参见“交叉算法（IH、IM、IAH、IAM）”一节第 562 页。
IM	MergeIntersect	请参见“交叉算法（IH、IM、IAH、IAM）”一节第 562 页。

简要文本计划	详细文本计划	其它信息
IN	InList	请参见“InList 算法 (IN)”一节第 567 页。
table-name<index-name>	IndexScan、ParallelIndexScan	在图形式计划中，索引扫描显示为梯形中的索引名。请参见“IndexScan 方法”一节第 551 页。
INSENSITIVE	Insensitive	请参见“交叉算法（IH、IM、IAH、IAM）”一节第 562 页。
INSERT	Insert	插入操作的根节点。请参见“INSERT 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。
IO	IndexOnlyScan、ParallelIndexOnlyScan	请参见“IndexOnlyScan 方法 (IO)”一节第 552 页和“ParallelIndexScan 方法”一节第 552 页。
JH	HashJoin	请参见“HashJoin 算法（JH、JHSP、JHFO、JHAP、JHO、JHPO）”一节第 555 页。
JHS	HashSemijoin	请参见“HashSemijoin 算法 (JHS)”一节第 556 页。
JHSP	ParallelHashSemijoin	请参见“HashJoin 算法（JH、JHSP、JHFO、JHAP、JHO、JHPO）”一节第 555 页。
JHFO	Full Outer HashJoin	请参见“HashJoin 算法（JH、JHSP、JHFO、JHAP、JHO、JHPO）”一节第 555 页。
JHA	HashAntisemijoin	请参见“HashAntisemijoin 算法 (JHA)”一节第 557 页。
JHAP	ParallelHashAntisemijoin	请参见“HashJoin 算法（JH、JHSP、JHFO、JHAP、JHO、JHPO）”一节第 555 页。
JHO	Left Outer HashJoin	请参见“HashJoin 算法（JH、JHSP、JHFO、JHAP、JHO、JHPO）”一节第 555 页。
JHP	ParallelHashJoin	请参见“HashJoin 算法（JH、JHSP、JHFO、JHAP、JHO、JHPO）”一节第 555 页。
JHPO	ParallelLeftOuterHashJoin	请参见“HashJoin 算法（JH、JHSP、JHFO、JHAP、JHO、JHPO）”一节第 555 页。
JHR	RecursiveHashJoin	请参见“RecursiveHashJoin 算法 (JHR)”一节第 556 页。

简要文本计划	详细文本计划	其它信息
JHRO	RecursiveLeftOuterHashJoin	请参见“RecursiveLeftOuterHashJoin 算法 (JHRO)”一节第 556 页。
JM	MergeJoin	请参见“MergeJoin 算法（JM、JMFO、JMO）”一节第 557 页。
JMFO	Full Outer MergeJoin	请参见“MergeJoin 算法（JM、JMFO、JMO）”一节第 557 页。
JMO	Left Outer MergeJoin	请参见“MergeJoin 算法（JM、JMFO、JMO）”一节第 557 页。
JNL	NestedLoopsJoin	请参见“NestedLoopsJoin 算法（JNL、JNLFO、JNLO）”一节第 558 页。
JNLA	NestedLoopsAntisemijoin	请参见“NestedLoopsAntisemijoin 算法 (JNLA)”一节第 558 页。
JNLFO	Full Outer NestedLoopsJoin	请参见“NestedLoopsJoin 算法（JNL、JNLFO、JNLO）”一节第 558 页。
JNLO	Left Outer NestedLoopsJoin	请参见“NestedLoopsJoin 算法（JNL、JNLFO、JNLO）”一节第 558 页。
JNLS	NestedLoopsSemijoin	请参见“NestedLoopsSemijoin 算法 (JNLS)”一节第 558 页。
KEYSET	Keyset	表示由键集驱动的游标。请参见“SQL Anywhere 游标”一节 《SQL Anywhere 服务器 - 编程》。
LOAD	Load	装载操作的根节点。请参见“LOAD TABLE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。
MultiIdx	MultipleIndexScan	请参见“MultipleIndexScan 方法 (MultIdx)”一节第 552 页。
OpenString	OpenString	请参见“OpenString 算法 (OpenString)”一节第 567 页。
	优化时间	优化程序在给定语句的所有枚举过程期间所花费的总时间。
PC	ProcCall	过程调用（表函数）。请参见“ProcCall 算法 (PC)”一节第 567 页。

简要文本计划	详细文本计划	其它信息
PreFilter	PreFilter	请参见“过滤器算法（过滤器、PreFilter）”一节第 566 页。
RL	RowLimit	请参见“RowLimit 算法 (RL)”一节第 568 页。
ROWID	RowIdScan	在图形式计划中，行 ID 扫描的显示为矩形中的表名。请参见“RowIdScan 方法 (ROWID)”一节第 554 页。
ROWS	RowConstructor	请参见“RowConstructor 算法 (ROWS)”一节第 568 页。
RR	RowReplicate	请参见“RowReplicate 算法 (RR)”一节第 563 页。
RT	RecursiveTable	请参见“RecursiveTable 算法 (RT)”一节第 563 页。
RU	RecursiveUnion	请参见“RecursiveUnion 算法 (RU)”一节第 563 页。
SELECT	Select	SELECT 操作的根节点。请参见“SELECT 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。
seq	TableScan、ParallelTableScan	在图形式计划中，表扫描显示为矩形中的表名。请参见“TableScan 方法 (seq)”一节第 552 页和“ParallelTableScan 方法”一节第 553 页。
Sort	Sort	索引或合并排序。请参见“排序算法（排序）”一节第 563 页。
SrtN	SortTopN	请参见“SortTopN 算法 (SrtN)”一节第 564 页。
TermBreak	TermBreak	全文搜索 termbreaker 算法。请参见“变更文本索引”一节第 302 页。
UA	UnionAll	请参见“UnionAll 算法 (UA)”一节第 563 页。
UPDATE	Update	UPDATE 操作的根节点。请参见“UPDATE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。
Window	Window	请参见“窗口算法 (Window)”一节第 568 页。
Work	Work table	表示中间结果的内部节点。

计划中使用的常见统计信息

下面的统计信息是实际的测量值。

统计	解释
Invocations	从子树中请求行的次数。
RowsReturned	为当前节点返回的行数。
RunTime	执行子树所需的时间（包括执行子项所需的时间）。
CacheHits	成功读取高速缓存的次数。
CacheRead	已经在高速缓存中查找的数据库页数。
CacheReadTable	已经从高速缓存中读取表页数。
CacheReadIndLeaf	已经从高速缓存中读取的索引叶页数。
CacheReadIndInt	已经从高速缓存中读取的索引内部节点页数。
DiskRead	已经从磁盘中读取的页数。
DiskReadTable	已经从磁盘中读取的表页数。
DiskReadIndLeaf	已经从磁盘中读取的索引叶页数。
DiskReadIndInt	已经从磁盘中读取的索引内部节点页数。
DiskWrite	已经写入磁盘的页数（工作表页数或修改表页数）。
IndAdd	已经添加到索引中的条目数。
IndLookup	已经在索引中查找的条目数。
FullCompare	已经在索引中的散列值之上执行的比较次数。

计划中的常见估计值

统计信息	解释
EstRowCount	节点在每次被调用时将返回的估计行数。
AvgRowCount	每次调用时返回的平均行数。这不是估计值，而是按照 RowsReturned/ Invocations 进行计算。如果该值与 EstRowCount 之间的差值很大，则说明选择性估计值可能会不正确。

统计信息	解释
EstRunTime	执行所需的估计时间（EstDiskReadTime、EstDiskWriteTime 和 EstCpuTime 的总和）。
AvgRunTime	执行所需的平均时间（测量值）。
EstDiskReads	从磁盘中读取的估计次数。
AvgDiskReads	从磁盘中读取的平均次数（测量值）。
EstDiskWrites	写入磁盘的估计次数。
AvgDiskWrites	写入磁盘的平均次数（测量值）。
EstDiskReadTime	从磁盘中读取行所需的估计时间。
EstDiskWriteTime	将行写入磁盘所需的估计时间。
EstCpuTime	执行所需的估计处理器时间。

计划中与 SELECT、INSERT、UPDATE 和 DELETE 相关的项

项	解释
优化目标	确定优化查询处理的意图：是迅速返回第一行，还是为最大程度地降低返回整个结果集的开销。请参见“optimization_goal 选项 [数据库]”一节 《SQL Anywhere 服务器 - 数据库管理》。
优化负载	确定优化查询处理的目标是针对更新和读取混合进行的负载还是针对主要基于读取的负载。请参见“optimization_workload 选项 [数据库]”一节 《SQL Anywhere 服务器 - 数据库管理》。
ANSI 更新约束	控制允许的更新范围（选项包括 Off、Cursors 和 Strict）。请参见“ansi_update_constraints 选项 [兼容性]”一节 《SQL Anywhere 服务器 - 数据库管理》
优化级别	保留。
选择列表	查询所选择的表达式的列表。

项	解释
实例化视图	优化程序所考虑的实例化视图列表。列表中的每个条目均为以下格式的元组：`view-name [ view-matching-outcome ] [ table-list ]`，其中 *view-matching-outcome* 显示实例化视图的用法；如果值为 COSTED，则说明视图已在枚举时使用。*table-list* 是可能已被此视图替代的查询表的列表。 *view-matching-outcome* 的值包括： ● 基本表不匹配 ● 权限不匹配 ● 谓语不匹配 ● 选择列表不匹配 ● 已用开销 ● 失效不匹配 ● 快照失效不匹配 ● 无法由优化程序使用 ● 无法由优化程序在内部使用 ● 无法建立定义 ● 无法访问 ● 已禁用 ● 选项不匹配 ● 已达到视图匹配阈值 ● 使用的视图 有关阻止优化程序使用实例化视图的限制和条件的详细信息，请参见“使用实例化视图提高性能”一节第 536 页和“实例化视图的限制”一节第 51 页。

计划中与锁相关的项

项	解释
被锁定的表	所有被锁定的表及其关联的隔离级别的列表。

计划中与扫描相关的项

项	解释
表名	表的实际名称。
相关名	表的别名。
估计行数	表中的估计行数。
估计页数	表中的估计页数。
估计行大小	表的估计行大小。

项	解释
页位置图	在使用页位置图来读取多页时为 [是]。

计划中与索引扫描相关的项

项	解释
选择性	匹配域范围的估计行数。
索引名	索引的名称。
键类型	可以是 PRIMARY KEY（主键）、FOREIGN KEY（外键）、CONSTRAINT（唯一约束）或 UNIQUE（唯一索引）之一。如果索引是非唯一的辅助索引，则不会显示键类型。
深度	索引的高度。请参见“表大小和页面大小”一节第 595 页。
估计叶页数	估计的叶页数。
顺序转换	用于指示索引聚簇方式的每个物理索引的统计信息。
随机转换	用于指示索引聚簇方式的每个物理索引的统计信息。
键值	索引中独特条目的数量。
基数	索引的基数（如果它不同于估计行数）。这仅适用于 SQL Anywhere 数据库版本 6.0.0 及更早版本。
方向	FORWARD（向前）或BACKWARD（向后）。
域范围	域范围显示为列表 (col_name=value) 或 col_name IN [low, high]。
主键表	外键索引扫描的主键表名称。
主键表预计行数	外键索引扫描的主键表中的行数。
主键列	外键索引扫描的主键列名称。

计划中与连接、过滤器和预过滤器相关的项

项	解释
谓语	在此节点中求出的搜索条件以及选择性估计值和测量值。请参见“在图形式计划中查看选择性”一节第 576 页

计划中与散列过滤相关的项

项	解释
生成值	输入中的不重复值的估计数。
探测值	查看谓语时输入中的不重复值的估计数。
位	选择生成散列映射的位数。
页	存储散列映射所需的页数。

计划中与联合相关的项

项	解释
联合列表	UNION 语句中涉及的列。

计划中与 GROUP BY 相关的项

项	解释
集合	所有集合函数。
GROUP BY 列表	GROUP BY 子句中的所有列。

计划中与 DISTINCT 相关的项

项	解释
DISTINCT 列表	DISTINCT 子句中的所有列。

计划中与 IN 列表相关的项

项	解释
IN 列表	指定集合中的所有表达式。

项	解释
表达式 SQL	要与该列表进行比较的表达式。

计划中与 SORT 相关的项

项	解释
Order-by	列出要作为排序依据的所有表达式。

计划中与行限制相关的项

项	解释
行限制计数	按照 FIRST 或 TOP n 的指定所返回的最大行数。

提高查询性能

每个表或条目的存储分配都对查询的效率有着较大的影响。以下所述的每一点都会影响查询执行的速度，所以它们特别重要。

对插入的行的磁盘分配

下面一节将说明数据库中的行是如何存储在磁盘上的。

SQL Anywhere 会在可能的情况下连续存储行

每一个小于数据库文件页面大小的新行都始终存储在一个页上。如果现有页没有足够的空间可用于新行，SQL Anywhere 就会将该行写入新页。例如，如果新行需要 600 字节的空间，但一个被部分填充的页上只有 500 字节的可用空间，SQL Anywhere 就会将该行放在一个新页上。

为了使表页在磁盘上更为连续，SQL Anywhere 将以块的形式（每块包含八页）来分配表页。例如，当它需要分配某个页时，它将先分配八页，将要分配的页插入块中，然后用后面的七页将块填满。此外，它会使用可用页位图来查找 dbspace 中的连续页块，并通过读取 64 KB 的组和使用位图查找相关页来执行顺序扫描。这样，顺序扫描的效率就会得到提高。

SQL Anywhere 可以按任意顺序存储行

SQL Anywhere 查找页上的空间并将行按接收顺序插入其中。它会将每一行分配给某一页，但它在表上选择的位置可能与行的插入顺序不对应。例如，为了连续存储一个很长的行，数据库服务器可能需要开始一个新页。如果下一行较短，它可能适合放在前一页上的空位置。

所有表的行都未进行排序。如果行的接收或处理顺序非常重要，请在 SELECT 语句中使用 ORDER BY 子句对结果进行排序。依赖于表中行顺序的应用程序可能会不发出警告就失败。

如果您经常需要让表的行具有特定的顺序，则应在查询的 ORDER BY 子句中所指定的那些列上创建索引。

没有为 NULL 列预留空间

缺省情况下，每当 SQL Anywhere 插入一行时，它所预留的存储空间仅为存储该行在创建时所含的值所必需的空间。它不会为存储 NULL 值或容纳大小可能会增加的字段（例如文本字符串）而预留空间。

可以通过在创建表时使用 PCTFREE 选项来强制 SQL Anywhere 预留空间。有关详细信息，请参见“CREATE TABLE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

行标识符在插入后不可变

在页上为行分配主位置之后，行就不会再从该页上移开。如果更新操作更改了该行中的某些值，导致行不能被装入为其指派的页，则行会拆分开来，额外信息将被插入到另一页。

这一特点需要特别注意，因为在插入行时，SQL Anywhere 不允许额外的空间。例如，假设您在表中插入大量的空行，然后使用 UPDATE 语句一次一列地填入值。结果将是几乎单个行中的每个值都存储在一个单独页上。为检索一行中的所有值，数据库服务器可能需要读取若干个磁盘页。这一简要操作将变得极为缓慢而且这种缓慢毫无必要。

您应该考虑在插入时用数据填充新行。插入之后，它们将具有足够的空间来保存您需要保存的数据。

数据库文件从不会缩小

当从数据库插入和删除行时，SQL Anywhere 会自动重新使用这些行占用的空间。因此，SQL Anywhere 可以将行插入先前被其它行占用的空间。

SQL Anywhere 会记录每个页上的空白空间大小。当您要求它插入新行时，它首先会搜索有关现有页空间的记录。如果它发现现有的页上有足够的空间，就会将新行放在该页上，并在必要时重组该页的内容。否则，它会开始一个新页。

如果您在一段时间内删除了几行，并且所插入的新行都比较大，以至于无法利用空白空间，则数据库中的信息就会变得非常分散。您可以重装该表，或使用 REORGANIZE TABLE 语句整理该表的碎片。

有关详细信息，请参见“REORGANIZE TABLE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

表大小和页面大小

您为数据库选择的页面大小可影响数据库的性能。通常，如果页面大小较小，可能有益于那些从任意位置检索较少行的操作。相比之下，较大的页往往有益于那些按顺序执行表扫描的查询 — 尤其是当行在页上的存储顺序与通过索引对行进行检索的顺序相同时。在这种情况下，如果将一页读入内存以获取一行的值，则可能会产生副作用：会将下几行的内容装入内存。通常，磁盘的物理设计会使磁盘检索较少的大块比检索较多的小块更有效率。

对于每个表，SQL Anywhere 将创建反映各表页在整个 dbspace 文件中位置的位图。数据库服务器使用位图一次性读取表页的大块 (64 KB)，而不是单个页面。此功能（又称作**组读取**）降低了对磁盘进行的 I/O 操作总数，同时提高了性能。用户不能控制数据库服务器有关创建或使用位图的标准。

如果选择较大的页面大小（例如 8 KB），则最好增加高速缓存的大小，因为同样大小的高速缓存所能容纳的大页较少。例如，1 MB 的内存可以保存 512 个大小均为 2 KB 的页，但只能保存 128 个大小均为 8 KB 的页。应根据数据库以及应用程序执行的查询的性质，确定页面大小与高速缓存大小的适当页比率。您可以用各种高速缓存大小进行性能测试。如果高速缓存不能保存足够的页，当数据库服务器开始与磁盘交换常用页时，性能就会受到影响。当在 Windows Mobile 设备上使用 SQL Anywhere 时，这一点很重要，因为较大的页面大小可能有更多的内部碎片。

SQL Anywhere 会尽可能地将页填满。只有当新对象太大而无法装入现有页上的空白空间时，空白空间才会累积。因此，调整页面大小可能不会对数据库的总体大小造成很大的影响。

页面大小也会影响索引。每一次索引查寻都需要对每个索引级别执行一次页读取并对表页执行一次页读取，而一个查询就可能需要数千次索引查寻。页面大小可能会大大地影响条目数，条目数又将影响表所需的索引深度。索引条目数较大，通常意味着需要的索引级别较少，这将大大地提高搜索性能。对于所拥有的表中含有大量行的大型数据库而言，使用大小为 8 KB 的页面可保证实现最佳性能。强烈建议您在选择页面大小时对性能（以及其它行为方面）进行测试。然后，选择可提供满意结果的最小的页面大小。如果在同一服务器上启动多个数据库，选择正确且合理的页面大小就很重要。

另请参见

- “使用适当的页大小”一节第 218 页
- “初始化实用程序 (dbinit)”一节 《SQL Anywhere 服务器 - 数据库管理》
- “CREATE DATABASE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》

索引

索引可以大大地提高索引列上的搜索性能。但是，索引会占用数据库中的空间并降低插入、更新和删除操作的速度。本节将帮助您确定应该何时创建索引，以及如何利用索引实现最佳性能。

在许多情况下，创建索引会提高数据库的性能。索引根据部分或所有列中的值对表的行进行排序。索引使 SQL Anywhere 可以快速地查找行。它可以通过限制所访问的数据库页数来支持较大的并发性。索引还使 SQL Anywhere 能够方便地对表中的行强制实施唯一性约束。

创建索引时，列会按您指定的顺序出现在索引中。索引定义中不允许重复引用列名称。

[索引顾问] 是一种工具，用于协助您为数据库选择一组适当的索引。请参见“索引顾问”一节第 167 页。

使用逻辑索引共享索引

SQL Anywhere 使用物理索引和逻辑索引。物理索引是存储在磁盘上的实际索引结构。逻辑索引是对物理索引的引用。在创建主键、辅助键、外键或唯一约束时，数据库服务器通过为约束创建逻辑索引来确保参照完整性。然后，数据库服务器开始查找是否已存在满足约束的物理索引。如果已存在满足约束的物理索引，则数据库服务器会将逻辑索引指向该物理索引。如果不存在这样的物理索引，则数据库服务器会创建新的物理索引，然后再将逻辑索引指向它。

为使物理索引满足逻辑索引的要求，实际列、列顺序以及各列中数据的排序（升序、降序）必须相同。

有关数据库中所有逻辑索引和物理索引的信息分别记录在 ISYSIDX 和 ISYSPHYSIDX 系统表中。当创建逻辑索引时，在 ISYSIDX 系统表中会生成一个条目以保存索引定义。对用于满足逻辑索引的物理索引的引用记录在 ISYSIDX.phys_id 列中。物理索引在 ISYSPHYSIDX 系统表中定义。

有关 ISYSIDX 和 ISYSPHYSIDX 系统表的详细信息，请参见其对应视图，“SYSIDX 系统视图”一节 《SQL Anywhere 服务器 - SQL 参考》和“SYSPHYSIDX 系统视图”一节 《SQL Anywhere 服务器 - SQL 参考》。

如果使用逻辑索引，则数据库服务器不需要创建和维护重复的物理索引，因为可以有多个逻辑索引指向一个物理索引。

删除逻辑索引时，其定义也会从 ISYSIDX 系统表中删除。如果它是引用特定物理索引的唯一逻辑索引，则还会删除该物理索引以及 ISYSPHYSIDX 系统表中的对应条目。

在创建索引之前，应始终慎重考虑是否需要一个索引。请参见“何时创建索引”一节第 598 页。

不会为远程表创建物理索引。对于临时表会创建物理索引，不过不记录到 ISYSPHYSIDX 中，且在使用之后即被丢弃。此外，临时表的物理索引是不共享的。

确定哪些逻辑索引共享一个物理索引

删除索引时，删除的是一个逻辑索引；但不会始终删除它所引用的物理索引。如果有其它逻辑索引引用同一物理索引，则不会删除该物理索引。了解这一点非常重要，特别是在希望通过删除索引释放磁盘空间时，或意在删除索引后以物理方式重新创建索引时。

要确定一个表的索引是否与任何其它索引共享一个物理索引，请在 Sybase Central 中选择该表，然后单击 [**索引**] 选项卡。注意该索引的 Phys.ID 值是否与列表中其它索引的 Phys.ID 值相同。如果 Phys.ID 值匹配，则说明这些索引共享同一物理索引。如果希望重新创建物理索引，可使用 ALTER INDEX ...REBUILD 语句。也可以删除所有索引，然后重新进行创建。

确定要共享哪些表中的物理索引

可随时通过执行类似下面的查询来获得要共享其物理索引的所有表的列表：

```
SELECT tab.table_name, idx.table_id, phys.phys_index_id, COUNT(*)
 FROM SYSIDX idx JOIN SYSTAB tab ON (idx.table_id = tab.table_id)
 JOIN SYSPHYSIDX phys ON ( idx.phys_index_id  = phys.phys_index_id
   AND idx.table_id = phys.table_id )
 GROUP BY tab.table_name, idx.table_id, phys.phys_index_id
   HAVING COUNT(*) > 1
ORDER BY tab.table_name;
```

以下是查询的示例结果集：

table_name	table_id	phys_index_id	COUNT(*)
ISYSCHECK	57	0	2
ISYSCOLSTAT	50	0	2
ISYSFKEY	6	0	2
ISYSSOURCE	58	0	2
MAINLIST	94	0	3
MAINLIST	94	1	2

每个表的行数表示该表共享的物理索引数。在此示例中，除虚构表 MAINLIST 有两个共享物理索引外，其它所有表都只有一个共享物理索引。phys_index_id 值标识要共享的物理索引，COUNT 列中的值表明共享该物理索引的逻辑索引的数量。

还可以使用 Sybase Central 查看给定表中的哪些索引共享一个物理索引。为此，请在左侧窗格中选择表，并在右侧窗格中单击 [**索引**] 选项卡，然后查找与 Phys.ID 列的值相同的多个行。Phys.ID 值相同的索引共享同一物理索引。

另请参见

- “重建索引”一节第 70 页
- “ALTER INDEX 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “SYSIDX 系统视图”一节 《SQL Anywhere 服务器 - SQL 参考》

何时创建索引

没有任何简要的公式可以确定是否应创建索引。您必须考虑权衡索引检索的优点与该索引的维护开销。以下因素可帮助确定是否应创建索引：

- **键和唯一列** SQL Anywhere 自动在主键、外键和唯一列上创建索引。您不应该在这些列创建附加的索引。组合键属于例外，它们有时可以通过附加的索引来得到改进。

  有关详细信息，请参见“复合索引”一节第 599 页。

- **搜索频率** 如果频繁地搜索某一特定列，则可以通过在该列上创建索引来改善性能。在很少进行搜索的列上创建索引可能没有意义。
- **表大小** 在包含许多行的相对较大的表上创建索引比在相对较小的表上创建索引更有好处。例如，对于只有 20 行的表，由于按顺序进行扫描所需的时间并不比进行索引查寻所需的时间长，因此在该表上创建索引没有多少好处。
- **更新次数** 每次在表中插入或删除行以及每次更新索引列时，都会更新索引。列上的索引会降低插入、更新和删除操作的速度。频繁更新的数据库的索引数应少于只读数据库的索引数。
- **空间考虑** 索引会占用数据库中的空间。如果数据库大小是您关注的主要问题，则应尽量少创建索引。
- **数据分布** 如果索引查寻返回的值太多，则索引查寻的开销会高于顺序扫描的开销。当 SQL Anywhere 发现这一情况时，它就不会使用索引。例如，SQL Anywhere 不会在只有两个值的列（如 SQL Anywhere 示例数据库中的 Employees.Sex）上使用索引。因此，不应在只有几个不同值的列上创建索引。

[索引顾问] 是一种工具，用于协助您为数据库选择一组适当的索引。请参见“索引顾问”一节第 167 页。

临时表

您可以在本地临时表和全局临时表上创建索引。如果临时表会很大，并且会按排序顺序或连接对其进行多次访问，则需要在临时表上创建索引。否则，查询的任何性能改善都可能会被创建和删除索引的开销所抵消。

有关详细信息，请参见“使用索引”一节第 66 页。

提高索引性能

如果索引没有按预期的方式执行，则可以考虑采取以下措施：

- 重组复合索引。
- 增加页面大小。

这些措施旨在提高索引选择性和索引条目数，下面将对此进行介绍。

索引选择性

索引选择性指的是索引无需读取额外数据即可定位所需索引条目的能力。

如果选择性较低，则必须从索引所引用的表页中检索额外信息。这种检索称为**完全比较**，它们会对索引性能产生负面影响。

FullCompare 属性函数会一直跟踪已发生的完全比较的次数。您也可以使用 Sybase Central 性能监控器或 Windows 性能监控器来监控这一统计信息。

注意
Windows Mobile 上可能未提供 Windows 性能监控器。

此外，完全比较的次数会在含统计信息的图形式计划中提供。有关详细信息，请参见“计划中使用的常见统计信息”一节第 588 页。

有关 FullCompare 函数的详细信息，请参见“数据库属性”一节 《SQL Anywhere 服务器 - 数据库管理》。

索引结构和索引条目数

索引是以多个级别组织的，就像树一样。索引的第一页被称作根页，它在下一级分叉为一个或多页，而这些页又再次分叉，直至达到索引的最低级别。位于最低级别的索引页被称作叶页。若要定位特定行，具有 *n* 级的索引需要对索引页读取 *n* 次并对包含实际行的数据页读取一次。通常，由于频繁使用的索引页往往被存储在高速缓存中，所以需要从磁盘中读取的次数会少于 *n* 次。

索引条目数是一个页面上存储的索引条目的数量。与条目数较小的索引的级别数相比，条目数较大的索引的级别数可能会更少。因此，如果索引条目数较大，通常意味着索引性能更佳。为数据库选择正确的页面大小可提高索引条目数。请参见“表大小和页面大小”一节第 595 页。

您可以使用 sa_index_levels 系统过程来查看索引中的级别数。请参见“sa_index_levels 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》。

复合索引

索引可以包含一个、两个或多个列。两个或更多个列上的索引被称作**复合索引**。例如，以下语句会创建一个具有两列的复合索引：

```
CREATE INDEX name
ON Employees ( Surname, GivenName );
```

如果第一列不能单独提供较高的选择性，复合索引将会非常有用。例如，如果许多雇员都同姓，则 Surname 和 GivenName 上的复合索引非常有用。因为每个雇员都有唯一的 ID，所以 EmployeeID 和 Surname 上的复合索引可能没有用处，因此列 Surname 不会提供任何附加选择性。

利用索引中的附加列，您可以缩小搜索的范围，但使用一个具有两列的索引不同于使用两个单独的索引。复合索引的结构与电话簿类似，它首先按姓对雇员进行排序，然后按名对所有同姓的雇员进行排序。如果您知道姓，电话簿将非常有用，如果您知道名和姓，电话簿则更为有用，但如果您只知道名而不知道姓，电话簿将没有用处。

列顺序

在创建复合索引时，应仔细考虑列的顺序。对索引中的所有列执行搜索或仅对前几列执行搜索时，复合索引非常有用；仅对后面的任意列执行搜索时，复合索引并没有用处。

如果您要仅对一个列执行多次搜索，则该列应该是复合索引中的第一个列。如果您要对一个两列索引中的两个列都执行单独搜索，则应创建另外一个仅包含第二列的索引。

例如，假定您在两个列上创建一个复合索引。其中一列包含雇员的名，另一列包含雇员的姓。您可以创建一个先包含名后包含姓的索引。或者，您也可以创建一个先包含姓后包含名的索引。虽然这两个索引都以两个列组织信息，但它们具有不同的功能。

```
CREATE INDEX IX_GivenName_Surname
   ON Employees ( GivenName, Surname );
CREATE INDEX IX_Surname_GivenName
   ON Employees ( Surname, GivenName );
```

假定您需要搜索指定的名字 John。唯一有用的索引是在索引的第一列包含指定名字的索引。由于名为 John 的雇员会出现在索引中的任意位置，因此先按姓再按名组织的索引没有用处。

如果您更可能仅按名或仅按姓查找雇员，则应考虑创建这两个索引。

或者，您也可以创建两个索引，每个索引仅包含一个列。但是请记住，SQL Anywhere 在处理单个查询时只使用一个索引来访问任何一个表。即使您知道名和姓，SQL Anywhere 也可能需要读取额外的行，以查找包含正确姓的行。

当使用 CREATE INDEX 命令创建索引时（如上例所示），列会按命令中所示的顺序显示。

复合索引和 ORDER BY

缺省情况下，索引的列按升序排列，但您可以选择通过在 CREATE INDEX 语句中指定 DESC 来将这些列按降序排列。

只要 ORDER BY 子句仅包含索引中的列，SQL Anywhere 就可以选择使用索引来优化 ORDER BY 查询。此外，索引列的排序方式必须与 ORDER BY 子句完全相同或完全相反。对于单列索引，这种排序方式始终会使查询可以得到优化，但复合索引则需要稍微多考虑一些问题。下表显示了一个两列索引的可能性。

索引列	可优化的 ORDER BY 查询	不可优化的 ORDER BY 查询
ASC、ASC	ASC、ASC 或 DESC、DESC	ASC、DESC 或 DESC、ASC
ASC、DESC	ASC、DESC 或 DESC、ASC	ASC、ASC 或 DESC、DESC
DESC、ASC	DESC、ASC 或 ASC、DESC	ASC、ASC 或 DESC、DESC
DESC、DESC	DESC、DESC 或 ASC、ASC	ASC、DESC 或 DESC、ASC

含有两个以上的列的索引遵循上述一般规则。例如，假定您有以下索引：

```
CREATE INDEX idx_example
ON table1 ( col1 ASC, col2 DESC, col3 ASC );
```

在这种情况下，以下查询可以得到优化：

```
SELECT col1, col2, col3 FROM table1
ORDER BY col1 ASC, col2 DESC, col3 ASC;

SELECT col1, col2, col3 FROM example
ORDER BY col1 DESC, col2 ASC, col3 DESC;
```


索引不用于优化在 ORDER BY 子句中具有 ASC 和 DESC 的任何其它模式的查询。例如，以下语句不会被优化：

```
SELECT col1, col2, col3 FROM table1
ORDER BY col1 ASC, col2 ASC, col3 ASC;
```

索引的其它用途

SQL Anywhere 使用索引来实现其它性能优点。通过索引，SQL Anywhere 可以强制列唯一性，减少必须锁定的行数和页数，以及更好地估计谓语的选择性。

- **强制列唯一性** 如果没有索引，每当插入值时，SQL Anywhere 都需要扫描整个表以确保该值唯一。因此，SQL Anywhere 会自动在每个具有唯一性约束的列上构建索引。
- **减少锁** 索引可以减少在插入、更新和删除过程中必须锁定的行数和页数。这种减少是索引在表上强制排序所导致的。

  有关索引和锁定的详细信息，请参见“锁定的工作方式”一节第 122 页。
- **估计选择性** 由于索引是经过排序的，优化程序可以通过扫描索引的上层级别来估算满足给定查询的值所占的百分比。此操作称为部分索引扫描。

B 链接索引

B 链接索引是 B- 和 B+- 树索引的变体，在这类树索引中，每个索引页、非叶和叶均包含其右侧兄弟节点的页数（或指向其右侧兄弟节点的链接）。此外，索引页无需立即在父页中出现。B 链接索引的主要优点是提高了并发性。

索引可以声明为聚簇索引，也可以声明为非聚簇索引。一个表上只有一个索引可以是聚簇索引。如果确定一个索引应为聚簇索引，则不需要删除或重新创建该索引；通过发出 ALTER INDEX 语句，可以删除或添加索引的聚簇特性。聚簇索引有助于提高性能，这是因为查询优化程序可以利用聚簇更准确地确定索引扫描的开销。

为提高条目数，SQL Anywhere 以压缩形式存储每个索引值，其中的前缀（与前面紧邻的值共享）不加以存储。为减少在页面内搜索时的 CPU 时间，还会存储完整索引键的小型后备映射（受数据长度的限制）。特别是，SQL Anywhere 索引会有效地处理相等（或大致相等）的索引值，因此索引值中的通用前缀对存储要求和性能的影响可以忽略不计。

另请参见

- “使用聚簇索引”一节第 67 页
- “ALTER INDEX 语句”一节 《SQL Anywhere 服务器 - SQL 参考》

SQL 方言和兼容性

本节介绍 Transact-SQL 兼容性以及其它 SQL 实现通常所不具有的那些 SQL Anywhere 特性。

SQL 方言

目录

SQL Anywhere 遵从性简介

SQL Anywhere 完全符合基于 SQL-92 的第 127 号美国联邦信息处理标准发布 (FIPS PUB)。除一些微小差异外，SQL Anywhere 也符合 ISO/ANSI SQL-2003 核心规范。SQL Anywhere 各项功能的参考文档中提供了有关遵从性的信息。

使用 SQL Flagger 测试 SQL 遵从性

在 SQL Anywhere 中，数据库服务器和 SQL 预处理器 (sqlpp) 可标识作为服务商扩充、不符合特定 ISO/ANSI SQL 标准、或者 UltraLite 不支持的 SQL 语句。此功能称为 SQL Flagger，且它是 SQL/1999 和 SQL/2003 ISO/ANSI SQL 标准的一部分。SQL Flagger 帮助应用程序开发人员标识违反 SQL 语言指定子集的 SQL 语言构造。SQL Flagger 还可用于确保符合 SQL 标准的核心特性，或符合核心特性和可选特性的组合。也可以在原型化使用 SQL Anywhere 的 UltraLite 应用程序时使用 SQL Flagger，以确保所使用的 SQL 受 UltraLite 支持。

尽管 SQL 语句的语法和语义元素均需要 SQL Flagger 进行分析，但 SQL Flagger 旨在提供对遵从性的静态、编译时检查。语法遵从性的一个示例测试是在 INSERT 语句中缺少可选 INTO 关键字（例如，`INSERT Products VALUES( ... )`），它是 SQL 语言的 SQL Anywhere 语法扩充。使用不带 INTO 关键字的 INSERT 语句会被标记为服务商扩充，因为 ANSI SQL/2003 标准要求使用 INTO 关键字。但是，请注意，INTO 关键字对于 UltraLite 应用程序是可选的。

键连接也会被标记为服务商扩充。如使用 JOIN 关键字并且没有 ON 子句，则缺省情况下使用键连接。键连接使用现有外键关系连接表。UltraLite 不支持键连接。例如，以下查询指定了 Products 和 SalesOrderItems 表之间的隐式连接条件。此查询被 SQL Flagger 标记为服务商扩充。

```
SELECT * FROM Products JOIN SalesOrderItems;
```

SQL Flagger 功能不依赖于 SQL 语句的执行；所有标记逻辑仅作为静态、编译时过程执行。

另请参见

- “SQLFLAGGER 函数 [Miscellaneous]”一节 《SQL Anywhere 服务器 - SQL 参考》
- “运行 SQL 预处理器”一节 《SQL Anywhere 服务器 - 编程》
- “INSERT 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “键连接”一节第 391 页

调用 SQL Flagger

SQL Anywhere 提供了多种调用 SQL Flagger 的方法，以检查一个 SQL 语句或一批 SQL 语句：

- **SQLFLAGGER 函数** SQLFLAGGER 函数分析作为字符串参数传递的单个 SQL 语句或批处理是否符合给定的 SQL 标准。对语句或批处理进行分析，但不会执行。请参见“SQLFLAGGER 函数 [Miscellaneous]”一节 《SQL Anywhere 服务器 - SQL 参考》。
- **sa_ansi_standard_packages 系统过程** sa_ansi_standard_packages 系统过程分析单个语句或者批处理，以确定是否使用了 ANSI SQL/2003 或 SQL/1999 国际标准中的可选程序包。对语句或批处理进行分析，但不会执行。请参见“sa_ansi_standard_packages 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》。
- **sql_flagger_error_level 和 sql_flagger_warning_level 选项** sql_flagger_error_level 和 sql_flagger_warning_level 选项调用 SQL Flagger，用于为连接做准备或执行的任何语句。如果语句不符合作为特定 ANSI 标准或 UltraLite 的选项设置，则根据选项设置，语句会以错误 (SQLSTATE 0AW03) 终止，或返回一个警告 (SQLSTATE 01W07)。如果语句符合选项设置，则它会正常执行。请参见“sql_flagger_error_level 选项 [兼容性]”一节 《SQL Anywhere 服务器 -

数据库管理》和“sql_flagger_warning_level 选项 [兼容性]”一节 《SQL Anywhere 服务器 - 数据库管理》。

- **SQL 预处理器 (sqlpp)** SQL 预处理器 (sqlpp) 可在编译时在嵌入式 SQL 应用程序中标记静态 SQL 语句。此特性在开发 UltraLite 应用程序时对于验证 SQL 语句的 UltraLite 兼容性特别有用。请参见“SQL 预处理器”一节 《SQL Anywhere 服务器 - 编程》和“运行 SQL 预处理器”一节 《SQL Anywhere 服务器 - 编程》。

另请参见

- “批处理简介”一节第 797 页

标准和兼容性

数据库服务器和 SQL 预处理器中使用的标记功能遵循 ANSI/ISO SQL/2003 国际标准第 1 部分（框架）中定义的 SQL Flagger 功能。SQL Flagger 使用以下 ANSI SQL 标准确定 SQL 构造的遵从性：

- SQL/1992 入门级、中间级和完整级
- SQL/1999 核心和 SQL/1999 可选程序包
- SQL/2003 核心和 SQL/2003 可选程序包

注意
不建议使用 SQL/1992（所有级别）的 SQL Flagger 支持。

此外，SQL Flagger 可标识不符合 UltraLite SQL 的语句。例如，UltraLite 对 CREATE 和 ALTER 模式对象只有有限的能力。

SQL Flagger 可分析所有的 SQL 语句。但是，大部分创建或修改模式对象的语句，包括创建表、索引、实例化视图、发布、预订和代理表的语句，均为 ANSI SQL 标准的服务商扩充，因而被标记为不符合要求。

永远不会因为不符合任何 SQL 标准或因为与 UltraLite 兼容，而标记 SET OPTION 语句，包括其可选组件。

另请参见

- “UltraLite SQL 元素” 《UltraLite - 数据库管理和参考》
- “SET OPTION 语句”一节 《SQL Anywhere 服务器 - SQL 参考》

其它 SQL 实现中没有的功能

SQL Anywhere 所支持的以下 SQL 功能在其它许多 SQL 实现中都没有。

日期

SQL Anywhere 具有日期、时间和时间戳类型，包括年月日、小时、分钟、秒和秒的小数值。插入或更新日期字段时，或与日期字段进行比较时，支持使用自由格式日期。

此外，还可以对日期进行以下运算：

- **日期 + 整数** 在日期中加上指定的天数。
- **日期 - 整数** 从日期中减去指定的天数。
- **日期 - 日期** 计算两个日期之间的天数。
- **日期 + 时间** 用日期和时间创建一个时间戳。

另外，还提供了许多用于操作日期和时间的函数。有关这些函数的说明，请参见“SQL 函数”《SQL Anywhere 服务器 - SQL 参考》。

完整性

SQL Anywhere 支持实体完整性和参照完整性。这是通过对 CREATE TABLE 和 ALTER TABLE 命令进行以下两个扩展实现的。

```
PRIMARY KEY ( column-name, ... )
[NOT NULL] FOREIGN KEY [role-name]
             [(column-name, ...)]
          REFERENCES table-name [(column-name, ...)]
             [ CHECK ON COMMIT ]
```

PRIMARY KEY 子句声明关系的主键。之后，SQL Anywhere 会强制实现主键的唯一性，并确保主键中的任何一列都不包含 NULL 值。

FOREIGN KEY 子句定义此表与另一个表之间的关系。这个关系由此表中的一列（或多列）表示，该列必须包含另一个表的主键中的值。之后，系统会确保这些列的参照完整性 - 只要修改了这些列，或在此表中插入了一行，都会对这些列进行检查，以确保有一列或多列为 NULL，或对于其它表的某行这些值与主键中的对应列相匹配。有关详细信息，请参见“CREATE TABLE 语句”一节《SQL Anywhere 服务器 - SQL 参考》。

连接

SQL Anywhere 允许表之间的自动连接。除其它实现中支持的 NATURAL 和 OUTER 连接操作符之外，SQL Anywhere 还允许在表之间基于外键关系建立 KEY 连接。这降低了执行连接时 WHERE 子句的复杂程度。

更新

SQL Anywhere 允许 UPDATE 命令引用多个表。在多个表上定义的视图也可以更新。许多 SQL 实现不允许更新连接的表。

变更表

对 ALTER TABLE 命令进行了扩展。除了用于实体完整性和参照完整性的更改外，还支持以下变更：

```
ADD column data-type
ALTER column data-type
DELETE column
RENAME new-table-name
RENAME old-column TO new-column
```

使用 ALTER 子句可以更改字符列的最大长度，也可以从一种数据类型转换为另一种数据类型。请参见“ALTER TABLE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

支持子查询 WHERE 表达式

SQL Anywhere 允许在任何可以放置表达式的位置放置子查询。许多 SQL 实现只允许将子查询放置在比较运算符的右边。例如，以下命令在 SQL Anywhere 中有效，但在大多数其它 SQL 实现中无效。

```
SELECT Surname,
  BirthDate,
  (  SELECT DepartmentName
     FROM Departments
     WHERE EmployeeID = Employees.EmployeeID
     AND DepartmentID = 200 )
FROM Employees;
```

附加函数

SQL Anywhere 支持若干 ANSI SQL 定义中没有的函数。有关可用函数的完整列表，请参见“SQL 函数” 《SQL Anywhere 服务器 - SQL 参考》。

游标

使用嵌入式 SQL 时，可以使用 FETCH 语句任意移动游标位置。可以相对于当前游标位置或相对于游标最前面或最后面给定数量的记录，前移或后移游标。

别名参照

SQL Anywhere 允许在查询的其它部分引用该查询的选择列表中带别名的表达式。大多数其它 SQL 实现不允许这么做。

Watcom-SQL

SQL Anywhere 所支持的 SQL 方言称为 Watcom-SQL。在 1992 年推出 SQL Anywhere 时，其原始版本称为 Watcom SQL。我们仍然使用术语 Watcom-SQL 来标识 SQL Anywhere 所支持的 SQL 方言。

SQL Anywhere 还支持 Transact-SQL 中相当大的一部分，Transact-SQL 是 Sybase Adaptive Server Enterprise 所支持的 SQL 方言。请参见“Transact-SQL 兼容性”一节第 612 页。

Transact-SQL 兼容性

SQL Anywhere 支持 Transact-SQL 中相当大的一部分，Transact-SQL 是 Sybase Adaptive Server Enterprise 所支持的 SQL 方言。本节介绍 SQL Anywhere 和 Adaptive Server Enterprise 之间的 SQL 兼容性。

目标

SQL Anywhere 对 Transact-SQL 的支持可以达到以下目的：

- **应用程序可移植性** 所编写的许多应用程序、存储过程和批处理文件都可以与 Adaptive Server Enterprise 数据库和 SQL Anywhere 数据库一起使用。
- **数据可移植性** SQL Anywhere 和 Adaptive Server Enterprise 数据库之间可以轻松地相互进行数据交换和复制。

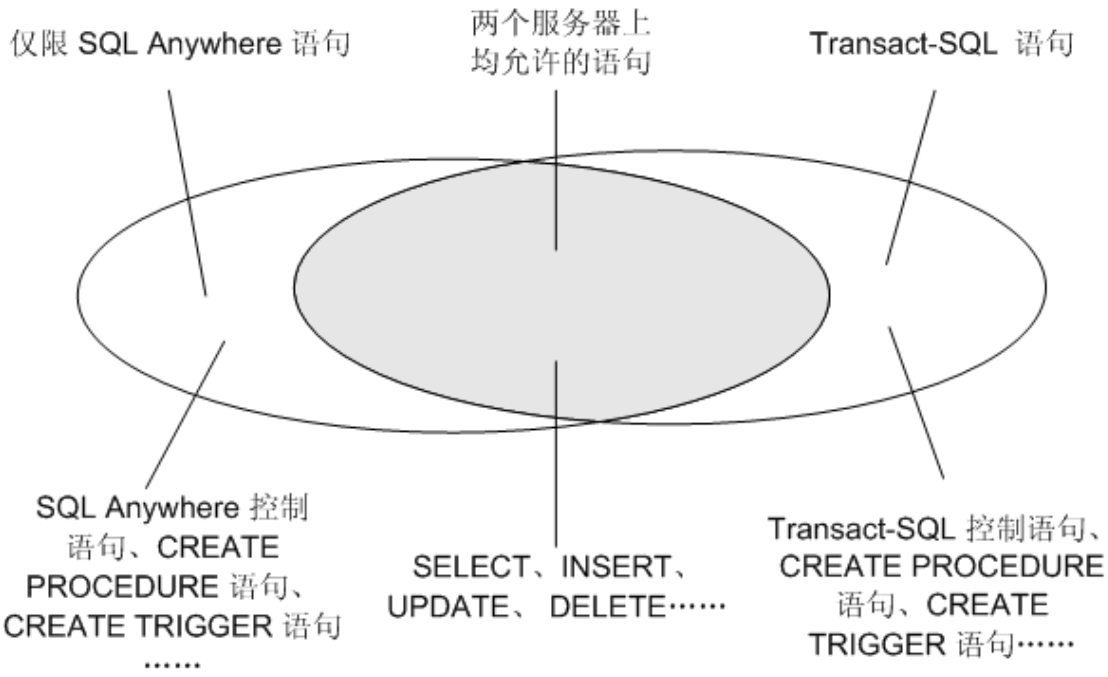
主要的目标是编写既可用于 Adaptive Server Enterprise 又可用于 SQL Anywhere 的应用程序。现有的 Adaptive Server Enterprise 应用程序通常需要做出一些更改才能在 SQL Anywhere 数据库上运行。

怎样才能支持 Transact-SQL

SQL Anywhere 通过以下形式支持 Transact-SQL：

- 许多 SQL 语句在 SQL Anywhere 和 Adaptive Server Enterprise 之间兼容。
- 对于某些语句（尤其是在过程、触发器和批处理语句中使用的过程语言），将支持单独的 Transact-SQL 语句以及 SQL Anywhere 的先前版本中所支持的语法。对于这些语句，SQL Anywhere 支持 SQL 的两种方言。这些方言称作 Transact-SQL—Adaptive Server Enterprise 的方言和 Watcom-SQL—SQL Anywhere 的方言。
- 过程、触发器或批处理将以 Transact-SQL 或 Watcom-SQL 方言执行。在整个批处理或过程中，必须仅使用一种方言的控制语句。例如，每种方言都有不同的流控制语句。

下图说明了这两种方言的重叠情况。



相似之处和差异

在处理现有数据时，SQL Anywhere 支持大多数的 Transact-SQL 语言元素、函数和语句。例如，SQL Anywhere 支持所有数值函数、集合函数、日期和时间函数以及除了一个字符串函数之外的所有字符串函数。又比如，SQL Anywhere 还支持使用连接的扩展 DELETE 和 UPDATE 语句。

另外，SQL Anywhere 还支持大多数的 Transact-SQL 存储过程语言（CREATE PROCEDURE 和 CREATE TRIGGER 语法、控制语句等）以及 Transact-SQL 数据定义语言语句的许多（但不是所有）方面。

每种产品所支持的体系结构和配置功能存在着设计上的差异。设备管理、用户管理和维护任务（如备份）往往是因系统而异的。即使在这种情况下，SQL Anywhere 仍将 Transact-SQL 系统表作为视图来提供（在视图中，在 SQL Anywhere 中没有任何意义的表将不显示任何行）。此外，SQL Anywhere 为某些常见的管理任务提供了一组系统过程。

在讨论这些方言的数据操作和数据定义语言等兼容性高的方面之前，本章将首先讨论一些差异最为明显的系统层面上的问题。

仅限 Transact-SQL

SQL Anywhere 所支持的某些 SQL 语句只在一种方言中使用，而不在另一种方言中使用。您不能在一个过程、触发器或批处理中混用两种方言。例如，SQL Anywhere 支持以下语句，但这些语句仅用于 Transact-SQL 方言：

- Transact-SQL 控制语句 IF 和 WHILE
- Transact-SQL 的 EXECUTE 语句
- Transact-SQL 的 CREATE PROCEDURE 和 CREATE TRIGGER 语句

- Transact-SQL 的 BEGIN TRANSACTION 语句
- Transact-SQL 过程或批处理中可以使用*未*以分号分隔的 SQL 语句

仅限 SQL Anywhere

Adaptive Server Enterprise 不支持以下语句：

- 控制语句 CASE、LOOP 和 FOR
- IF 和 WHILE 的 SQL Anywhere 版本
- CALL 语句
- CREATE PROCEDURE、CREATE FUNCTION 和 CREATE TRIGGER 语句的 SQL Anywhere 版本
- 由分号分隔的 SQL 语句

注意

不能在一个过程、触发器或批处理中混用两种方言。这意味着：

- 可以在一个批处理、过程或触发器中同时包含仅限 Transact-SQL 的语句以及属于两种方言的语句。
- 可以在一个批处理、过程或触发器中同时包含 Adaptive Server Enterprise 不支持的语句以及两种服务器都支持的语句。
- 不能在一个批处理、过程或触发器中同时包含仅限 Transact-SQL 的语句和仅限 SQL Anywhere 的语句。

Adaptive Server Enterprise 体系结构

Adaptive Server Enterprise 和 SQL Anywhere 是互补的产品，其体系结构分别满足不同的需求。

本节将介绍 Adaptive Server Enterprise 和 SQL Anywhere 在体系结构上的差异。还将介绍 SQL Anywhere 为达到数据库管理的兼容性而包含的类似于 Adaptive Server Enterprise 的工具。

服务器和数据库

Adaptive Server Enterprise 中服务器和数据库之间的关系不同于 SQL Anywhere 中服务器和数据库之间的关系。

在 Adaptive Server Enterprise 中，每个数据库都存在于服务器中，而每个服务器也都可以包含多个数据库。用户可以具有服务器的登录权限并且可以连接到服务器。连接之后，用户可以使用该服务器上他们对其具有权限的每个数据库。全系统范围的系统表保存在 master 数据库中，这些表包含服务器上所有数据库共有的信息。

SQL Anywhere 上没有 master 数据库

在 SQL Anywhere 中，不存在与 Adaptive Server Enterprise 的 master 数据库相对应的级别。相反，每个数据库都是一个独立的实体，它们包含所有系统表。用户可以具有数据库（而不是服务器）的连接权限。当用户建立连接后，他们将连接到单独的数据库。没有在 master 数据库级别维护的全系统范围的系统表集。每个 SQL Anywhere 数据库服务器都可以动态地装载和卸载多个数据库，而用户可以在每个数据库上建立独立的连接。

SQL Anywhere 为支持 Transact-SQL 和 Open Server 提供了一些工具，以便按照与 Adaptive Server Enterprise 类似的方式执行某些任务。例如，SQL Anywhere 提供了 Adaptive Server Enterprise sp_addlogin 系统过程的实现，它执行几乎是等效的操作：将用户添加到数据库中。请参见“将 SQL Anywhere 用作 Open Server” 《SQL Anywhere 服务器 - 数据库管理》。

文件操作语句

对于备份和恢复，SQL Anywhere 不支持 Transact-SQL 的 DUMP DATABASE 和 LOAD DATABASE 语句。SQL Anywhere 具有其自己的 BACKUP DATABASE 和 RESTORE DATABASE 语句，但这些语句的语法不同。

设备管理

SQL Anywhere 和 Adaptive Server Enterprise 使用不同的模型来管理设备和磁盘空间，这反映了这两种产品的不同用途。Adaptive Server Enterprise 使用大量的 Transact-SQL 语句制定出一套综合的资源管理方案，而 SQL Anywhere 则自动管理它自己的资源，其数据库是常规的操作系统文件。

SQL Anywhere 不支持 Transact-SQL 的 DISK 语句，例如 DISK INIT、DISK MIRROR、DISK REFIT、DISK REINIT、DISK REMIRROR 和 DISK UNMIRROR。

有关磁盘管理的信息，请参见“使用数据库文件” 《SQL Anywhere 服务器 - 数据库管理》。

缺省值和规则

SQL Anywhere 不支持 Transact-SQL 的 CREATE DEFAULT 语句或 CREATE RULE 语句。CREATE DOMAIN 语句用于将缺省值和规则（称作 CHECK 条件）合并为域的定义，因此提供了与 Transact-SQL 的 CREATE DEFAULT 和 CREATE RULE 语句相似的功能。

在 SQL Anywhere 中，域可以有与之关联的缺省值和 CHECK 条件（它们将应用于在该数据类型上定义的所有列）。可以使用 CREATE DOMAIN 语句来创建域。

您可以使用 CREATE TABLE 语句或 ALTER TABLE 语句为单独的列定义缺省值和规则（或 CHECK 条件）。

在 Adaptive Server Enterprise 中，CREATE DEFAULT 语句创建指定的缺省值。通过使用 sp_bindefault 系统过程，将此缺省值绑定到特定的列，此缺省值可以用作列的缺省值；或者将此缺省值绑定到数据类型，此缺省值可以用作域中所有列的缺省值。CREATE RULE 语句创建指定的规则，该规则可用于定义列的域（方法是将此规则绑定到特定的列）或者用作域中所有列的规则（方法是将此规则绑定到数据类型）。规则是使用 sp_bindrule 系统过程来绑定到数据类型或列的。

另请参见

- “CREATE DOMAIN 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “CREATE TABLE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “ALTER TABLE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “搜索条件”一节 《SQL Anywhere 服务器 - SQL 参考》

系统表

除了自己的系统表之外，SQL Anywhere 还提供了一组系统视图，这些视图模拟 Adaptive Server Enterprise 系统表中的相关部分。

有关列表和各个说明，包括两种产品的系统目录说明，请参见“Transact-SQL 兼容性视图”一节 《SQL Anywhere 服务器 - SQL 参考》。

SQL Anywhere 系统表完全位于每个数据库中，而 Adaptive Server Enterprise 系统表则部分位于每个数据库中，部分位于 master 数据库中。SQL Anywhere 体系结构中不包含 master 数据库。

在 Adaptive Server Enterprise 中，数据库所有者（用户 dbo）拥有系统表。而在 SQL Anywhere 中，系统所有者（用户 SYS）拥有系统表。用户 dbo 拥有 SQL Anywhere 提供的与 Adaptive Server Enterprise 兼容的系统视图。

管理角色

与 SQL Anywhere 相比，Adaptive Server Enterprise 具有一组更为详细的管理角色。虽然可以向 Adaptive Server Enterprise 上的多个登录帐户授予任意角色，而且一个帐户可以拥有多个角色，但 Adaptive Server Enterprise 中有一组不应重复的角色。

Adaptive Server Enterprise 的角色

在 Adaptive Server Enterprise 中，不应重复的角色包括：

- **系统管理员** 负责与特定应用程序无关的常规管理任务；可以访问任何数据库对象。
- **系统安全员** 负责 Adaptive Server Enterprise 中与安全相关的任务，但对数据库对象不具有特殊的权限。
- **数据库所有者** 对自己拥有的数据库中的对象具有完全权限，可以将用户添加到数据库中并向其他用户授予在数据库中创建对象和执行命令的权限。
- **数据定义语句** 可以向用户授予执行特定的数据定义语句（CREATE TABLE 或 CREATE VIEW）的权限，使用户能够创建数据库对象。
- **对象所有者** 每个数据库对象都有一个所有者，该所有者可以向其他用户授予访问该对象的权限。对象的所有者自动具有该对象的各种权限。

在 SQL Anywhere 中，以下全数据库范围的权限具有管理角色：

- 与 Adaptive Server Enterprise 数据库所有者类似，数据库管理员（DBA 权限）对该数据库中的所有对象（SYS 所拥有的对象除外）具有完全权限，并且可以向其他用户授予在该数据库中创建对象和执行命令的权限。缺省的数据库管理员是用户 DBA。
- RESOURCE 权限允许用户在数据库中创建任意类型的对象。它取代 Adaptive Server Enterprise 中授予单独的 CREATE 语句权限的方案。
- SQL Anywhere 的对象所有者与 Adaptive Server Enterprise 的对象所有者的权限相同。对象的所有者自动具有该对象上的任何权限（包括授予权限的权利）。

为了顺利地访问 Adaptive Server Enterprise 和 SQL Anywhere 中保存的数据，应该在数据库中创建具有适当权限（在 SQL Anywhere 中为 RESOURCE 权限，或者在 Adaptive Server Enterprise 中为执行单个 CREATE 语句的权限）的用户 ID，并利用该用户 ID 创建对象。如果在每个环境中都使用相同的用户 ID，对象名和限定符可以在这两种数据库中完全相同，从而可确保能进行兼容的访问。

另请参见

- “数据库权限和特权概述”一节 《SQL Anywhere 服务器 - 数据库管理》
- “DBA 特权”一节 《SQL Anywhere 服务器 - 数据库管理》
- “RESOURCE 特权”一节 《SQL Anywhere 服务器 - 数据库管理》

用户和组

Adaptive Server Enterprise 和 SQL Anywhere 在用户和组模式上存在着一些差异。

在 Adaptive Server Enterprise 中，用户连接到某个服务器。每个用户都需要该服务器的登录 ID 和口令以及该服务器上他们要访问的每个数据库的用户 ID。数据库的每个用户只能是一个组的成员。

在 SQL Anywhere 中，用户可以直接连接到数据库，并且不需要单独的登录 ID 即可连接到数据库服务器。实际上，每个用户都会接收数据库上的用户 ID 和口令，这样他们便可以使用该数据库。用户可以是许多组的成员，并且组的层次是允许存在的。

这两种服务器都支持组，所以您可以一次向多个用户授予权限。但是，这两种服务器中的组在某些细节上存在差异。例如，Adaptive Server Enterprise 只允许每个用户成为一个组的成员，而 SQL Anywhere 没有这种限制。有关具体信息，请比较这两种产品中有关用户和组的文档。

Adaptive Server Enterprise 和 SQL Anywhere 都具有公共组，该组用于定义缺省权限。每个用户都会自动成为公共组的成员。

SQL Anywhere 支持使用以下 Adaptive Server Enterprise 系统过程来管理用户和组。请参见“Adaptive Server Enterprise 系统和分类过程”一节 《SQL Anywhere 服务器 - SQL 参考》。

系统过程	说明
sp_addlogin	在 Adaptive Server Enterprise 中，将用户添加到服务器。在 SQL Anywhere 中，将用户添加到数据库。
sp_adduser	在 Adaptive Server Enterprise 和 SQL Anywhere 中，将用户添加到数据库。在 Adaptive Server Enterprise 中，它与 sp_addlogin 是截然不同的两个任务，但在 SQL Anywhere 中，它们是相同的。
sp_addgroup	将组添加到数据库中。
sp_changegroup	将用户添加到组中，或者将用户从一个组移到另一个组。
sp_droplogin	在 Adaptive Server Enterprise 中，从服务器中删除用户。在 SQL Anywhere 中，从数据库中删除用户。
sp_dropuser	从数据库中删除用户。
sp_dropgroup	从数据库中删除组。

在 Adaptive Server Enterprise 中，登录 ID 是全服务器范围的。在 SQL Anywhere 中，用户却属于单个的数据库。

数据库对象权限

Adaptive Server Enterprise 和 SQL Anywhere 在授予特定数据库对象的权限时所使用的 GRANT 和 REVOKE 语句非常相似。它们都允许在数据库表和视图上拥有 SELECT、INSERT、DELETE、UPDATE 和 REFERENCES 权限以及在数据库表的选定列拥有 UPDATE 权限。它们都允许在存储过程上授予 EXECUTE 权限。

例如，以下语句在 Adaptive Server Enterprise 和 SQL Anywhere 中都有效：

```
GRANT INSERT, DELETE
ON Employees
TO MARY, SALES;
```

此语句向用户 MARY 和 SALES 组授予对 Employees 表使用 INSERT 语句和 DELETE 语句的权限。

虽然 SQL Anywhere 不允许在 GRANT EXECUTE 语句上使用 WITH GRANT OPTION，但 SQL Anywhere 和 Adaptive Server Enterprise 都支持 WITH GRANT OPTION 子句，该子句允许权限的接收者向其他用户授予这种权限。在 SQL Anywhere 中，您只能为用户指定 WITH GRANT OPTION。如果将 WITH GRANT OPTION 授予组，则组的成员不会继承 WITH GRANT OPTION。

全数据库范围的权限

对于全数据库范围的用户权限，Adaptive Server Enterprise 和 SQL Anywhere 使用不同的模式。SQL Anywhere 使用 DBA 权限来向用户授予对数据库的完全权限。Adaptive Server Enterprise 中的系统管理员享有服务器上所有数据库的这一权限。但是，SQL Anywhere 数据库上的 DBA 权限不同于 Adaptive Server Enterprise 数据库所有者的权限，后者必须使用 Adaptive Server Enterprise 的 SETUSER 语句获取对其他用户拥有的对象的权限。请参见“用户和组”一节第 617 页。

SQL Anywhere 使用 RESOURCE 权限来授予用户在数据库中创建对象的权限。相应的 Adaptive Server Enterprise 权限为 GRANT ALL，它由数据库所有者来使用。

为实现 Transact-SQL 兼容性配置数据库

通过在创建数据库时（或者如果正在处理现有的数据库，则是在重建数据库时）选择相应的选项，可以排除 SQL Anywhere 和 Adaptive Server Enterprise 之间的一些行为差异。其它差异则可以通过用 SET TEMPORARY OPTION 语句（在 SQL Anywhere 中）或 SET 语句（在 Adaptive Server Enterprise 中）设置连接级别的选项来进行控制。

创建与 Transact-SQL 兼容的数据库

本节将介绍您在创建或重建数据库时必须做出的选择。

快速入门

下面是创建与 Transact-SQL 兼容的数据库时所需要执行的步骤。本节的其余部分将介绍您需要设置的选项。

♦ **创建与 Transact-SQL 兼容的数据库 (Sybase Central)**

1. 启动 Sybase Central。
2. 选择 [**工具**] » [**SQL Anywhere 11**] » [**创建数据库**]。
3. 请按照向导中的说明进行操作。
4. 当您看到一个名为 [**模拟 Adaptive Server Enterprise**] 的按钮时，单击该按钮，然后单击 [**下一步**]。
5. 请按照向导中的其余说明进行操作。

♦ **创建与 Transact-SQL 兼容的数据库（命令行）**

- 运行以下 dbinit 命令：

```
dbinit -b -c -k db-name.db
```

有关这些选项的详细信息，请参见“初始化实用程序 (dbinit)”一节 《SQL Anywhere 服务器 - 数据库管理》。

♦ **创建与 Transact-SQL 兼容的数据库 (SQL)**

1. 连接到任一 SQL Anywhere 数据库。
2. 例如，在 Interactive SQL 中输入以下语句：

```
CREATE DATABASE 'dbname.db'
ASE COMPATIBLE
CASE RESPECT
BLANK PADDING ON;
```

在此语句中，ASE COMPATIBLE 表示与 Adaptive Server Enterprise 兼容。它会防止创建 SYS.SYSCOLUMNS 和 SYS.SYSINDEXES 视图。

使数据库区分大小写

缺省情况下，Adaptive Server Enterprise 数据库中的字符串比较是区分大小写的，而 SQL Anywhere 中的字符串比较却不区分大小写。

在使用 SQL Anywhere 建立与 Adaptive Server Enterprise 兼容的数据库时，请选中区分大小写选项。

- 如果您在使用 Sybase Central，则此选项位于 [**创建数据库向导**] 中。
- 如果要使用 dbinit 实用程序，请指定 -c 选项。

在比较中忽略尾随空白

在使用 SQL Anywhere 建立与 Adaptive Server Enterprise 兼容的数据库时，请选择在比较中忽略尾随空白的选项。

- 如果您在使用 Sybase Central，则此选项位于 [**创建数据库向导**] 中。
- 如果要使用 dbinit 实用程序，请指定 -b 选项。

当您选择此选项后，Adaptive Server Enterprise 和 SQL Anywhere 会认为以下两个字符串是相同的：

```
'ignore the trailing blanks   '
'ignore the trailing blanks'
```

如果您不选择此选项，SQL Anywhere 就会认为上面的两个字符串不相同。

选择此选项的副作用是当客户端应用程序读取字符串时，字符串将被填上空白。

删除历史系统视图

较早的 SQL Anywhere 版本使用的两种系统视图的名称与为实现兼容性而提供的 Adaptive Server Enterprise 系统视图的名称相冲突。这两种视图包括 SYSCOLUMNS 和 SYSINDEXES。如果您在使用 Open Client 或 JDBC 接口，请在创建数据库时排除这两种视图。可使用 dbinit -k 选项实现这一操作。

如果您在创建数据库时没有使用这一选项，执行语句 `SELECT * FROM SYSCOLUMNS;` 将导致错误 [**表名 'SYSCOLUMNS' 不明确**]。

为实现 Transact-SQL 兼容性设置选项

可以使用 SET OPTION 语句来设置 SQL Anywhere 数据库选项。有几项数据库选项设置与 Transact-SQL 的行为相关。

设置 allow_nulls_by_default 选项

缺省情况下，如果您没有显式定义新列允许 NULL，Adaptive Server Enterprise 就不允许新列中出现 NULL。SQL Anywhere 在缺省情况下允许在新列中出现 NULL，这符合 SQL/2003 ISO 标准。

要使 Adaptive Server Enterprise 以符合 SQL/2003 的方式运行，请使用 sp_dboption 系统过程将 allow_nulls_by_default 选项设置为 true。

要使 SQL Anywhere 以与 Transact-SQL 兼容的方式运行，请将 allow_nulls_by_default 选项设置为 Off。如下使用 SET OPTION 语句就可以达到这一目的：

```
SET OPTION PUBLIC.allow_nulls_by_default = 'Off';
```

设置 quoted_identifier 选项

缺省情况下，Adaptive Server Enterprise 处理标识符和字符串的方式与 SQL Anywhere 有所不同，后者符合 SQL/2003 ISO 标准。

quoted_identifier 选项在 Adaptive Server Enterprise 和 SQL Anywhere 中均可用。为了以兼容的方式处理标识符和字符串，应确保在这两种数据库中将该选项设置为相同的值。请参见“quoted_identifier 选项 [兼容性]”一节 《SQL Anywhere 服务器 - 数据库管理》。

要想获得 SQL/2003 行为，请在 Adaptive Server Enterprise 和 SQL Anywhere 中均将 quoted_identifier 选项设置为 On。

要想获得 Transact-SQL 行为，请在 Adaptive Server Enterprise 和 SQL Anywhere 中均将 quoted_identifier 选项设置为 Off。如果您选择此设置，则不能在双引号中使用与关键字相同的标识符。作为将 quoted_identifier 设置为 Off 的替代选择，确保将在应用程序的 SQL 语句中使用的所有字符串括在单引号而不是双引号中。

设置 string_rtruncation 选项

Adaptive Server Enterprise 和 SQL Anywhere 都支持 string_rtruncation 选项，该选项会影响在 INSERT 或 UPDATE 字符串被截断时所报告的错误消息。请确保在每种数据库中将该选项设置为相同的值。请参见“string_rtruncation 选项 [兼容性]”一节 《SQL Anywhere 服务器 - 数据库管理》。

请参见“兼容性选项”一节 《SQL Anywhere 服务器 - 数据库管理》。

区分大小写

数据库中的是否区分大小写的特性指的是：

- **数据** 数据的区分大小写会反映在索引等内容中。
- **标识符** 标识符包括表名、列名等。
- **口令** 在 SQL Anywhere 数据库中，口令始终区分大小写。

数据的区分大小写特性

在创建数据库时，应确定 SQL Anywhere 数据在比较时是否区分大小写。缺省情况下，虽然数据始终保持输入时的大小写，但 SQL Anywhere 数据库在比较时不区分大小写。

Adaptive Server Enterprise 是否区分大小写取决于在 Adaptive Server Enterprise 系统上安装的排序顺序。通过重新配置 Adaptive Server Enterprise 排序顺序，可以更改单字节字符集的区分大小写特性。

标识符的区分大小写特性

SQL Anywhere 不支持区分大小写的标识符。在 Adaptive Server Enterprise 中，标识符的区分大小写特性遵循数据的区分大小写特性。数据库的缺省用户 ID 是 DBA。

在 Adaptive Server Enterprise 中，域名是区分大小写的。在 SQL Anywhere 中，除 Java 数据类型之外，域名不区分大小写。

口令的区分大小写特性

在 SQL Anywhere 中，口令始终区分大小写。DBA 用户 ID 的缺省口令都是小写字母的 **sql**。

在 Adaptive Server Enterprise 中，用户 ID 和口令的区分大小写特性遵循服务器的区分大小写特性。

确保兼容的对象名

每个数据库对象在特定的**命名空间**中都必须有唯一的名称。在此命名空间之外，允许出现重复的名称。某些数据库对象在 Adaptive Server Enterprise 和 SQL Anywhere 中占用不同的命名空间。

对于触发器名称，Adaptive Server Enterprise 的命名空间限制比 SQL Anywhere 的更严格。触发器名称在数据库中必须是唯一的。为了达到 SQL 兼容性，您应该遵守 Adaptive Server Enterprise 的限制并确保触发器名称在数据库中是唯一的。

特殊的 Transact-SQL 时间戳列和数据类型

SQL Anywhere 支持 Transact-SQL 的特殊时间戳列。时间戳列与 tsequal 系统函数一起检查行是否已更新。

时间戳的两种含义
SQL Anywhere 具有 TIMESTAMP 数据类型，它保存精确的日期和时间信息。它不同于特殊的 Transact-SQL TIMESTAMP 列和数据类型。

在 SQL Anywhere 中创建 Transact-SQL 时间戳列

要创建 Transact-SQL 时间戳列，请创建一个 (SQL Anywhere) 数据类型为 TIMESTAMP 且具有缺省时间戳设置的列。虽然名称 timestamp 比较常用，但该列可以具有任何名称。

例如，以下 CREATE TABLE 语句包含一个 Transact-SQL 时间戳列：

```
CREATE TABLE tablename (
   column_1 INTEGER,
   column_2 TIMESTAMP DEFAULT TIMESTAMP
);
```

以下 ALTER TABLE 语句将一个 Transact-SQL 时间戳列添加到 SalesOrders 表中：

```
ALTER TABLE SalesOrders
ADD timestamp TIMESTAMP DEFAULT TIMESTAMP;
```

在 Adaptive Server Enterprise 中，名称为 timestamp 且未指定数据类型的列会自动接收 TIMESTAMP 数据类型。在 SQL Anywhere 中，您必须自己显式指定数据类型。

时间戳列的数据类型

Adaptive Server Enterprise 将时间戳列当作允许 NULL 的 VARBINARY(8) 域，而 SQL Anywhere 将时间戳列当作 TIMESTAMP 数据类型，它包括日期和时间（秒的小数部分占六个小数位）。

在为随后的更新从表中读取时，读取时间戳值的变量应该与列说明相对应。

在 Interactive SQL 中，您可能需要设置 timestamp_format 选项以查看行值的差异。以下语句将 timestamp_format 选项设置为显示秒的小数部分的所有六位数字：

```
SET OPTION timestamp_format='YYYY-MM-DD HH:NN:SS.SSSSSS';
```

如果所有六位都未显示，则某些时间戳列值可能看起来相同，但实际上却不同。

将 tsequal 用于更新

利用 tsequal 系统函数，您可以确定时间戳列是否已得到更新。

例如，某一应用程序可能用 SELECT 将一个时间戳列添加到变量中。当某一选定行的 UPDATE 被提交时，它可能会使用 tsequal 函数检查该行是否已被修改。tsequal 函数将比较表中的时间戳值和在 SELECT 中获取的时间戳值。时间戳相同意味着没有变化。如果时间戳不同，则意味着该行在执行 SELECT 后出现了变化。

使用 tsequal 函数的典型 UPDATE 语句如下所示：

```
UPDATE publishers
SET City = 'Springfield'
WHERE pub_id = '0736'
AND TSEQUAL(timestamp, '2005/10/25 11:08:34.173226');
```

tsequal 函数的第一个参数是特殊时间戳列的名称，第二个参数是在 SELECT 语句中检索的时间戳。在嵌入式 SQL 中，第二个参数可能是一个主机变量，包含该列上进行最近的 FETCH 操作后的 TIMESTAMP 值。

特殊 IDENTITY 列

IDENTITY 列存储顺序编号，如自动生成的发票编号和雇员编号。IDENTITY 列的值唯一地标识表中的每一行。

在 Adaptive Server Enterprise 中，数据库中的每个表只能有一个 IDENTITY 列。其数据类型必须是不带小数位的数值，而且 IDENTITY 列不应允许空值。

在 SQL Anywhere 中，IDENTITY 列是列缺省设置。您可以使用 INSERT 语句将不在序列中的值显式地插入列中。如果 identity_insert 选项不是 *on*，Adaptive Server Enterprise 将不允许使用 INSERT 将值插入 IDENTITY 列。在 SQL Anywhere 中，您需要自己设置 NOT NULL 属性并确保只有一个列是 IDENTITY 列。SQL Anywhere 允许任何数字数据类型成为 IDENTITY 列。推荐使用整数数据类型以获得更佳性能。

在 SQL Anywhere 中，IDENTITY 列和某列的 AUTOINCREMENT 缺省设置是相同的。

要创建 IDENTITY 列，请使用以下 CREATE TABLE 语法，其中 *n* 足够大，可以容纳可能在表中插入的最大行数的值：

```
CREATE TABLE table-name (
   ...
   column-name numeric(n,0) IDENTITY NOT NULL,
   ...
)
```

使用 @@identity 检索 IDENTITY 列值

当您第一次将行插入表中时，IDENTITY 列将被分配值 1。对于每次后继的插入，该列的值都会递增 1。最近插入标识列的值可在 @@identity 全局变量中提供。

有关 @@identity 的行为的详细信息，请参见“@@identity 全局变量”一节 《SQL Anywhere 服务器 - SQL 参考》。

编写兼容的 SQL 语句

本节提供了有关编写将在多个数据库管理系统上使用的 SQL 的一般原则，并讨论 Adaptive Server Enterprise 和 SQL Anywhere 在 SQL 语句级别的兼容性问题。

编写可移植的 SQL 的一般原则

在编写将在多个数据库管理系统上使用的 SQL 语句时，应该使您的 SQL 语句尽可能地清晰。即使多个服务器支持给定的 SQL 语句，每个系统上的缺省行为也可能不相同。

在 SQL Anywhere 中，数据库服务器和 SQL 预处理器 (sqlpp) 可标识作为服务商扩充、不符合特定 ISO/ANSI SQL 标准、或者 UltraLite 不支持的 SQL 语句。此功能称为 SQL Flagger。请参见“使用 SQL Flagger 测试 SQL 遵从性”一节第 607 页。

编写兼容的 SQL 语句适用的一般原则包括：

- 包括所有可用选项，而不是使用缺省行为。
- 使用括号来明确语句中的执行顺序，而不是对运算符采用相同的缺省优先级。
- 使用在变量名前加 @ 符号这一 Transact-SQL 约定，以实现 Adaptive Server Enterprise 可移植性。
- 在过程、触发器和批处理中声明变量和游标要紧跟在 BEGIN 语句之后。这是 SQL Anywhere 的要求，尽管 Adaptive Server Enterprise 允许在过程、触发器或批处理中的任意位置进行声明。
- 避免将 Adaptive Server Enterprise 或 SQL Anywhere 中的保留字用作数据库中的标识符。
- 采用大命名空间。例如，要确保每个索引都应有唯一的名称。

创建兼容表

SQL Anywhere 支持那些允许在数据类型定义中封装约束和缺省定义的域。它还支持在 CREATE TABLE 语句中使用显式缺省值和 CHECK 条件。但是，它不支持命名的缺省值。

NULL

SQL Anywhere 和 Adaptive Server Enterprise 在处理 NULL 的某些方面存在差异。在 Adaptive Server Enterprise 中，有时将 NULL 视为一个值处理。

例如，Adaptive Server Enterprise 中的唯一索引不能包含具有空值的行，而在其它方面都相同。在 SQL Anywhere 中，唯一索引则可以包含这样的行。

缺省情况下，Adaptive Server Enterprise 中的列缺省为 NOT NULL，而 SQL Anywhere 中的缺省设置为 NULL。您可以使用 allow_nulls_by_default 选项控制此设置。通过显式指定 NULL 或 NOT NULL，可以使您的数据定义语句成为可移植的语句。

有关此选项的信息，请参见“为实现 Transact-SQL 兼容性设置选项”一节第 621 页。

临时表

您可以通过在 CREATE TABLE 语句中的表名前放置井号 (#) 来创建临时表。这些临时表是 SQL Anywhere 声明的临时表，并且只能在当前连接中才可用。

在 Adaptive Server Enterprise 中表的实际放置方式不同于在 SQL Anywhere 中表的实际放置方式。SQL Anywhere 支持 **ON** *segment-name* 子句，但 *segment-name* 会引用 SQL Anywhere dbspace。

另请参见

- “使用 SQL Flagger 测试 SQL 遵从性”一节第 607 页
- “DECLARE LOCAL TEMPORARY TABLE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “CREATE TABLE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》

编写兼容的查询

若要编写在 SQL Anywhere 数据库和 Adaptive Server Enterprise 数据库上都能运行的查询，应遵循下面两条标准：

- 查询中的数据类型、表达式和搜索条件必须兼容。
- SELECT 语句本身的语法必须兼容。

本节将介绍兼容的 SELECT 语句语法，并说明兼容的数据类型、表达式和搜索条件。此示例假定 quoted_identifier 设置为 Off：这是 Adaptive Server Enterprise 的缺省设置，但不是 SQL Anywhere 的缺省设置。

SQL Anywhere 支持 Transact-SQL SELECT 语句的以下子集：

语法

```
SELECT [ ALL | DISTINCT ] select-list
...[ INTO #temporary-table-name ]
...[ FROM table-spec [ HOLDLOCK | NOHOLDLOCK ],
...  table-spec [ HOLDLOCK | NOHOLDLOCK ], ... ]
...[ WHERE search-condition ]
...[ GROUP BY column-name, ... ]
...[ HAVING search-condition ]
   [ ORDER BY { expression | integer }
      [ ASC | DESC ], ... ]
```

参数

```
select-list:
 table-name.*
| *
| expression
| alias-name = expression
| expression as identifier
| expression as string

table-spec:
[ owner . ]table-name
```

...[[**AS**] *correlation-name*]
...[**(INDEX** *index_name* [**PREFETCH** *size*][**LRU** | **MRU**] **)**]

alias-name:
identifier | '*string*' | "*string*"

SQL Anywhere 不支持 Transact-SQL SELECT 语句语法的以下关键字和子句：

- SHARED 关键字
- COMPUTE 子句
- FOR BROWSE 子句
- FOR UPDATE 子句
- GROUP BY ALL 子句

注意

- SQL Anywhere 不支持对 GROUP BY 子句的 Transact-SQL 扩展（它允许引用未用于创建组的列和表达式）。在 Adaptive Server Enterprise 中，此扩展会生成摘要报告。
- 将会对表的说明的性能参数部分进行语法分析，但不会产生任何影响。
- SQL Anywhere 支持 HOLDLOCK 关键字。利用 HOLDLOCK，指定表或视图上的共享锁会更为严格，因为当不再需要数据页时不会释放共享锁。对于已为其指定了 HOLDLOCK 的表，查询将在隔离级别 3 执行。
- HOLDLOCK 选项仅应用于指定该选项的表或视图，并且仅应用于该选项所在的语句所定义的事务期间。如果将隔离级别设置为 3，则会将 HOLDLOCK 应用于事务中的每一个 SELECT 语句。您不能在一个查询中同时指定 HOLDLOCK 和 NOHOLDLOCK 选项。
- SQL Anywhere 识别 NOHOLDLOCK 关键字，但该关键字不会产生任何影响。
- Transact-SQL 使用 SELECT 语句向局部变量赋值：

  ```
  SELECT @localvar = 42;
  ```

  SQL Anywhere 中的相应语句为 SET 语句：

  ```
  SET @localvar = 42;
  ```

- Adaptive Server Enterprise 不支持 SELECT 语句语法的以下子句：
  - INTO *host-variable-list*
  - INTO *variable-list*
  - 带括号的查询
- Adaptive Server Enterprise 在 WHERE 子句中使用连接运算符，而不使用 FROM 子句和 ON 条件来进行连接。

另请参见

- “使用 SQL Flagger 测试 SQL 遵从性”一节第 607 页
- “SELECT 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “OLAP 支持”第 423 页

连接的兼容性

在 Transact-SQL 中，WHERE 子句中的连接的语法如下：

start of select, update, insert, delete, or subquery
  FROM { *table-list* | *view-list* } **WHERE** [**NOT**]
  [*table-name*.| *view name*.]*column-name*
    join-operator
  [*table-name*.| *view-name*.]*column_name*
  [{ **AND** | **OR** } [**NOT**]
  [*table-name*.| *view-name*.]*column_name*
    join-operator
  [*table-name*.| *view-name*.]*column-name*]...
end of select, update, insert, delete, or subquery

WHERE 子句中的 *join-operator* 可以是任意比较运算符，或者也可以是以下两个**外连接运算符**之一：

- ***=** 左外连接运算符
- **=*** 右外连接运算符

SQL Anywhere 支持 Transact-SQL 外连接运算符作为本地 SQL/2003 语法的替代语法。不能在一个查询中混用这些方言。这一规则也适用于查询使用的视图—视图上的外连接查询必须遵循视图定义查询所使用的方言。

注意
不建议使用 Transact-SQL 外连接运算符 *= 和 =*，它们将在以后的版本中删除。

有关 SQL Anywhere 中的连接和 ANSI/ISO SQL 标准中的连接的信息，请参见“连接：从多个表检索数据”第 361 页和“FROM 子句”一节 《SQL Anywhere 服务器 - SQL 参考》。

有关连接的 Transact-SQL 兼容性的详细信息，请参见“Transact-SQL 外连接（*= 或 =*）”一节第 377 页。

另请参见

- “使用 SQL Flagger 测试 SQL 遵从性”一节第 607 页

Transact-SQL 过程语言概述

存储过程语言是 SQL 语言中用于存储过程、触发器和批处理的那一部分。

除了基于 SQL/2003 的 Watcom-SQL 方言之外，SQL Anywhere 还支持 Transact-SQL 存储过程语言中相当大的一部分。

Transact-SQL 存储过程概述

基于 ISO/ANSI 标准草案的 SQL Anywhere 存储过程语言在许多方面不同于 Transact-SQL 方言。它们的许多概念和功能都相似，但语法却不同。SQL Anywhere 对 Transact-SQL 的支持是通过提供方言之间的自动转换来利用类似的概念的。但是，过程必须仅使用两种方言中的一种来编写，而不能混用这两种方言。

SQL Anywhere 对 Transact-SQL 存储过程的支持

SQL Anywhere 对 Transact-SQL 存储过程的支持包括多个方面，例如：

- 传递参数
- 返回结果集
- 返回状态信息
- 提供参数的缺省值
- 控制语句
- 错误处理
- 用户定义函数

Transact-SQL 触发器概述

触发器兼容性需要触发器功能和语法的兼容性。本节提供有关 Transact-SQL 和 SQL Anywhere 触发器的功能兼容性的概述。

Adaptive Server Enterprise 支持语句级 AFTER 触发器；即，在触发语句完成后执行的触发器。SQL Anywhere 支持行级 BEFORE、AFTER 和 INSTEAD OF 触发器，以及语句级 AFTER 和 INSTEAD OF 触发器。请参见“触发器简介”一节第 789 页。

行级触发器不属于 Transact-SQL 兼容性功能，在“使用过程、触发器和批处理”第 777 页中会讨论行级触发器。

有关不支持或不同的 Transact-SQL 触发器的说明

SQL Anywhere 中不支持的或不同的 Transact-SQL 触发器功能包括：

- **触发器触发其它触发器** 假定触发器执行将触发其它触发器的操作（在该操作由用户直接执行的情况下）。对于这种情况，SQL Anywhere 和 Adaptive Server Enterprise 的响应方式略有不同。缺省情况下，Adaptive Server Enterprise 中的触发器最多能够引发可配置的嵌套级别（缺省值为 16）的其它触发器。您可以使用 Adaptive Server Enterprise 的嵌套触发器选项来控制嵌套级别。在 SQL Anywhere 中，触发器可以毫无限制地触发其它触发器（除非内存不足）。

- **触发器自行触发** 假定触发器执行将触发同一触发器的操作（在该操作由用户直接执行的情况下）。对于这种情况，SQL Anywhere 和 Adaptive Server Enterprise 的响应方式略有不同。缺省情况下，在 SQL Anywhere 中，非 Transact-SQL 触发器以递归方式自行触发，而 Transact-SQL 方言触发器则不以递归方式自行触发。然而，对于 Transact-SQL 方言触发器，您可以使用 SET 语句 [T-SQL] 的 self_recursion 选项，来允许触发器以递归方式调用其自身。请参见“SET 语句 [T-SQL]”一节 《SQL Anywhere 服务器 - SQL 参考》。

  缺省情况下，Adaptive Server Enterprise 中的触发器不以递归方式调用其自身，但您可以使用 self_recursion 选项允许递归发生。

- **触发器中的 ROLLBACK 语句** Adaptive Server Enterprise 允许在触发器中使用 ROLLBACK TRANSACTION 语句来回退触发器所属的整个事务。由于触发操作及其触发器一起形成一个原子语句，SQL Anywhere 不允许在触发器中使用 ROLLBACK（或 ROLLBACK TRANSACTION）语句。

  但 SQL Anywhere 提供了与 Adaptive Server Enterprise 兼容的 ROLLBACK TRIGGER 语句，以撤消触发器中的操作。请参见“ROLLBACK TRIGGER 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

Transact-SQL 批处理概述

在 Transact-SQL 中，批处理是一起提交并作为一个组依次执行的 SQL 语句组。批处理可以存储在命令文件中。SQL Anywhere 中的 Interactive SQL 和 Adaptive Server Enterprise 中的 Interactive SQL 实用程序为交互地执行批处理提供了类似的功能。

过程中使用的控制语句也可以在批处理中使用。SQL Anywhere 支持在批处理中使用控制语句，并支持以类似于 Transact-SQL 方式使用以 go 语句终止的非分隔语句组来表示批处理的结尾。

对于命令文件中存储的批处理，SQL Anywhere 支持在命令文件中使用参数。请参见“PARAMETERS 语句 [Interactive SQL]”一节 《SQL Anywhere 服务器 - SQL 参考》。

存储过程的自动转换

除了支持 Transact-SQL 替代语法之外，SQL Anywhere 还提供了用于在 Watcom-SQL 方言和 Transact-SQL 方言之间进行转换的辅助工具。返回有关 SQL 语句的信息并可实现 SQL 语句自动转换的函数包括：

- **SQLDialect(statement)** 返回 Watcom-SQL 或 Transact-SQL。
- **WatcomSQL(statement)** 返回语句的 Watcom-SQL 语法。
- **TransactSQL(statement)** 返回语句的 Transact-SQL 语法。

它们是函数，因此可以使用 Interactive SQL 中的 SELECT 语句来访问它们。例如，以下语句返回值 Watcom-SQL：

```
SELECT SQLDialect( 'SELECT * FROM Employees' );
```

使用 Sybase Central 转换存储过程

Sybase Central 具有创建、查看和变更过程和触发器的功能。

♦ 使用 Sybase Central 转换存储过程

1. 使用 Sybase Central 作为要更改的过程的所有者（或者作为 DBA 用户）连接到数据库。
2. 打开 [**过程和函数**] 文件夹。
3. 单击右窗格中的 [**SQL**] 选项卡，然后在编辑器中单击。
4. 根据要使用的方言，从 [**文件**] 菜单中选择 [**转换为**] 命令之一。

   该过程即会以选定方言出现在右窗格中。如果选定方言不是用来存储该过程的方言，服务器就会将其转换为该方言。任何未经转换的行都会显示为注释。
5. 根据需要重写任何未经转换的行。
6. 完成后，选择 [**文件**] » [**保存**]，将转换后的版本保存到数据库中。您也可以将文本导出至文件，以便在 Sybase Central 之外进行编辑。

从 Transact-SQL 过程中返回结果集

SQL Anywhere 使用 RESULT 子句来指定所返回的结果集。在 Transact-SQL 过程中，第一个查询的列名或别名将返回到调用环境。

Transact-SQL 过程的示例

以下 Transact-SQL 过程说明了 Transact-SQL 存储过程如何返回结果集：

```
CREATE PROCEDURE ShowDepartment (@deptname varchar(30))
AS
   SELECT Employees.Surname, Employees.GivenName
   FROM Departments, Employees
   WHERE Departments.DepartmentName = @deptname
   AND Departments.DepartmentID = Employees.DepartmentID;
```

Watcom-SQL 过程的示例

以下是相应的 SQL Anywhere 过程：

```
CREATE PROCEDURE ShowDepartment(in deptname varchar(30))
RESULT ( LastName char(20), FirstName char(20))
BEGIN
   SELECT Employees.Surname, Employees.GivenName
   FROM Departments, Employees
   WHERE Departments.DepartmentName = deptname
   AND Departments.DepartmentID = Employees.DepartmentID
END;
```

有关过程和结果的详细信息，请参见“从过程返回结果”一节第 806 页。

Transact-SQL 过程中的变量

SQL Anywhere 使用 SET 语句来向过程中的变量赋值。在 Transact-SQL 中，将使用含空表列表的 SELECT 语句或 SET 语句来赋值。下面的简单过程说明了 Transact-SQL 语法的工作原理：

```
CREATE PROCEDURE multiply
                @mult1 int,
                @mult2 int,
                @result int output
AS
SELECT @result = @mult1 * @mult2;
```

可以按如下方式来调用该过程：

```
CREATE VARIABLE @product int
go
EXECUTE multiply 5, 6, @product OUTPUT
go
```

当该过程执行后，变量 @product 的值为 30。

有关使用 SELECT 语句给变量赋值的详细信息，请参见“编写兼容的查询”一节第 627 页。有关使用 SET 语句给变量赋值的详细信息，请参见“SET 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

Transact-SQL 过程中的错误处理

Watcom-SQL 方言中的缺省过程错误处理方式与 Transact-SQL 方言中的处理方式不同。缺省情况下，Watcom-SQL 方言过程在它们遇到错误时退出，并且将 SQLSTATE 值和 SQLCODE 值返回到发起调用的环境中。

显式的错误处理可以使用 EXCEPTION 语句来置入到 Watcom-SQL 存储过程中，或者，您可以使用 ON EXCEPTION RESUME 语句指示该过程在遇到错误时从下一个语句继续执行。

当 Transact-SQL 方言过程遇到错误时，将从下一个语句继续执行。全局变量 @@error 保存最近执行过的语句的错误状态。您可以在语句之后检查这一变量，以强制从过程中返回。例如，以下语句在出错时将导致退出。

```
IF @@error != 0 RETURN
```

当该过程执行完之后，返回值将表明该过程是成功还是失败。此返回状态是一个整数，并且可以按如下方式对其进行访问：

```
DECLARE @Status INT
EXECUTE @Status = proc_sample
IF @Status = 0
   PRINT 'procedure succeeded'
ELSE
   PRINT 'procedure failed'
```

下表介绍了内置的过程返回值以及这些值的含义：

值	定义	SQL Anywhere SQLSTATE
0	过程执行时没有发生错误	
-1	缺少对象	42W33、52W02、52003、52W07、42W05
-2	数据类型错误	53018
-3	进程被选作死锁牺牲品	40001、40W06
-4	权限错误	42501
-5	语法错误	42W04
-6	其它用户错误	
-7	资源错误，如空间不足	08W26
-10	致命的内部不一致	40W01
-11	致命的内部不一致	40000
-13	数据库损坏	WI004

值	定义	SQL Anywhere SQLSTATE
-14	硬件错误	08W17、40W03、40W04

当 SQL Anywhere SQLSTATE 不适用时，将返回缺省值 -6。

RETURN 语句可用于返回其它整数，由用户定义它们的含义。

在过程中使用 RAISERROR 语句

可以使用 RAISERROR 语句生成用户定义的错误。RAISERROR 语句的功能类似于 SIGNAL 语句。请参见“RAISERROR 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

RAISERROR 语句本身并不会导致从过程中退出，但它可以同 RETURN 语句或 @@error 全局变量的测试组合在一起控制用户定义错误之后的执行过程。

如果 on_tsql_error 数据库选项被设置为 Continue，RAISERROR 语句将不再发出有关执行结束错误的信号。相反，该过程会完成并存储 RAISERROR 状态代码和消息，然后返回最近的 RAISERROR。如果导致 RAISERROR 的过程是从其它过程中调用的，RAISERROR 将在最外部的调用过程终止后返回。如果将 on_tsql_error 选项设置为缺省设置 (Conditional)，continue_after_raiserror 选项会控制执行 RAISERROR 语句后的行为。如果将 on_tsql_error 选项设置为 Stop 或 Continue，on_tsql_error 设置会优先于 continue_after_raiserror 设置。

当该过程终止后，您将丢失中间的 RAISERROR 状态和代码。如果错误在返回时随 RAISERROR 一起发生，则将返回错误信息并丢失 RAISERROR 信息。应用程序可以通过在不同的执行点检查 @@error 全局变量来查询中间 RAISERROR 状态。

Watcom-SQL 方言中与 Transact-SQL 类似的错误处理

您可以通过向 CREATE PROCEDURE 语句提供 ON EXCEPTION RESUME 子句来使 Watcom-SQL 方言过程以类似于 Transact-SQL 的方式处理错误：

```
CREATE PROCEDURE sample_proc()
ON EXCEPTION RESUME
BEGIN
   ...
END
```

ON EXCEPTION RESUME 子句的存在将防止执行显式的异常处理代码，因此应该避免同时使用这两个子句。

数据库中的 XML

本节介绍如何在数据库中使用 XML。

在数据库中使用 XML

目录

可扩展标记语言（Extensible Markup Language，简称 XML）以文本格式呈现结构化数据。XML 是专门为解决大规模电子发布的难题而设计的。

XML 是一种简单的标记语言，就像 HTML 一样，但它也很灵活，就像 SGML。XML 是分层的，其主要用途是描述数据结构，以便人类和计算机软件都能进行编写和读取。

XML 并不提供用于描述不同形式数据的一组静态元素，而是让您来定义元素。这样，使用 XML 就可以描述很多种结构化数据。XML 文档可以选择使用文档类型定义（Document Type Definition，简称 DTD）或 XML 模式来定义 XML 文件中使用的结构、元素和属性。

可以采用多种方法来搭配使用 XML 和 SQL Anywhere：

- 在数据库中存储 XML 文档
- 以 XML 形式导出关系数据
- 将 XML 导入数据库
- 以 XML 的形式查询关系数据

有关 XML 的更多详细信息，请访问 http://www.w3.org/XML/。

在关系数据库中存储 XML 文档

SQL Anywhere 支持以下两种可用于在您的数据库中存储 XML 文档的数据类型：XML 数据类型和 LONG VARCHAR 数据类型。这两种数据类型都会将 XML 文档以字符串形式存储在数据库中。

XML 数据类型使用数据库服务器的字符集编码。XML 编码属性应与数据库服务器所使用的编码相匹配。XML 编码属性不会指定如何完成自动字符集转换。

您可以在 XML 数据类型与任何其它能够与字符串相互转换的数据类型之间进行转换。请注意，在将字符串转换为 XML 时，不会检查其格式是否正确。

从关系数据中生成元素时，所有在 XML 中无效的字符都会被转义，除非该数据是 XML 类型的。例如，假定您想要使用以下内容生成一个 <product> 元素，使元素内容中包含小于号和大于号：

```
<hat>bowler</hat>
```

如果您编写一个指定元素内容属于 XML 类型的查询，则不会引用大于号和小于号，如下所示：

```
SELECT XMLFOREST( CAST( '<hat>bowler</hat>' AS XML )
AS product );
```

您将得到以下结果：

```
<product><hat>bowler</hat></product>
```

但是，如果该查询没有指定元素内容属于 XML 类型，例如：

```
SELECT XMLFOREST( '<hat>bowler</hat>' AS product );
```

这种情况下，小于号和大于号就会由实体引用所替换，如下所示：

```
<product>&lt;hat&gt;bowler&lt;/hat&gt;</product>
```

请注意，无论选择了何种数据类型，属性始终会进行引用。

有关如何将元素内容转义的详细信息，请参见“对非法的 XML 名称编码”一节第 651 页。

有关 XML 数据类型的详细信息，请参见“XML 数据类型”一节 《SQL Anywhere 服务器 - SQL 参考》。

以 XML 形式导出关系数据

SQL Anywhere 提供了以下两种用于以 XML 形式导出关系数据的方法：Interactive SQL OUTPUT 语句和 ADO.NET DataSet 对象。

利用 FOR XML 子句和 SQL/XML 函数，能够以 XML 形式从数据库的关系数据中生成结果集。然后，您可以使用 UNLOAD 语句或 xp_write_file 系统过程将生成的 XML 导出到一个文件中。

以 XML 形式从 Interactive SQL 中导出关系数据

Interactive SQL OUTPUT 语句支持一种将查询结果输出到生成的 XML 文件的 XML 格式。

此生成的 XML 文件以 UTF-8 格式编码，其中包含一个嵌入式 DTD。在 XML 文件中，二进制值在字符数据 (CDATA) 块中进行编码，块中的二进制数据呈现为两位十六进制数的字符串。

有关使用 OUTPUT 语句导出 XML 的详细信息，请参见“OUTPUT 语句 [Interactive SQL]”一节《SQL Anywhere 服务器 - SQL 参考》。

INPUT 语句不接受 XML 作为文件格式。然而，您可以使用 openxml 过程或 ADO.NET DataSet 对象导入 XML。

有关导入 XML 的详细信息，请参见“导入 XML 文档作为关系数据”一节第 642 页。

使用 DataSet 对象以 XML 形式导出关系数据

使用 ADO.NET DataSet 对象，您可以将 DataSet 的内容保存在 XML 文档中。您在填充了 DataSet（例如，用对您的数据进行查询的结果）后，就可以在 XML 文件中保存 DataSet 中的模式或既保存其模式又保存其数据。WriteXml 方法在 XML 文件中既保存模式又保存数据，而 WriteXmlSchema 方法在 XML 文件中只保存模式。您可以使用 SQL Anywhere ADO.NET 数据提供程序来填充 DataSet 对象。

有关使用 DataSet 以 XML 形式导出关系数据的信息，请参见“使用 SACommand 对象插入、更新和删除行”一节 《SQL Anywhere 服务器 - 编程》。

导入 XML 文档作为关系数据

SQL Anywhere 支持以下两种用于将 XML 导入数据库中的不同方法：

- 使用 openxml 过程从 XML 文档中生成结果集
- 使用 ADO.NET DataSet 对象将 XML 文档中的数据和/或模式读入 DataSet

使用 openxml 导入 XML

在查询的 FROM 子句中使用 openxml 过程，以便从 XML 文档中生成结果集。openxml 使用 XPath 查询语言的子集从 XML 文档中选择节点。

使用 XPath 表达式

在您使用 openxml 时，会对 XML 文档进行分析，而结果会采用树形式建模。这个树由节点组成。XPath 表达式用于选择树中的节点。以下列表介绍了一些常用的 XPath 表达式：

- **/** 指明 XML 文档的根节点
- **//** 指明根的所有下级节点，包括根节点
- **.（一个句点）** 指明 XML 文档的当前节点
- **.//** 指明当前节点的所有子代，包括当前节点
- **..** 指明当前节点的父节点
- ***./@attributename*** 指明名称为 *attributename* 的当前节点的属性
- ***./childname*** 指明当前节点中名称为 *childname* 的子项

请看下面的 XML 文档：

```
<inventory>
  <product ID="301" size="Medium">Tee Shirt
    <quantity>54</quantity>
  </product>
  <product ID="302" size="One Size fits all">Tee Shirt
    <quantity>75</quantity>
  </product>
  <product ID="400" size="One Size fits all">Baseball Cap
    <quantity>112</quantity>
  </product>
</inventory>
```

<inventory> 元素是根节点。可使用以下 XPath 表达式来引用它：

```
/inventory
```

假定当前节点是 <quantity> 元素。可使用以下 XPath 表达式来引用此节点：

```
.
```

要查找作为 <inventory> 元素子项的所有 <product> 元素，请使用以下 XPath 表达式：


```
/inventory/product
```

如果当前节点是 <product> 元素，且您需要引用 size 属性，请使用以下 XPath 表达式：

```
./@size
```

有关 openxml 支持的 XPath 语法的完整列表，请参见“openxml 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》。

有关 XPath 查询语言的信息，请参见 http://www.w3.org/TR/xpath。

使用 openxml 生成结果集

openxml 的第一个 *xpath-query* 参数每匹配一次，就会在结果集中生成一行。WITH 子句指定结果集的模式以及如何为结果集中的每一列找到值。例如，假定有以下查询：

```
SELECT * FROM openxml( '<inventory>
                          <product>Tee Shirt
                            <quantity>54</quantity>
                            <color>Orange</color>
                          </product>
                          <product>Baseball Cap
                    <quantity>112</quantity>
                            <color>Black</color>
                          </product>
                        </inventory>',
                       '/inventory/product' )
WITH ( Name CHAR (25) './text()',
       Quantity CHAR(3) 'quantity',
       Color CHAR(20) 'color');
```

第一个 *xpath-query* 参数是 /inventory/product，并且 XML 中有两个 <product> 元素，因此，此查询会生成两行。

WITH 子句指定其中包含三列：Name、Quantity 和 Color。这三列的值来自于 <product>、<quantity> 和 <color> 元素。以上查询会生成以下结果：

Name	Quantity	Color
Tee Shirt	54	Orange
Baseball Cap	112	Black

有关详细信息，请参见“openxml 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》。

使用 openxml 生成边缘表

openxml 过程可以用来生成边缘表，在这种表中，XML 文档中的每一个元素都会有对应的一行。您可能会希望生成一个边缘表，以便能够使用 SQL 来查询结果集中的数据。

以下 SQL 语句创建了一个变量 x，其中包含一个 XML 文档。该查询生成的 XML 包含一个名为 <root> 的根元素，它是使用 XMLELEMENT 函数生成的，另外还使用指定了 ELEMENTS 修饰符的 FOR XML AUTO 为 Employees、SalesOrders 和 Customers 表中的每一列生成元素。

有关 XMLELEMENT 函数的信息，请参见“XMLELEMENT 函数 [String]”一节 《SQL Anywhere 服务器 - SQL 参考》。

有关 FOR XML AUTO 的信息，请参见“使用 FOR XML AUTO”一节第 654 页。

```
CREATE VARIABLE x XML;
SET x=(SELECT XMLELEMENT( NAME root,
          (SELECT * FROM Employees
          KEY JOIN SalesOrders
          KEY JOIN Customers
          FOR XML AUTO, ELEMENTS)));
SELECT x;
```

生成的 XML 将如下所示（已对结果进行了格式化处理以便于阅读—由查询返回的结果是一个连续字符串）：

```
<root>
 <Employees>
  <EmployeeID>299</EmployeeID>
  <ManagerID>902</ManagerID>
  <Surname>Overbey</Surname>
  <GivenName>Rollin</GivenName>
  <DepartmentID>200</DepartmentID>
  <Street>191 Companion Ct.</Street>
  <City>Kanata</City>
  <State>CA</State>
  <Country>USA</Country>
  <PostalCode>94608</PostalCode>
  <Phone>5105557255</Phone>
  <Status>A</Status>
  <SocialSecurityNumber>025487133</SocialSecurityNumber>
  <Salary>39300.000</Salary>
  <StartDate>1987-02-19</StartDate>
  <BirthDate>1964-03-15</BirthDate>
  <BenefitHealthInsurance>Y</BenefitHealthInsurance>
  <BenefitLifeInsurance>Y</BenefitLifeInsurance>
  <BenefitDayCare>N</BenefitDayCare>
  <Sex>M</Sex>
  <SalesOrders>
  <ID>2001</ID>
  <CustomerID>101</CustomerID>
  <OrderDate>2000-03-16</OrderDate>
  <FinancialCode>r1</FinancialCode>
  <Region>Eastern</Region>
  <SalesRepresentative>299</SalesRepresentative>
    <Customers>
    <ID>101</ID>
    <Surname>Devlin</Surname>
    <GivenName>Michael</GivenName>
    <Street>114 Pioneer Avenue</Street>
    <City>Kingston</City>
    <State>NJ</State>
    <PostalCode>07070</PostalCode>
    <Phone>2015558966</Phone>
    <CompanyName>The Power Group</CompanyName>
    </Customers>
  </SalesOrders>
</Employees>
...
```

以下查询使用下级或自身 (//*) XPath 表达式来匹配以上 XML 文档中的每一个元素，对于每一个元素，使用 id 元属性来获取节点的 ID，将父 (../) XPath 表达式与 ID 元属性搭配使用来获取父节点。localname 元属性用于获取每个元素的名称。元属性名称是区分大小写的，因此不能将 ID 或 LOCALNAME 用作元属性名称。


```
SELECT * FROM openxml( x, '//*' )
 WITH (ID INT '@mp:id',
       parent INT '../@mp:id',
       name CHAR(25) '@mp:localname',
       text LONG VARCHAR 'text()' )
ORDER BY ID;
```

此查询生成的结果集会显示 XML 文档中每个节点的 ID、父节点的 ID 以及各元素的名称和内容。

ID	parent	name	text
5	(NULL)	root	(NULL)
16	5	Employees	(NULL)
28	16	EmployeeID	299
55	16	ManagerID	902
79	16	Surname	Overbey
...	...	...	...

搭配使用 openxml 和 xp_read_file

到目前为止，已对通过诸如 XMLELEMENT 之类的过程生成的 XML 进行了使用。您还可以从文件中读取 XML 并使用 xp_read_file 过程对其进行分析。假定文件 *c:\inventory.xml* 中包含以下内容：

```
<inventory>
   <product>Tee Shirt
      <quantity>54</quantity>
      <color>Orange</color>
   </product>
   <product>Baseball Cap
      <quantity>112</quantity>
      <color>Black</color>
   </product>
</inventory>
```

您可以使用以下语句读取并分析文件中的 XML：

```
CREATE VARIABLE x XML;
SELECT xp_read_file( 'c:\\inventory.xml' )
 INTO x;
SELECT * FROM openxml( x, '//*' )
 WITH (ID INT '@mp:id',
       parent INT '../@mp:id',
       name CHAR(128) '@mp:localname',
       text LONG VARCHAR 'text()' )
ORDER BY ID;
```

查询列中的 XML

如果您有一个表，它的某一列包含 XML，您可以使用 openxml 一次查询出该列中的所有 XML 值。使用横向派生表可以实现这一点。

以下语句将创建一个含有两列（ManagerID 和 Reports）的表。Reports 列包含从 Employees 表中生成的 XML 数据。

```
CREATE TABLE test (ManagerID INT, Reports XML);
INSERT INTO test
SELECT ManagerID, XMLELEMENT( NAME reports,
            XMLAGG( XMLELEMENT( NAME e, EmployeeID)))
FROM Employees
GROUP BY ManagerID;
```

执行以下查询可查看 test 表中的数据：

```
SELECT * FROM test
ORDER BY ManagerID;
```

此查询会生成以下结果：

ManagerID	Reports
501	<reports> <e>102</e> <e>105</e> <e>160</e> <e>243</e> ... </reports>
703	<reports> <e>191</e> <e>750</e> <e>868</e> <e>921</e> ... </reports>
902	<reports> <e>129</e> <e>195</e> <e>299</e> <e>467</e> ... </reports>
1293	<reports> <e>148</e> <e>390</e> <e>586</e> <e>757</e> ... </reports>
...	...

以下查询使用横向派生表来生成一个含有两列的结果集：一列中列出了每位经理的 ID，另一列中列出了会向该经理报告的每位雇员的 ID：

```
SELECT ManagerID, EmployeeID
FROM test, LATERAL( openxml( test.Reports, '//e' )
WITH (EmployeeID INT '.') ) DerivedTable
ORDER BY ManagerID, EmployeeID;
```


此查询会生成以下结果：

ManagerID	EmployeeID
501	102
501	105
501	160
501	243
...	...

有关横向派生表的详细信息，请参见“FROM 子句”一节 《SQL Anywhere 服务器 - SQL 参考》。

使用 DataSet 对象导入 XML

使用 ADO.NET DataSet 对象，您可以将 XML 文档中的数据和/或模式读入 DataSet。

- ReadXml 方法用既包含模式又包含数据的 XML 文档中的内容填充 DataSet。
- ReadXmlSchema 方法只从 XML 文档中读取模式。在用 XML 文档中的数据填充了 DataSet 之后，您可以用 DataSet 中的更改来更新数据库中的表。

使用 SQL Anywhere ADO.NET 数据提供程序也可操作 DataSet 对象。

有关利用 SQL Anywhere .NET 数据提供程序通过 DataSet 从 XML 文档中读取数据和/或模式的信息，请参见“使用 SADataAdapter 对象获取数据”一节 《SQL Anywhere 服务器 - 编程》。

定义缺省 XML 命名空间

在 XML 文档具有 `xmlns="URI"` 形式属性的元素中定义缺省命名空间。在以下示例中，文档的缺省命名空间绑定到 URI http://www.iAnywhere.com/EmployeeDemo：

```
<x xmlns="http://www.iAnywhere.com/EmployeeDemo"/>
```

如果元素的名称没有前缀，则该元素及定义该元素的位置的所有下级都将应用缺省命名空间。用冒号将前缀与元素名称的其余部分隔开。例如，<x/> 没有前缀，而 <p:x/> 具有前缀 p。定义的命名空间将与具有 `xmlns:prefix="URI"` 形式属性的前缀绑定。在以下示例中，文档将前缀 p 绑定到与上一个实例相同的 URI：

```
<x xmlns:p="http://www.iAnywhere.com/EmployeeDemo"/>
```

缺省命名空间不会应用于属性。除非它具有前缀，否则属性会始终绑定到 NULL 命名空间 URI。在以下示例中，根元素和子元素具有 iAnywhere1 命名空间，而 x 属性具有 NULL 命名空间 URI，y 元素具有 iAnywhere2 命名空间：

```
<root xmlns="iAnywhere1" xmlns:p="iAnywhere2">
<child x='1' p:y='2' />
</root>
```

将 XML 文档作为 openxml 查询的 *namespace-declaration* 参数传递时，在查询中将应用文档根元素中定义的命名空间。文档在根元素之后的所有部分都将被忽略。在以下示例中，p1 绑定到文档中的 iAnywhere1 以及 *namespace-declaration* 参数中的 p2，且查询可以使用前缀 p2：

```
SELECT *
FROM openxml( '<p1:x xmlns:p1="iAnywhere1"> 1 </x>', '/p2:x', 1, '<root
xmlns:p2="iAnywhere1"/>' )
WITH ( c1 int '.' );
```

当匹配某个元素时，必须正确指定前缀绑定到的 URI。在上面的示例中，xpath 查询中的 x 名称与文档中的 x 元素相匹配，因为它们都具有 iAnywhere1 命名空间。

不要在 openxml 系统过程的 *namespace-declaration* 中使用缺省命名空间。请使用 /*:x 形式的通配符查询（与绑定到包括 NULL 命名空间的任意 URI 的 x 元素相匹配），或者将所需的 URI 与特定前缀绑定并在查询中使用。有关从 XML 文档生成结果集的详细信息，请参见“openxml 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》。

以 XML 格式获取查询结果

SQL Anywhere 支持使用以下两种不同的方法来以 XML 形式从您的关系数据中获取查询结果：

- **FOR XML 子句** FOR XML 子句可以用在 SELECT 语句中，以生成 XML 文档。

  有关使用 FOR XML 子句的信息，请参见“使用 FOR XML 子句以 XML 格式检索查询结果”一节第 649 页和“SELECT 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

- **SQL/XML** SQL Anywhere 支持基于从关系数据中生成 XML 文档的 SQL/XML 标准草案的函数。

  有关在查询中使用一个或多个此类函数的信息，请参见“使用 SQL/XML 以 XML 形式获取查询结果”一节第 666 页。

SQL Anywhere 支持的 FOR XML 子句和 SQL/XML 函数为您在从关系数据中生成 XML 时提供了两种选择。在大多数情况下，这两种方法生成的 XML 是相同的，您可以使用任意一种。

例如，此查询使用 FOR XML AUTO 生成 XML：

```
SELECT ID, Name
FROM Products
WHERE Color='black'
FOR XML AUTO;
```

以下查询使用 XMLELEMENT 函数生成 XML：

```
SELECT XMLELEMENT(NAME product,
          XMLATTRIBUTES(ID, Name))
FROM Products
WHERE Color='black';
```

这两种查询都会生成下面的 XML（已对结果集进行了格式化处理以便于阅读）：

```
<product ID="302" Name="Tee Shirt"/>
<product ID="400" Name="Baseball Cap"/>
<product ID="501" Name="Visor"/>
<product ID="700" Name="Shorts"/>
```

提示

如果您要生成深度嵌套的文档，FOR XML EXPLICIT 查询可能会比 SQL/XML 查询效率高，因为 EXPLICIT 模式查询通常会使用 UNION 来生成嵌套，而 SQL/XML 使用子查询来生成所需嵌套。

使用 FOR XML 子句以 XML 格式检索查询结果

使用 SQL Anywhere 时，通过在您的 SELECT 语句中使用 FOR XML 子句，您可以针对数据库执行 SQL 查询并以 XML 文档的形式返回结果。XML 文档属于 XML 类型。

有关 XML 数据类型的信息，请参见“XML 数据类型”一节 《SQL Anywhere 服务器 - SQL 参考》。

FOR XML 子句可以用在任何 SELECT 语句中，包括子查询、使用 GROUP BY 子句或集合函数的查询，以及视图定义。

有关如何使用 FOR XML 子句的示例，请参见“FOR XML 示例”一节第 652 页。

SQL Anywhere 并不为由 FOR XML 子句所生成的 XML 文档生成模式。

在 FOR XML 子句中，您会指定控制所生成的 XML 的格式的三种 XML 模式中的一种：

- **RAW** 表示作为 XML <row> 元素与查询匹配的每一行以及作为属性的每一列。

  有关详细信息，请参见“使用 FOR XML RAW”一节第 652 页。

- **AUTO** 以嵌套的 XML 元素的形式返回查询结果。*选择列表*中引用的每一个表都会表示为 XML 中的一个元素。元素的嵌套顺序视 *select-list* 中表的顺序而定。

  有关详细信息，请参见“使用 FOR XML AUTO”一节第 654 页。

- **EXPLICIT** 允许您编写包含有关正确嵌套的信息的查询，这样您就能够控制最终 XML 的形式。

  有关详细信息，请参见“使用 FOR XML EXPLICIT”一节第 657 页。

下面的部分介绍了 FOR XML 子句的所有三种模式：二进制数据、NULL 值和无效的 XML 名称等方面的行为。该部分还包括 FOR XML 子句的使用方法示例。

FOR XML 和二进制数据

您在 SELECT 语句中使用 FOR XML 子句时，无论使用哪种模式，BINARY、LONG BINARY、IMAGE 或 VARBINARY 列都会输出为自动以 base64 编码格式表示的属性或元素。

如果您要使用 openxml 从 XML 生成结果集，openxml 会假定类型 BINARY、LONG BINARY、IMAGE 和 VARBINARY 以 base64 格式编码并自动对它们进行解码。

有关 openxml 的详细信息，请参见“openxml 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》。

FOR XML 和 NULL 值

缺省情况下，结果中会忽略包含 NULL 值的元素和属性。此行为是由 for_xml_null_treatment 选项控制的。

假定 Customers 表中存在一个包含 NULL 公司名的条目。

```
INSERT INTO
      Customers( ID, Surname, GivenName, Street, City, Phone)
VALUES (100,'Robert','Michael',
       '100 Anywhere Lane','Smallville','519-555-3344');
```

如果您将 for_xml_null_treatment 选项设置为 Omit（缺省值）后执行以下查询，则不会为 NULL 列值生成属性。

```
SELECT ID, GivenName, Surname, CompanyName
FROM Customers
WHERE GivenName LIKE 'Michael%'
ORDER BY ID
FOR XML RAW;
```


在本例中，不会为 Michael Robert 生成 CompanyName 属性。

```
<row ID="100" GivenName="Michael" Surname="Robert"/>
<row ID="101" GivenName="Michaels" Surname="Devlin" CompanyName="The Power
Group"/>
<row ID="110" GivenName="Michael" Surname="Agliori" CompanyName="The Pep
Squad"/>
```

如果 for_xml_null_treatment 选项设置为 Empty，那么结果中就会包括一个空属性：

```
<row ID="100" GivenName="Michael" Surname="Robert" CompanyName=""/>
<row ID="101" GivenName="Michaels" Surname="Devlin" CompanyName="The Power
Group"/>
<row ID="110" GivenName="Michael" Surname="Agliori" CompanyName="The Pep
Squad"/>
```

在本例中，将会为 Michael Robert 生成一个空 CompanyName 属性。

有关 for_xml_null_treatment 选项的信息，请参见“for_xml_null_treatment 选项 [数据库]”一节《SQL Anywhere 服务器 - 数据库管理》。

对非法的 XML 名称编码

SQL Anywhere 在对不是合法 XML 名称的名称（例如，包括空格的列名）进行编码时会使用以下规则：

XML 的名称规则与 SQL 名称规则不同。例如，XML 名称中不允许使用空格。在将 SQL 名称（如列名）转换为 XML 名称时，对于 XML 名称无效的字符将被编码或转义。

对于每个被编码的字符，编码是基于字符的 Unicode 代码点值（以十六进制形式表示）进行的。

- 对于大多数字符，可将代码点值表示为 16 位或 4 个十六进制数字（使用编码 _xHHHH_）。这些字符与 UTF-16 值为一个 16 位字的 Unicode 字符相对应。
- 对于代码点值所需位数超过 16 位的字符，将在编码 _xHHHHHHHH_ 中使用 8 个十六进制数字。这些字符与 UTF-16 值为两个 16 位字的 Unicode 字符相对应。但 Unicode 代码点值（通常为 5 或 6 个十六进制数字）用于编码，而非 UTF-16 值。

  例如，以下查询包含带有空格的列名：

  ```
  SELECT EmployeeID AS "Employee ID"
  FROM Employees
  FOR XML RAW;
  ```

  返回以下结果：

  ```
  <row Employee_x0020_ID="102"/>
  <row Employee_x0020_ID="105"/>
  <row Employee_x0020_ID="129"/>
  <row Employee_x0020_ID="148"/>
  ...
  ```

- 下划线 (_) 后如果跟有字符 x，就会被转义。例如，名称 Linu_x 将被编码为 Linu_x005F_x。
- 冒号 (:) 不会被转义，这样，使用 FOR XML 查询就可以生成命名空间声明和限定的元素名和属性名。

有关 FOR XML 子句的语法信息，请参见“SELECT 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

提示
在 Interactive SQL 中执行包含 FOR XML 子句的查询时，您可能会需要通过设置 truncation_length 选项来增加列的长度。

有关设置截断长度的信息，请参见“truncation_length 选项 [Interactive SQL]”一节 《SQL Anywhere 服务器 - 数据库管理》。

FOR XML 示例

以下示例说明了如何在 SELECT 语句中使用 FOR XML 子句。

- 以下示例说明了如何在子查询中使用 FOR XML 子句：

```
SELECT XMLELEMENT(
   NAME root,
      (SELECT * FROM Employees
       FOR XML RAW));
```

- 以下示例说明了如何在使用了 GROUP BY 子句和集合函数的查询中使用 FOR XML 子句：

```
SELECT Name, AVG(UnitPrice) AS Price
FROM Products
GROUP BY Name
FOR XML RAW;
```

- 以下示例说明了如何在视图定义中使用 FOR XML 子句：

```
CREATE VIEW EmployeesDepartments
AS SELECT Surname, GivenName, DepartmentName
FROM Employees JOIN Departments
ON Employees.DepartmentID = Departments.DepartmentID
FOR XML AUTO;
```

使用 FOR XML RAW

当您在查询中指定 FOR XML RAW 时，每一行都会表示为一个 <row> 元素，而每一列都是 <row> 元素的一个属性。

语法

FOR XML RAW[, **ELEMENTS**]

参数

ELEMENTS 让 FOR XML RAW 为结果中的每一列都生成一个 XML 元素，而不是生成一个属性。如果有 NULL 值，生成的 XML 文档中就会忽略该元素。以下查询会生成 <EmployeeID> 元素和 <DepartmentName> 元素：

```
SELECT Employees.EmployeeID, Departments.DepartmentName
FROM Employees JOIN Departments
```



```
   ON Employees.DepartmentID=Departments.DepartmentID
FOR XML RAW, ELEMENTS;
```

此查询会得到以下结果：

```
<row>
   <EmployeeID>102</EmployeeID>
   <DepartmentName>R &amp;amp; D</DepartmentName>
</row>
<row>
   <EmployeeID>105</EmployeeID>
   <DepartmentName>R &amp;amp; D</DepartmentName>
</row>
<row>
   <EmployeeID>160</EmployeeID>
   <DepartmentName>R &amp;amp; D</DepartmentName>
</row>
<row>
   <EmployeeID>243</EmployeeID>
   <DepartmentName>R &amp;amp; D</DepartmentName>
</row>
...
```

用法

在您执行包含 FOR XML RAW 的查询时，BINARY、LONG BINARY、IMAGE 和 VARBINARY 列中的数据会自动以 base64 编码格式返回。

缺省情况下，结果中会忽略 NULL 值。此行为是由 for_xml_null_treatment 选项控制的。

有关如何在包含 FOR XML 子句的查询中返回 NULL 值的信息，请参见“FOR XML 和 NULL 值”一节第 650 页。

FOR XML RAW 不会返回结构完好的 XML 文档，因为该文档没有单一的根节点。如果需要 <root> 元素，可以使用 XMLELEMENT 函数插入一个。例如，

```
SELECT XMLELEMENT( NAME root,
                   (SELECT EmployeeID AS id, GivenName AS name
                   FROM Employees FOR XML RAW));
```

有关 XMLELEMENT 函数的信息，请参见“XMLELEMENT 函数 [String]”一节 《SQL Anywhere 服务器 - SQL 参考》。

通过指定别名，可以更改 XML 文档中使用的属性名或元素名。以下查询会将 ID 属性重命名为 product_ID：

```
SELECT ID AS product_ID
FROM Products
WHERE Color='black'
FOR XML RAW;
```

此查询会得到以下结果：

```
<row product_ID="302"/>
<row product_ID="400"/>
<row product_ID="501"/>
<row product_ID="700"/>
```

结果的顺序取决于优化程序所选择的计划，除非您以其它方式请求。如果您希望结果按某种特定顺序显示，必须在查询中加入 ORDER BY 子句，例如：

```
SELECT Employees.EmployeeID, Departments.DepartmentName
FROM Employees JOIN Departments
   ON Employees.DepartmentID=Departments.DepartmentID
ORDER BY EmployeeID
FOR XML RAW;
```

示例

假定像下面这样，您想检索有关某名雇员隶属于哪个部门的信息：

```
SELECT Employees.EmployeeID, Departments.DepartmentName
FROM Employees JOIN Departments
   ON Employees.DepartmentID=Departments.DepartmentID
FOR XML RAW;
```

返回以下 XML 文档：

```
<row EmployeeID="102" DepartmentName="R &amp; D"/>
<row EmployeeID="105" DepartmentName="R &amp; D"/>
<row EmployeeID="160" DepartmentName="R &amp; D"/>
<row EmployeeID="243" DepartmentName="R &amp; D"/>
...
```

使用 FOR XML AUTO

AUTO 模式会在 XML 文档中生成嵌套的元素。选择列表中引用的每一个表在生成的 XML 中都会表示为一个元素。嵌套的顺序基于选择列表中引用表的顺序而定。在您指定 AUTO 模式时，会为选择列表中的每个表创建一个元素，而该表中的每一列都是一个单独的属性。

语法

FOR XML AUTO[, **ELEMENTS**]

参数

ELEMENTS 让 FOR XML AUTO 为结果中的每一列都生成一个 XML 元素，而不是生成一个属性。例如，

```
SELECT Employees.EmployeeID, Departments.DepartmentName
FROM Employees JOIN Departments
   ON Employees.DepartmentID=Departments.DepartmentID
ORDER BY EmployeeID
FOR XML AUTO, ELEMENTS;
```

在本例中，结果集中的每一列都会作为一个单独的元素返回，而不是 <Employees> 元素的一个属性。如果有 NULL 值，生成的 XML 文档中就会忽略该元素。

```
<Employees>
   <EmployeeID>102</EmployeeID>
   <Departments>
      <DepartmentName>R &amp; D</DepartmentName>
   </Departments>
</Employees>
<Employees>
   <EmployeeID>105</EmployeeID>
   <Departments>
      <DepartmentName>R &amp; D</DepartmentName>
   </Departments>
```



```
</Employees>
<Employees>
   <EmployeeID>129</EmployeeID>
   <Departments>
   <DepartmentName>Sales</DepartmentName>
   </Departments>
</Employees>
...
```

用法

在您使用 FOR XML AUTO 执行查询时，BINARY、LONG BINARY、IMAGE 和 VARBINARY 列中的数据会自动以 base64 编码格式返回。缺省情况下，结果中会忽略 NULL 值。通过将 for_xml_null_treatment 选项设置为 EMPTY，您可以将 NULL 值作为空属性返回。

有关设置 for_xml_null_treatment 选项的信息，请参见“for_xml_null_treatment 选项 [数据库]”一节 《SQL Anywhere 服务器 - 数据库管理》。

除非另有要求，否则数据库服务器返回表中行的顺序没有任何意义。如果您希望结果按某种特定顺序显示，或者，对于有多个子项的父元素，您必须在查询中加入 ORDER BY 子句，以让所有子项都相邻。如果您不指定 ORDER BY 子句，结果的嵌套就会取决于优化程序所选择的计划，而且您可能得不到需要的嵌套。

FOR XML AUTO 不会返回结构完好的 XML 文档，因为该文档没有单一的根节点。如果需要 <root> 元素，可以使用 XMLELEMENT 函数插入一个。例如，

```
SELECT XMLELEMENT( NAME root,
                   (SELECT EmployeeID AS id, GivenName AS name
                   FROM Employees FOR XML AUTO ) );
```

有关 XMLELEMENT 函数的信息，请参见“XMLELEMENT 函数 [String]”一节 《SQL Anywhere 服务器 - SQL 参考》。

通过指定别名，您可以更改 XML 文档中使用的属性名或元素名。以下查询会将 ID 属性重命名为 product_ID：

```
SELECT ID AS product_ID
FROM Products
WHERE Color='Black'
FOR XML AUTO;
```

生成的 XML 如下：

```
<Products product_ID="302"/>
<Products product_ID="400"/>
<Products product_ID="501"/>
<Products product_ID="700"/>
```

您也可以用别名来重命名表。以下查询将表重命名为 product_info：

```
SELECT ID AS product_ID
FROM Products AS product_info
WHERE Color='Black'
FOR XML AUTO;
```

生成的 XML 如下：

```
<product_info product_ID="302"/>
<product_info product_ID="400"/>
```

```
<product_info product_ID="501"/>
<product_info product_ID="700"/>
```

示例

以下查询所生成的 XML 同时包含 <employee> 元素和 <department> 元素，其中，<employee> 元素（在选择列表中列在第一位的表）是 <department> 元素的父项。

```
SELECT EmployeeID, DepartmentName
FROM Employees AS employee JOIN Departments AS department
   ON Employees.DepartmentID=Departments.DepartmentID
ORDER BY EmployeeID
FOR XML AUTO;
```

以上查询将生成以下 XML：

```
<employee EmployeeID="102">
   <department DepartmentName="R &amp;amp; D"/>
</employee>
<employee EmployeeID="105">
   <department DepartmentName="R &amp;amp; D"/>
</employee>
<employee EmployeeID="129">
   <department DepartmentName="Sales;"/>
</employee>
<employee EmployeeID="148">
   <department DepartmentName="Finance;"/>
</employee>
...
```

如果您按以下所示更改选择列表中列的顺序：

```
SELECT DepartmentName, EmployeeID
FROM Employees AS employee JOIN Departments AS department
   ON Employees.DepartmentID=Departments.DepartmentID
ORDER BY 1, 2
FOR XML AUTO;
```

结果会如下所示嵌套：

```
<department DepartmentName="Finance">
   <employee EmployeeID="148"/>
   <employee EmployeeID="390"/>
   <employee EmployeeID="586"/>
   ...
</department>
<Department name="Marketing">
   <employee EmployeeID="184"/>
   <employee EmployeeID="207"/>
   <employee EmployeeID="318"/>
   ...
</department>
...
```

为该查询生成的 XML 还是同时包含 <employee> 元素和 <department> 元素，但这一次，<department> 元素是 <employee> 元素的父项。

使用 FOR XML EXPLICIT

使用 FOR XML EXPLICIT，您可以控制查询所返回的 XML 文档的结构。查询必须用特定的方法编写，以在查询结果中指定有关您需要的嵌套的信息。使用 FOR XML EXPLICIT 支持的可选指令，您可以配置各列的处理方式。例如，您可以对以下情况加以控制：让列显示为元素内容还是显示为属性内容，或者，列是否只用于给结果排序，而不会出现在生成的 XML 中。

有关如何使用 FOR XML EXPLICIT 编写查询的示例，请参见“编写 EXPLICIT 模式的查询”一节第 658 页。

参数

在 EXPLICIT 模式中，SELECT 语句中的前两列必须分别命名为 **Tag** 和 **Parent**。Tag 和 Parent 是元数据列，它们的值用来确定由查询返回的 XML 文档中各元素之间的父子关系（或嵌套）。

- **Tag 列** 这是在选择列表中指定的第一列。Tag 列存储当前元素的标记值。标记号可以使用的值是 1 到 255。
- **Parent 列** 此列存储当前元素的父项的标记号。如果此列中的值为 NULL，则行就会被放置在 XML 层次的顶层。

例如，假定有这样一个查询，它在没有指定 FOR XML EXPLICIT 时会返回以下结果集。（将会在下一节“向查询添加数据列”一节第 657 页中介绍 GivenName!1 和 ID!2 数据列的用途。）

Tag	Parent	GivenName!1	ID!2
1	NULL	'Beth'	NULL
2	NULL	NULL	'102'

在本示例中，Tag 列中的值是结果集中各元素的标记号。两行的 Parent 列都包含 NULL 值。这意味着两个元素都会在层次的顶层生成，当查询包括 FOR XML EXPLICIT 子句时，将给出以下结果：

```
<GivenName>Beth</GivenName>
<ID>102</ID>
```

但是，如果第二行的 Parent 列的值是 1，结果将如下所示：

```
<GivenName>Beth
   <ID>102</ID>
</GivenName>
```

有关如何使用 FOR XML EXPLICIT 编写查询的示例，请参见“编写 EXPLICIT 模式的查询”一节第 658 页。

向查询添加数据列

除 Tag 列和 Parent 列之外，查询还必须包含一个或多个数据列。这些数据列的名称控制着在标记过程中解释列的方式。每个列名都拆分为由惊叹号 (!) 分隔的字段。可以为数据列指定以下字段：

ElementName!*TagNumber*!*AttributeName*!*Directive*

ElementName 元素的名称。对于给定的行，会从标记号匹配的第一列的 *ElementName* 字段中提取为该行生成的元素的名称。如果有多个 *TagNumber* 相同的列，对于后面的 *TagNumber* 相同的列，会忽略 *ElementName*。在上面的示例中，第一行会生成一个名为 <GivenName> 的元素。

TagNumber 元素的标记号。对于有给定标记值的行，所有在其 *TagNumber* 字段中具有相同值的列都会向与该行相对应的元素提供内容。

AttributeName 指定列值是 *ElementName* 元素的属性。例如，如果数据列具有名称 productID!1!Color，那么 Color 就会显示为 <productID> 元素的一个属性。

Directive 使用此可选字段，您可以进一步控制 XML 文档的格式。您可以为 *Directive* 指定以下任意一个值：

- **hide** 指明在生成结果时忽略此列。这条指令可以用于加入只用来给表排序的列。属性名会被忽略，且不出现在结果中。

  有关使用 hide 指令的示例，请参见“使用 hide 指令”一节第 662 页。

- **element** 指明列值作为名称为 *AttributeName* 的嵌套元素插入，而不是作为属性。

  有关使用 element 指令的示例，请参见“使用 element 指令”一节第 661 页。

- **xml** 指明列值插入时不进行引用。如果指定了 *AttributeName*，则值作为使用该名称的元素插入。否则，它在插入时没有包装元素。如果不使用此指令，标记字符就会被转义，除非列是 XML 类型。例如，值 <a/> 将以 <a/> 形式插入。

  有关使用 xml 指令的示例，请参见“使用 xml 指令”一节第 663 页。

- **cdata** 指明列值将作为 CDATA 区段插入。*AttributeName* 会被忽略。

  有关使用 cdata 指令的示例，请参见“使用 cdata 指令”一节第 664 页。

用法

在执行包含 FOR XML EXPLICIT 的查询时，BINARY、LONG BINARY、IMAGE 和 VARBINARY 列中的数据会自动以 base64 编码格式返回。缺省情况下，结果集中的任何 NULL 值都会被忽略。通过更改 for_xml_null_treatment 选项的设置，您可以更改此行为。

有关 for_xml_null_treatment 选项的详细信息，请参见“for_xml_null_treatment 选项 [数据库]”一节 《SQL Anywhere 服务器 - 数据库管理》和“FOR XML 和 NULL 值”一节第 650 页。

编写 EXPLICIT 模式的查询

假定您要使用 FOR XML EXPLICIT 编写一个生成以下 XML 文档的查询：

```
<employee EmployeeID='129'>
   <customer CustomerID='107' Region='Eastern'/>
   <customer CustomerID='119' Region='Western'/>
   <customer CustomerID='131' Region='Eastern'/>
</employee>
<employee EmployeeID='195'>
   <customer CustomerID='109' Region='Eastern'/>
   <customer CustomerID='121' Region='Central'/>
</employee>
```

实现方法是：编写一条按指定的准确顺序返回以下结果集的 SELECT 语句，然后将 FOR XML EXPLICIT 附加到该查询。

Tag	Parent	employee!1!EmployeeID	customer!2!CustomerID	customer!2!Region
1	NULL	129	NULL	NULL
2	1	129	107	Eastern
2	1	129	119	Western
2	1	129	131	Central
1	NULL	195	NULL	NULL
2	1	195	109	Eastern
2	1	195	121	Central

在编写查询时，对于某指定行，只有部分列会成为所生成的 XML 文档的组成部分。仅当 *TagNumber* 字段（列名中的第二个字段）中的值与 Tag 列中的值匹配时，XML 文档中才会加入一列。

在该示例中，第三列用于在其 Tag 列中有值 1 的两个行。在第四和第五列中，值用于在其 Tag 列中有值 2 的行。从列名中的第一个字段中提取元素名。在本例中，会创建 <employee> 元素和 <customer> 元素。

属性名来自列名中的第三个字段，所以，EmployeeID 属性是为 <employee> 元素创建的，而 CustomerID 属性和 Region 属性是为 <customer> 元素生成的。

以下步骤介绍了如何使用 SQL Anywhere 示例数据库构建一个 FOR XML EXPLICIT 查询，让该查询生成一个与本节开始时出现的 XML 文档类似的 XML 文档。

♦ 编写 FOR XML EXPLICIT 查询

1. 编写 SELECT 语句以生成顶层元素。

   在本示例中，查询中的第一个 SELECT 语句会生成 <employee> 元素。查询中的前两个值必须是 Tag 和 Parent 列值。<employee> 元素位于层次的顶层，所以会给它分配一个 Tag 值 1 以及一个 Parent 值 NULL。

   注意
   如果您要编写一个使用 UNION 的 EXPLICIT 模式查询，则只有在第一条 SELECT 语句中指定的列名才会被使用。要用作元素名或属性名的列名必须在第一条 SELECT 语句中指定，因为在后续 SELECT 语句中指定的列名会被忽略。

   要为上面的表生成 <employee> 元素，您的第一条 SELECT 语句应如下所示：

```
SELECT
      1          AS tag,
      NULL       AS parent,
      EmployeeID AS [employee!1!EmployeeID],
      NULL       AS [customer!2!CustomerID],
      NULL       AS [customer!2!Region]
FROM Employees;
```

2. 编写 SELECT 语句来生成子元素。

   第二个查询会生成 <customer> 元素。由于这是一个 EXPLICIT 模式查询，所以在所有 SELECT 语句中指定的前两个值都必须是 Tag 值和 Parent 值。<customer> 元素会得到标记号 2，而由于它是 <employee> 元素的子项，因此具有 Parent 值 1。第一条 SELECT 语句已经指出 EmployeeID、CustomerID 和 Region 都是属性。

```
SELECT
      2,
      1,
      EmployeeID,
      CustomerID,
      Region
FROM Employees KEY JOIN SalesOrders
```

3. 给查询添加 UNION ALL 以便将两条 SELECT 语句合并到一起：

```
SELECT
      1          AS tag,
      NULL       AS parent,
      EmployeeID AS [employee!1!EmployeeID],
      NULL       AS [customer!2!CustomerID],
      NULL       AS [customer!2!Region]
FROM Employees
UNION ALL
SELECT
      2,
      1,
      EmployeeID,
      CustomerID,
      Region
FROM Employees KEY JOIN SalesOrders
```

4. 添加 ORDER BY 子句以指定结果中的行的顺序。行的顺序即在最终生成的文档中使用的顺序。

```
SELECT
      1          AS tag,
      NULL       AS parent,
      EmployeeID AS [employee!1!EmployeeID],
      NULL       AS [customer!2!CustomerID],
      NULL       AS [customer!2!Region]
FROM Employees
UNION ALL
SELECT
      2,
      1,
      EmployeeID,
      CustomerID,
      Region
FROM Employees KEY JOIN SalesOrders
ORDER BY 3, 1
FOR XML EXPLICIT;
```

有关 EXPLICIT 模式的语法的信息，请参见“参数”一节第 657 页。

FOR XML EXPLICIT 示例

以下示例查询会检索有关雇员所下订单的信息。在本示例中，有以下三种类型的元素：<employee>、<order> 和 <department>。<employee> 元素具有 ID 属性和 name 属性，<order> 元素具有 date 属性，<department> 元素具有 name 属性。


```
SELECT
        1           tag,
        NULL        parent,
        EmployeeID  [employee!1!ID],
        GivenName   [employee!1!name],
        NULL        [order!2!date],
        NULL        [department!3!name]
FROM Employees
UNION ALL
SELECT
        2,
        1,
        EmployeeID,
        NULL,
        OrderDate,
        NULL
FROM Employees KEY JOIN SalesOrders
UNION ALL
SELECT
        3,
        1,
        EmployeeID,
        NULL,
        NULL,
        DepartmentName
FROM Employees e JOIN Departments d
   ON e.DepartmentID=d.DepartmentID
ORDER BY 3, 1
FOR XML EXPLICIT;
```

您将从此查询得到以下结果：

```
<employee ID="102" name="Fran">
   <department name="R &amp; D"/>
</employee>
<employee ID="105" name="Matthew">
   <department name="R &amp; D"/>
</employee>
<employee ID="129" name="Philip">
   <order date="2000-07-24"/>
   <order date="2000-07-13"/>
   <order date="2000-06-24"/>
   <order date="2000-06-08"/>
   ...
   <department name="Sales"/>
</employee>
<employee ID="148" name="Julie">
   <department name="Finance"/>
</employee>
...
```

使用 element 指令

如果您想生成子元素而不是属性，可以向查询中添加 element 指令，如下所示：

```
SELECT
        1           tag,
        NULL        parent,
        EmployeeID  [employee!1!id!element],
        GivenName   [employee!1!name!element],
        NULL        [order!2!date!element],
        NULL        [department!3!name!element]
FROM Employees
```

```
UNION ALL
SELECT
        2,
        1,
        EmployeeID,
        NULL,
        OrderDate,
        NULL
FROM Employees KEY JOIN SalesOrders
UNION ALL
SELECT
        3,
        1,
        EmployeeID,
        NULL,
        NULL,
        DepartmentName
FROM Employees e JOIN Departments d
   ON e.DepartmentID=d.DepartmentID
ORDER BY 3, 1
FOR XML EXPLICIT;
```

您将从此查询得到以下结果：

```
<employee>
   <id>102</id>
   <name>Fran</name>
   <department>
      <name>R &amp; D</name>
   </department>
</employee>
<employee>
   <id>105</id>
   <name>Matthew</name>
   <department>
      <name>R &amp; D</name>
   </department>
</employee>
<employee>
   <id>129</id>
   <name>Philip</name>
   <order>
      <date>2000-07-24</date>
   </order>
   <order>
      <date>2000-07-13</date>
   </order>
   <order>
      <date>2000-06-24</date>
   </order>
   ...
   <department>
      <name>Sales</name>
   </department>
</employee>
...
```

使用 hide 指令

在以下查询中，雇员 ID 用于为结果排序，但结果中不会出现雇员 ID，因为指定了 hide 指令：

```
SELECT
        1           tag,
```



```
        NULL        parent,
        EmployeeID  [employee!1!id!hide],
        GivenName   [employee!1!name],
        NULL        [order!2!date],
        NULL        [department!3!name]
FROM Employees
UNION ALL
SELECT
        2,
        1,
        EmployeeID,
        NULL,
        OrderDate,
        NULL
FROM Employees KEY JOIN SalesOrders
UNION ALL
SELECT
        3,
        1,
        EmployeeID,
        NULL,
        NULL,
        DepartmentName
FROM Employees e JOIN Departments d
   ON e.DepartmentID=d.DepartmentID
ORDER BY 3, 1
FOR XML EXPLICIT;
```

此查询会返回以下结果：

```
<employee name="Fran">
   <department name="R &amp; D"/>
</employee>
<employee name="Matthew">
   <department name="R &amp; D"/>
</employee>
<employee name="Philip">
   <order date="2000-04-21"/>
   <order date="2001-07-23"/>
   <order date="2000-12-30"/>
   <order date="2000-12-20"/>
   ...
   <department name="Sales"/>
</employee>
<employee name="Julie">
   <department name="Finance"/>
</employee>
...
```

使用 xml 指令

缺省情况下，当 FOR XML EXPLICIT 查询的结果包含的字符不是有效 XML 字符时，无效字符会被转义（有关信息，请参见“对非法的 XML 名称编码”一节第 651 页），除非该列属于 XML 类型。例如，以下查询会生成包含和符号 (&) 的 XML：

```
SELECT
        1               AS tag,
        NULL            AS parent,
        ID              AS [customer!1!ID!element],
        CompanyName     AS [customer!1!CompanyName]
FROM Customers
```

```
WHERE ID = '115'
FOR XML EXPLICIT;
```

在此查询所生成的结果中，和符号会被转义，因为该列不属于 XML 类型：

```
<Customers CompanyName="Sterling &amp;amp; Co.">
   <ID>115</ID>
</Customers>
```

xml 指令表明列值在插入生成的 XML 中时不进行引用。如果用 xml 指令执行与以上查询相同的查询：

```
SELECT
        1               AS tag,
        NULL            AS parent,
        ID              AS [customer!1!ID!element],
        CompanyName     AS [customer!1!CompanyName!xml]
FROM Customers
WHERE ID = '115'
FOR XML EXPLICIT;
```

结果中的和符号将不进行引用：

```
<customer>
 <ID>115</ID>
 <CompanyName>Sterling & Co.</CompanyName>
</customer>
```

请注意，此 XML 不是一个结构完好的 XML，因为它含有和符号，而该符号在 XML 中是一个特殊字符。在查询生成 XML 之后，由您负责确保该 XML 结构完好且有效：SQL Anywhere 并不检查所生成的 XML 是否结构完好和有效。

当您指定 xml 指令时，*AttributeName* 字段会被忽略，并且会生成元素而不是属性。

使用 cdata 指令

以下查询使用 cdata 指令在 CDATA 区段中返回客户名称：

```
SELECT
        1               AS tag,
        NULL            AS parent,
        ID              AS [product!1!ID],
        Description     AS [product!1!!cdata]
FROM Products
FOR XML EXPLICIT;
```

此查询生成的结果将在 CDATA 区段中列出对每种产品的说明。CDATA 区段中包含的数据不进行引用：

```
<product ID="300">
   <![CDATA[Tank Top]]>
</product>
<product ID="301">
   <![CDATA[V-neck]]>
</product>
<product ID="302">
   <![CDATA[Crew Neck]]>
</product>
<product ID="400">
   <![CDATA[Cotton Cap]]>
```



```
</product>
...
```

使用 SQL/XML 以 XML 形式获取查询结果

SQL/XML 是一项描述 XML 与 SQL 语言有效集成的标准草案：它描述了能够将 SQL 与 XML 结合使用的方法。通过使用所支持的函数，您可以编写查询来利用关系数据构建 XML 文档。

无效的名称和 SQL/XML

在 SQL/XML 中，不是合法 XML 名称的表达式（例如，包含空格的表达式）会被转义，方式与处理 FOR XML 子句一样。XML 类型的元素内容不会进行引用。

有关对无效表达式进行引用的详细信息，请参见“对非法的 XML 名称编码”一节第 651 页。

有关使用 XML 数据类型的信息，请参见“在关系数据库中存储 XML 文档”一节第 640 页。

使用 XMLAGG 函数

XMLAGG 函数用于从 XML 元素集合中生成 XML 元素林。XMLAGG 是一个集合函数，它会为查询中的所有行生成单一的 XML 集合结果。

在以下查询中，XMLAGG 用于为每行生成一个 <name> 元素，而 <name> 元素会按雇员姓名排序。指定 ORDER BY 子句是为了给 XML 元素排序：

```
SELECT XMLELEMENT( NAME Departments,
                   XMLATTRIBUTES ( DepartmentID ),
                   XMLAGG( XMLELEMENT( NAME name,
                                       Surname )
                                       ORDER BY Surname )
                 ) AS department_list
FROM Employees
GROUP BY DepartmentID
ORDER BY DepartmentID;
```

此查询会生成以下结果：

department_list
<Departments DepartmentID="100"> <name>Breault</name> <name>Cobb</name> <name>Diaz</name> <name>Driscoll</name> ... </Departments>
<Departments DepartmentID="200"> <name>Chao</name> <name>Chin</name> <name>Clark</name> <name>Dill</name> ... </Departments>

department_list
<Departments DepartmentID="300"> <name>Bigelow</name> <name>Coe</name> <name>Coleman</name> <name>Davidson</name> ... </Departments>
...

有关 XMLAGG 函数的详细信息，请参见“XMLAGG 函数 [Aggregate]”一节 《SQL Anywhere 服务器 - SQL 参考》。

使用 XMLCONCAT 函数

XMLCONCAT 函数通过连接所有传入的 XML 值来创建 XML 元素林。例如，以下查询会将 Employees 表中的每一名雇员的 <given_name> 元素和 <surname> 元素连接起来：

```
SELECT XMLCONCAT( XMLELEMENT( NAME given_name, GivenName ),
                  XMLELEMENT( NAME surname, Surname )
                ) AS "Employee_Name"
FROM Employees;
```

此查询会返回以下结果：

Employee_Name
<given_name>Fran</given_name> <surname>Whitney</surname>
<given_name>Matthew</given_name> <surname>Cobb</surname>
<given_name>Philip</given_name> <surname>Chin</surname>
<given_name>Julie</given_name> <surname>Jordan</surname>
...

有关详细信息，请参见“XMLCONCAT 函数 [String]”一节 《SQL Anywhere 服务器 - SQL 参考》。

使用 XMLELEMENT 函数

XMLELEMENT 函数利用关系数据来构建 XML 元素。您可以指定所生成元素的内容，而且，如果需要，还可以为该元素指定属性和属性内容。

生成嵌套元素

以下查询将生成嵌套的 XML，也就是为每个产品生成一个 <product_info> 元素，而该元素中还包含一些用于提供每种产品的名称、数量和说明的元素：

```
SELECT ID,
XMLELEMENT( NAME product_info,
            XMLELEMENT( NAME item_name, Products.name ),
            XMLELEMENT( NAME quantity_left, Products.Quantity ),
            XMLELEMENT( NAME description, Products.Size || ' ' ||
                        Products.Color || ' ' || Products.name )
          ) AS results
FROM Products
WHERE Quantity > 30;
```

此查询会生成以下结果：

ID	results
301	<product_info> <item_name>Tee Shirt </item_name> <quantity_left>54 </quantity_left> <description>Medium Orange Tee Shirt</description> </product_info>
302	<product_info> <item_name>Tee Shirt </item_name> <quantity_left>75 </quantity_left> <description>One Size fits all Black Tee Shirt </description> </product_info>
400	<product_info> <item_name>Baseball Cap </item_name> <quantity_left>112 </quantity_left> <description>One Size fits all Black Baseball Cap </description> </product_info>
...	...

指定元素内容

使用 XMLELEMENT 函数，您可以指定元素的内容。以下语句将生成一个包含内容 **hat** 的 XML 元素。

```
SELECT ID, XMLELEMENT( NAME product_type, 'hat' )
FROM Products
WHERE Name IN ( 'Baseball Cap', 'Visor' );
```

生成有属性的元素

通过在查询中加入 XMLATTRIBUTES 参数为元素添加属性。此参数指定属性名称和内容。以下语句会为每个项目的 name、Color 和 UnitPrice 生成一个属性。

```
SELECT ID, XMLELEMENT( NAME item_description,
                       XMLATTRIBUTES( Name,
                                      Color,
                                      UnitPrice )
                     ) AS item_description_element
FROM Products
WHERE ID > 400;
```

通过指定 AS 子句可以命名属性：

```
SELECT ID, XMLELEMENT( NAME item_description,
                       XMLATTRIBUTES ( UnitPrice AS
                                       price ),
                       Products.name
                     ) AS products
FROM Products
WHERE ID > 400;
```

有关详细信息，请参见“XMLELEMENT 函数 [String]”一节 《SQL Anywhere 服务器 - SQL 参考》。

示例

以下示例将 XMLELEMENT 与 HTTP web 服务搭配使用。

```
ALTER PROCEDURE "DBA"."http_header_example_with_table_proc"()
RESULT ( res LONG VARCHAR )
BEGIN
   DECLARE var LONG VARCHAR;
   DECLARE varval LONG VARCHAR;
   DECLARE I INT;
 DECLARE res LONG VARCHAR;
 DECLARE tabl XML;
   SET var  = NULL;
loop_h:
   LOOP
      SET var = NEXT_HTTP_HEADER( var );
      IF var IS NULL THEN LEAVE leave loop_h END IF;
      SET varval = http_header( var );
      -- ... do some action for <var,varval> pair...
        SET tabl = tabl ||
                       XMLELEMENT( name "tr",
                       XMLATTRIBUTES( 'left' AS "align", 'top' AS
"valign" ),
                       XMLELEMENT( name "td", var ),
                       XMLELEMENT( name "td", varval ) )  ;

   END LOOP;

    SET res = XMLELEMENT( NAME "table",
                  XMLATTRIBUTES( '' AS "BORDER", '10' as "CELLPADDING", '0' AS
"CELLSPACING" ),

                  XMLELEMENT( NAME "th",
                      XMLATTRIBUTES( 'left' AS "align", 'top' AS "valign" ),
                      'Header Name' ),
```

```
                    XMLELEMENT( NAME "th",
                        XMLATTRIBUTES( 'left' AS "align", 'top' AS "valign" ),
                        'Header Value' ),

                     tabl ) ;
    SELECT res;
END
```

使用 XMLFOREST 函数

使用 XMLFOREST 可以构建 XML 元素林。对于每个 XMLFOREST 参数，都会生成一个元素。

以下查询会生成一个 <item_description> 元素，其中又包含有 <name>、<color> 和 <price> 元素：

```
SELECT ID, XMLELEMENT( NAME item_description,
                       XMLFOREST( Name as name,
                                  Color as color,
                                  UnitPrice AS price )
                     ) AS product_info
FROM Products
WHERE ID > 400;
```

此查询将生成以下结果：

ID	product_info
401	<item_description> <name>Baseball Cap</name> <color>White</color> <price>10.00</price> </item_description>
500	<item_description> <name>Visor</name> <color>White</color> <price>7.00</price> </item_description>
501	<item_description> <name>Visor</name> <color>Black</color> <price>7.00</price> </item_description>
...	...

有关详细信息，请参见“XMLFOREST 函数 [String]”一节 《SQL Anywhere 服务器 - SQL 参考》。

使用 XMLGEN 函数

XMLGEN 函数用于在 XQuery 构造函数的基础上生成 XML 值。

以下查询生成的 XML 提供了有关 SQL Anywhere 示例数据库中客户订单的信息。它使用以下变量引用：

- **{$ID}** 使用 SalesOrders 表的 ID 列中的值为 <ID> 元素生成内容。
- **{$OrderDate}** 使用 SalesOrders 表的 OrderDate 列中的值为 <date> 元素生成内容。
- **{$Customers}** 使用 Customers 表的 CompanyName 列中的值为 <customer> 元素生成内容。

```
SELECT XMLGEN ( '<order>
                <ID>{$ID}</ID>
                <date>{$OrderDate}</date>
                <customer>{$Customers}</customer>
                </order>',
                SalesOrders.ID,
                SalesOrders.OrderDate,
                Customers.CompanyName AS Customers
                ) AS order_info
FROM SalesOrders JOIN Customers
ON Customers.ID = SalesOrders.CustomerID
ORDER BY SalesOrders.CustomerID;
```

此查询会生成以下结果：

order_info
<order> <ID>2001</ID> <date>2000-03-16</date> <customer>The Power Group</customer> </order>
<order> <ID>2005</ID> <date>2001-03-26</date> <customer>The Power Group</customer> </order>
<order> <ID>2125</ID> <date>2001-06-24</date> <customer>The Power Group</customer> </order>
<order> <ID>2206</ID> <date>2000-04-16</date> <customer>The Power Group</customer> </order>
...

生成属性

如果您希望订单 ID 号显示为 <order> 元素的属性，需如下所示编写查询（请注意，变量引用包含在双引号中，因为它指定了一个属性值）：

```
SELECT XMLGEN ( '<order ID="{$ID}">
                  <date>{$OrderDate}</date>
                  <customer>{$Customers}</customer>
                  </order>',
                SalesOrders.ID,
                SalesOrders.OrderDate,
```

```
                Customers.CompanyName AS Customers
              ) AS order_info
FROM SalesOrders JOIN Customers
ON Customers.ID = SalesOrders.CustomerID
ORDER BY SalesOrders.OrderDate;
```

此查询会生成以下结果：

order_info
<order ID="2131"> <date>2000-01-02</date> <customer>BoSox Club</customer> </order>
<order ID="2065"> <date>2000-01-03</date> <customer>Bloomfield's</customer> </order>
<order ID="2126"> <date>2000-01-03</date> <customer>Leisure Time</customer> </order>
<order ID="2127"> <date>2000-01-06</date> <customer>Creative Customs Inc.</customer> </order>
...

在两个结果集中，客户名 Bloomfield's 都进行了引用，变成了 Bloomfield's，因为撇号在 XML 中是一个特殊符号，而且从中生成 <customer> 元素的列不是 XML 类型。

有关对 XMLGEN 中的非法字符进行引用的详细信息，请参见“无效的名称和 SQL/XML”一节 第 666 页。

指定 XML 文档的标头信息

SQL Anywhere 支持的 FOR XML 子句和 SQL/XML 函数不会在它们所生成的 XML 文档中加入版本声明信息。您可以使用 XMLGEN 函数来生成标头信息。

```
SELECT XMLGEN( '<?xml version="1.0"
                encoding="ISO-8859-1" ?>
                <r>{$x}</r>',
                (SELECT GivenName, Surname
      FROM Customers FOR XML RAW) AS x );
```

这会生成以下结果：

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<r>
 <row GivenName="Michaels" Surname="Devlin"/>
 <row GivenName="Beth" Surname="Reiser"/>
 <row GivenName="Erin" Surname="Niedringhaus"/>
 <row GivenName="Meghan" Surname="Mason"/>
 ...
</r>
```


有关 XMLGEN 函数的详细信息，请参见“XMLGEN 函数 [String]”一节 《SQL Anywhere 服务器 - SQL 参考》。

远程数据和批量操作

本节介绍如何装载和卸载数据库以及如何访问远程数据。

导入和导出数据

目录

术语**批量操作**用于描述导入和导出数据的过程。批量操作必须由具有 DBA 权限的用户执行，并且不属于典型的最终用户应用程序。批量操作可能会影响并发和事务日志，因此应在用户未连接到数据库时执行。

以下是导入或导出数据的典型情况：

- 将一组初始数据导入一个新数据库
- 建立数据库的新副本，该副本可能具有已修改的结构
- 将数据从您的数据库导出以用于其它应用程序，例如电子表格
- 创建数据库的抽取用于复制或同步
- 修复损坏的数据库
- 重建数据库以提高其性能
- 获取数据库软件的更高版本并完成软件升级

批量操作的性能问题

批量操作的性能取决于多个因素，包括操作相对于数据库服务器来说是内部操作还是外部操作。

内部批量操作

内部批量操作也称为*服务器端*批量操作，是通过使用 LOAD TABLE 和 UNLOAD 语句由数据库服务器执行的导入和导出操作。

执行内部批量操作时，您可以从 ASCII 文本文件或 Adaptive Server Enterprise BCP 文件中装载或者卸载到这些文件中。这些文件可能存在于数据库服务器所在的计算机或客户端计算机上。被写入或读取的文件的指定路径是相对于数据库服务器的路径。内部批量操作是将数据导入和导出数据库的最快速的方法。

外部批量操作

外部批量操作也称为*客户端*批量操作，是通过使用 INPUT 和 OUTPUT 语句由客户端（如 Interactive SQL）执行的导入和导出操作。当客户端发出 INPUT 语句时，对于处理在 INPUT 语句中指定的文件时所读取的每一行，都会在事务日志中记录一个 INSERT 语句。因此，客户端装载比服务器端装载要慢得多。此外，执行 INPUT 时 INSERT 触发器将触发。

使用 OUTPUT 语句可将 SELECT 语句的结果集写入多种不同的文件格式。

对于外部批量操作，所读取或写入的文件的指定路径相对于正在运行客户端应用程序的计算机的路径。

另请参见

- “LOAD TABLE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “UNLOAD 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “INPUT 语句 [Interactive SQL]”一节 《SQL Anywhere 服务器 - SQL 参考》
- “OUTPUT 语句 [Interactive SQL]”一节 《SQL Anywhere 服务器 - SQL 参考》
- “导入数据的性能提示”一节第 680 页
- “-b 服务器选项”一节 《SQL Anywhere 服务器 - 数据库管理》

批量操作的数据恢复问题

您可以在批量操作模式（-b 服务器选项）下运行数据库服务器。使用此按钮时，数据库服务器不会执行某些重要功能。具体包括：

功能	影响
维护事务日志	不对更改进行记录。每次执行 COMMIT 都导致执行检查点操作。
锁定所有记录	无严重影响。

或者，您可能需要确保批量装载的数据在恢复时仍然可用。您可以通过保持原始数据源完整且使它们处于原始位置来实现此目的。您还可以使用某些可用于 LOAD TABLE 语句的记录选项，这些选项允许将批量装载的数据记录到事务日志中。请参见“LOAD TABLE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

小心
在使用批量操作模式之前和之后都应对数据库进行备份，这是因为在此模式中数据库未受保护，可能会因介质故障而受损。

另请参见

- “-b 服务器选项”一节 《SQL Anywhere 服务器 - 数据库管理》

导入数据

导入数据是一项管理任务，此任务以批量操作模式将数据读入数据库。使用 SQL Anywhere 可执行以下操作：

- 从文本文件导入整个表或表的某些部分
- 从变量中导入数据
- 通过使用脚本自动执行导入过程来连续导入多个表
- 在表中插入或添加数据
- 替换表中的数据
- 在导入前或导入过程中创建表
- 从客户端计算机上的文件装载数据
- 使用 BCP FORMAT 子句在 SQL Anywhere 和 Adaptive Server Enterprise 之间传输文件

如果尝试创建一个完整的新数据库，请考虑使用 LOAD TABLE 装载数据，以获得最佳性能。

有关卸载和重装完整数据库的详细信息，请参见“重建数据库”一节第 709 页。

另请参见

- “LOAD TABLE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “UNLOAD 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “INPUT 语句 [Interactive SQL]”一节 《SQL Anywhere 服务器 - SQL 参考》
- “OUTPUT 语句 [Interactive SQL]”一节 《SQL Anywhere 服务器 - SQL 参考》
- “导入数据的性能提示”一节第 680 页
- “批量操作的性能问题”一节第 678 页
- “-b 服务器选项”一节 《SQL Anywhere 服务器 - 数据库管理》
- “用于导入的表结构”一节第 694 页
- “访问客户端计算机上的数据”一节第 706 页

导入数据的性能提示

导入大量数据可能要花费很多时间。可以通过以下方法来节省时间：

- 将数据文件与数据库放在不同的物理磁盘驱动器上。这可以避免装载过程中过多的磁头移动。
- 扩展数据库的大小。此命令允许在需要空间之前大规模扩展数据库，而不是在需要空间时小规模扩展数据库。该命令还可在装载大量数据时提高性能，并使数据库在文件系统中保持更好的连续性。请参见“ALTER DBSPACE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。
- 使用临时表装载数据。当需要重复装载一组数据或需要合并具有不同结构的表时，本地或全局临时表将十分有用。
- 如果使用 LOAD TABLE 语句，则启动数据库服务器时不使用 -b 选项（批量操作模式）。

- 如果使用 INPUT 或 OUTPUT 语句，则在数据库服务器所在的计算机上运行 Interactive SQL 或客户端应用程序。通过网络装载数据将增加额外的通信开销。最好在数据库服务器不忙时装载新数据。

另请参见

- “LOAD TABLE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “INPUT 语句 [Interactive SQL]”一节 《SQL Anywhere 服务器 - SQL 参考》
- “OUTPUT 语句 [Interactive SQL]”一节 《SQL Anywhere 服务器 - SQL 参考》
- “-b 服务器选项”一节 《SQL Anywhere 服务器 - 数据库管理》

使用 [导入向导] 导入数据

使用 Interactive SQL [**导入向导**] 可以选择数据的来源、格式和目标表。可以从 TEXT 和 FIXED 格式文件导入数据。可以将数据导入现有表或新表中。还可以使用 [**导入向导**] 在以下数据库之间导入数据：

- 不同类型的数据库，如 SQL Anywhere 数据库和 UltraLite 数据库之间。
- 不同版本的数据库（只要具备每一个数据库的 ODBC 驱动程序），如 SQL Anywhere 版本 11.0.0 数据库和 SQL Anywhere 版本 10.0.0 数据库之间。

可以在以下情况下使用 Interactive SQL [**导入向导**]：

- 要在导入数据的同时创建表
- 希望使用点击界面以非文本格式导入数据

♦ 导入数据

1. 在 Interactive SQL 中，选择 [**数据**] » [**导入**]。
2. 按照 [**导入向导**] 中的说明进行操作。

♦ 将数据从文件导入 SQL Anywhere 示例数据库

1. 使用以下值（在一行中输入）创建并保存名为 *import.txt* 的文本文件：

```
100,500,'Chan','Julia',100,'300 Royal Drive',
'Springfield','OR','USA','97015','6175553985',
'A','017239033',55700,'1984-09-29',,'1968-05-05',
1,1,0,'F'
```

2. 在 Interactive SQL 中，选择 [**数据**] » [**导入**]。
3. 选择 [**文本文件中**] 并单击 [**下一步**]。
4. 在 [**文件名**] 字段中，键入 **import.txt**。
5. 选择 [**现有表中**]。
6. 选择 [**Employees**]，然后单击 [**下一步**]。

7. 在 [**字段分隔符**] 列表中，选择 [**逗号 (,)**]。单击 [**下一步**]。
8. 单击 [**导入**]。
9. 单击 [**关闭**]。

   导入完成后，由向导创建的 SQL 语句存储在命令历史记录中。

   可以从 [**SQL**] 菜单中选择 [**上一个 SQL**] 以查看生成的 SQL 语句。

   由 [**导入向导**] 生成的 IMPORT 语句出现在 [**SQL 语句**] 窗格中：

```
--  Generated by the Import Wizard
INPUT INTO "GROUPO"."Employees" from 'C:\\Tobedeleted\\import.txt'
FORMAT TEXT ESCAPES ON ESCAPE CHARACTER '\\' DELIMITED BY ',' ENCODING
'Cp1252'
```

♦ 将数据从 SQL Anywhere 示例数据库导入 UltraLite 数据库

1. 连接到 UltraLite 数据库，如 *C:\Documents and Settings\All Users\Documents\SQL Anywhere 11\Samples\UltraLite\CustDB\custdb.udb*。
2. 在 Interactive SQL 中，选择 [**数据**] » [**导入**]。
3. 单击 [**数据库中**]。单击 [**下一步**]。
4. 在 [**数据库类型**] 列表中选择 [**SQL Anywhere**]
5. 在 [**标识**] 选项卡中，单击 [**ODBC 数据源名称**]，然后键入 [**SQL Anywhere 11 Demo**]。单击 [**下一步**]。
6. 在 [**表名称**] 列表中，选择 [**Customers**]。单击 [**下一步**]。
7. 单击 [**新表中**]。
8. 在 [**所有者**] 字段中键入 **dba**。
9. 在 [**表名称**] 字段中，键入 **SQLAnyCustomers**。
10. 单击 [**导入**]。
11. 单击 [**关闭**]。
12. 要查看生成的 SQL 语句，请选择 [**SQL**] » [**上一个 SQL**]。

    由 [**导入向导**] 创建并使用的 IMPORT 语句出现在 [**SQL 语句**] 窗格中。

```
--  Generated by the Import Wizard
INPUT USING 'DSN=SQL Anywhere 11 Demo;CON='''''
FROM "GROUPO.Customers" INTO "dba"."SQLAnyCustomers"
CREATE TABLE ON
```

使用 INPUT 语句导入数据

使用 INPUT 语句可将不同文件格式的数据导入现有表或新表中。如果有数据库的 ODBC 驱动程序，可使用 USING 子句从不同类型的数据库和不同版本的 SQL Anywhere 数据库导入数据。

使用 INPUT 语句可以从 TEXT 和 FIXED 格式导入数据。要从其它文件格式导入数据，可以将 USING 子句和 ODBC 数据源一起使用。

可以使用缺省输入格式，或者为每个 INPUT 语句指定文件格式。因为 INPUT 语句是 Interactive SQL 命令，所以不能在任何复合语句（例如 IF 语句）或存储过程中使用 INPUT 语句。

如果要从文件或其它数据库导入数据，请使用 INPUT 语句。

有关详细信息，请参见“INPUT 语句 [Interactive SQL]”一节 《SQL Anywhere 服务器 - SQL 参考》。

有关实例化视图的注意事项

对于快速视图，如果尝试将批量数据装载到基础表，将会返回错误。必须首先在视图中截断数据，然后再执行批量装载操作。

对于手动视图，可以将批量数据装载到基础表。但是，视图中的数据会一直保持失效状态，直到下次刷新为止。

在尝试执行批量装载操作（例如，对表执行 INPUT）之前，请首先考虑在相关的实例化视图中截断数据。装载数据之后，请刷新视图。请参见“TRUNCATE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》和“REFRESH MATERIALIZED VIEW 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

有关文本索引的注意事项

对于快速文本索引，在执行了批量装载操作（例如对基础表执行 INPUT）之后更新文本索引可能需要一些时间，即使自动执行更新也是如此。对于手动文本索引，即使是刷新也可能需要一些时间。

在执行批量装载操作（例如，对表执行 INPUT）之前，请首先考虑删除相关的文本索引。装载数据之后，重新创建文本索引。请参见“DROP TEXT INDEX 语句”一节 《SQL Anywhere 服务器 - SQL 参考》和“CREATE TEXT INDEX 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

对数据库的影响

使用 INPUT 语句时更改将记录在事务日志中。如果发生介质故障，则有详细的更改记录。但由于此方法将所有行都写入事务日志，所以使用此方法导入大量数据会对性能产生影响。

相比之下，LOAD TABLE 语句不会将每一行都保存在事务日志中，因此执行速度比 INPUT 语句快。但是就支持的数据库和文件格式而言，INPUT 语句更加灵活。

♦ 导入数据（INPUT 语句）

1. 使用以下值（在一行中输入）创建并保存名为 *new_employees.txt* 的文本文件：

```
101,500,'Chan','Julia',100,'300 Royal Drive',
'Springfield','OR','USA','97015','6175553985',
'A','017239033',55700,'1984-09-29',,'1968-05-05',
1,1,0,'F'
```

2. 打开 Interactive SQL 并连接到 SQL Anywhere 11 Demo 数据库。
3. 在 [**SQL 语句**] 窗格中输入 INPUT 语句。

```
INPUT INTO Employees
FROM c:\new_employees.txt
FORMAT TEXT;
SELECT * FROM Employees;
```

在本语句中，SQL Anywhere 11 Demo 数据库中目标表的名称为 Employees，*new_employees.txt* 是源文件的名称。

4. 执行该语句。

   如果导入成功，则 [**消息**] 选项卡显示导入数据花费的时间。如果导入不成功，则将显示一条消息指示导入不成功的原因。

♦ 将数据从 Microsoft Excel 电子表格导入 SQL Anywhere 数据库

1. 在 Microsoft Excel 中打开电子表格。
2. 在 Excel 中，选择要导入的单元格，然后选择 [**插入**] » [**名称**] » [**定义**]。

   键入所选单元格的名称，如 **myData**。
3. 单击 [**确定**]。
4. 保存并关闭电子表格。
5. 为电子表格创建 ODBC 数据源。
   - 选择 [**开始**] » [**程序**] » [**SQL Anywhere 11**] » [**ODBC 管理器**]。
   - 选择 [**用户 DSN**] 选项卡，为当前用户创建 DSN，或选择 [**系统 DSN**] 选项卡创建整个系统范围的 DSN。
   - 单击 [**添加**]。

     从驱动程序列表中，选择 **Microsoft Excel Driver**，然后单击 [**完成**]。
   - 指定所需参数，单击 [**确定**]，关闭该窗口并创建数据源。

     例如，在 [**数据源名**] 字段中键入 **myExcelFile**。单击 [**选择工作簿**] 并浏览找到 Excel 电子表格文件。
   - 单击 [**确定**] 保存 DSN。
6. 打开 Interactive SQL 并连接到 SQL Anywhere 数据库。
7. 执行以下 INPUT 语句，从 Excel 电子表格导入数据并将其保存到名为 t 的新表中：

```
INPUT USING 'dsn=myExcelFile;DSN=myExcelFile'
FROM "myData" INTO "t"
CREATE TABLE ON
```

使用 LOAD TABLE 语句导入数据

使用 LOAD TABLE 语句可将位于数据库服务器或客户端计算机上的数据以文本/ASCII 格式导入现有表中。

还可以使用 LOAD TABLE 语句从另一个表或从值的表达式（例如，从函数或系统过程结果）导入数据。

LOAD TABLE 语句将行添加到表中，但不替换它们。

使用 LOAD TABLE 语句（无 WITH ROW LOGGING 和 WITH CONTENT LOGGING 选项）装载数据比使用 INPUT 语句装载数据快得多。

使用 LOAD TABLE 语句装载数据，触发器不触发。

有关实例化视图的注意事项

对于快速视图，如果尝试将批量数据装载到基础表，将会返回错误。必须首先在视图中截断数据，然后再执行批量装载操作。

对于手动视图，可以将批量数据装载到基础表；但是，在下次刷新前，视图中的数据会一直为失效状态。

在尝试执行批量装载操作（例如，对表执行 LOAD TABLE）之前，请首先考虑在相关的实例化视图中截断数据。装载数据之后，请刷新视图。请参见“TRUNCATE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》和“REFRESH MATERIALIZED VIEW 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

有关文本索引的注意事项

对于快速文本索引，在执行了批量装载操作（例如对基础表执行 LOAD TABLE）之后更新文本索引可能需要一些时间，即使自动执行更新也是如此。对于手动文本索引，即使是刷新也可能需要一些时间。

在执行批量装载操作（例如，对表执行 LOAD TABLE）之前，请考虑删除相关的文本索引。装载数据之后，重新创建文本索引。请参见“DROP TEXT INDEX 语句”一节 《SQL Anywhere 服务器 - SQL 参考》和“CREATE TEXT INDEX 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

数据库恢复和同步的注意事项

缺省情况下，从文件装载数据时（例如 `LOAD TABLE table-name FROM filename;`），在事务日志中仅记录 LOAD TABLE 语句，而不记录正在装载数据的实际行。如果原始装载文件已更改、移动或删除，这就会在使用事务日志尝试恢复数据库时带来问题。这还意味着参与同步或复制的数据库不会获得新数据。

为解决恢复和同步注意事项问题，有两个记录选项可用于 LOAD TABLE 语句：WITH ROW LOGGING 为已装载的所有行在事务日志中创建 INSERT 语句，而 WITH CONTENT LOGGING 将已装载的行组成块并将块记录在事务日志中。即使已装载数据的数据源不再可用，这些选项也允许重复进行装载操作。请参见“LOAD TABLE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

数据库镜像的注意事项

如果数据库参与镜像，则请谨慎使用 LOAD TABLE 语句。例如，如果正从文件装载数据，则应考虑文件是否可用于在镜像服务器上装载，或者从中装载的数据源中的数据是否会在镜像数据库处理装载时发生更改。如果存在任何一个上述风险，请考虑指定 WITH ROW LOGGING 或 WITH CONTENT LOGGING 作为 LOAD TABLE 语句中的记录级别。这样，装载到镜像数据库的数据与

在镜像数据库中装载的数据就是相同的。请参见“LOAD TABLE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

另请参见

- “访问客户端计算机上的数据”一节第 706 页
- “数据库镜像简介”一节 《SQL Anywhere 服务器 - 数据库管理》
- “INPUT 语句 [Interactive SQL]”一节 《SQL Anywhere 服务器 - SQL 参考》
- “LOAD TABLE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》

使用 INSERT 语句导入数据

使用 INSERT 语句可将行添加到数据库中。由于 INSERT 语句中包括了目标表中的导入数据，因此可将其视为交互式输入。还可以使用具有远程数据访问的 INSERT 语句从另一个数据库（而不是文件）导入数据。

可以在以下情况下使用 INSERT 语句导入数据：

- 要将少量数据导入到单个表中
- 文件格式灵活
- 要从外部数据库而不是从文件导入远程数据

INSERT 语句提供 ON EXISTING 子句以指定已在目标表中找到正在插入的行时要采取的操作。但是，如果预期有许多满足 ON EXISTING 条件的行，则应考虑使用 MERGE 语句。MERGE 语句提供更多对用于匹配行的操作的控制。该语句还提供更复杂的语法来定义构成匹配的元素。请参见“MERGE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

有关实例化视图的注意事项

对于快速视图，如果尝试将批量数据装载到基础表，将会返回错误。必须首先在视图中截断数据，然后再执行批量装载操作。

对于手动视图，可以将批量数据装载到基础表；但是，在下次刷新前，视图中的数据会一直为失效状态。

在尝试执行批量装载操作（例如，对表执行 INSERT）之前，请首先考虑在相关的实例化视图中截断数据。装载数据之后，请刷新视图。请参见“TRUNCATE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》和“REFRESH MATERIALIZED VIEW 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

有关文本索引的注意事项

对于快速文本索引，在执行了批量装载操作（例如对基础表执行 INSERT）之后更新文本索引可能需要一些时间，即使自动执行更新也是如此。对于手动文本索引，即使是刷新也可能需要一些时间。

在执行批量装载操作（例如，对表执行 INSERT）之前，请首先考虑删除相关的文本索引。装载数据之后，重新创建文本索引。请参见“DROP TEXT INDEX 语句”一节 《SQL Anywhere 服务器 - SQL 参考》和“CREATE TEXT INDEX 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

对数据库的影响

使用 INSERT 语句时更改将记录在事务日志中。这意味着如果发生涉及数据库文件的介质故障，则可以通过事务日志恢复关于所做更改的信息。

♦ **导入数据（INSERT 语句）：**

在以下示例中，数据被添加到 SQL Anywhere 示例数据库的 Departments 表中。

1. 确保目标表存在。
2. 执行 INSERT 语句。例如，

   以下示例在 SQL Anywhere 示例数据库的 Departments 表中插入新行。

```
INSERT
INTO Departments ( DepartmentID, DepartmentName, DepartmentHeadID )
VALUES ( 700, 'Training', 501)
SELECT * FROM Departments;
```

   插入值操作将新数据添加到现有表中。

另请参见

- “事务日志”一节 《SQL Anywhere 服务器 - 数据库管理》
- “INSERT 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “LOAD TABLE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “INPUT 语句 [Interactive SQL]”一节 《SQL Anywhere 服务器 - SQL 参考》

使用 MERGE 语句导入数据

使用 MERGE 语句可执行更新操作和更新大量表数据。合并数据时，可以指定源数据中的行与目标数据中的行匹配或不匹配时要采取的操作。

定义合并行为

为了便于说明，以下是 MERGE 语句语法的简化版本。有关 MERGE 语句的完整语法，请参见“MERGE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

MERGE INTO *target-object*
USING *source-object*
ON *merge-search-condition*
{ **WHEN MATCHED** | **WHEN NOT MATCHED** } [...]

数据库执行合并操作时，该语句将根据 ON 子句中包含的定义比较 *source-object* 和 *target-object* 中的行以找到匹配或不匹配的行。如果 *target-table* 中至少存在一行使 *merge-search-condition* 的值为 true，则将 *source-object* 中的行视为匹配。

source-object 可以是基表、视图、实例化视图、派生表或过程的结果。*target-object* 可以是上述对象中除实例化视图和过程以外的任意一种。有关这些对象类型的限制的更多信息，请参见“MERGE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

ANSI SQL/2003 标准不允许在合并操作中 *target-object* 中的行被 *source-object* 中的多个行更新。

一旦 *source-object* 中的行被视为匹配或不匹配，将针对相应的匹配或非匹配 WHEN 子句（WHEN MATCHED 或 WHEN NOT MATCHED）进行评估。WHEN MATCHED 子句定义对 *target-object* 中的行执行的操作（例如 WHEN MATCHED ...UPDATE 指定更新 *target-object* 中的行）。WHEN NOT MATCHED 子句定义使用 *source-object* 的非匹配行对 *target-object* 执行的操作。

可指定不受限制的 WHEN 子句，按照指定的顺序对其进行处理。还可以在 WHEN 子句内使用 AND 子句指定对行的子集的操作。例如，以下 WHEN 子句根据用于匹配行的 Quantity 列的值来定义要执行的不同操作：

```
WHEN MATCHED AND myTargetTable.Quantity<=500 THEN SKIP
WHEN MATCHED AND myTargetTable.Quantity>500 THEN UPDATE SET
myTargetTable.Quantity=500
```

合并操作中的分支

通过操作对匹配行和非匹配行的分组称作**分支**，每一组称为一个**分支**。**分支**相当于一个 WHEN MATCHED 或 WHEN NOT MATCHED 子句。例如，一个分支可能包含必须被插入的 *source-object* 中的非匹配行集。只有在所有分支活动完成后（已计算 *source-object* 中的所有行），才能开始执行分支操作。数据库服务器根据指定 WHEN 子句的顺序执行分支的操作。

分支中一旦有 *source-object* 中的非匹配行或者 *source-object* 和 *target-object* 中的匹配行对，就不会对后续分支进行评估。这样指定 WHEN 子句的顺序就非常重要。

将忽略 *source-object* 中被视为匹配或非匹配但不属于任何分支的行（也就是说，它不满足任何 WHEN 子句）。当 WHEN 子句包含 AND 子句且行不满足任何 AND 子句的条件时，可能发生这种情况。在这种情况下，由于没有对其定义任何操作，因此忽略该行。

在事务日志中，修改数据的操作记录为各个 INSERT、UPDATE 和 DELETE 语句。

在目标表中定义的触发器

合并操作期间执行每个 INSERT、UPDATE 和 DELETE 语句时触发器正常触发。例如，处理为其定义 UPDATE 操作的分支时，数据库服务器将：

1. 触发所有 BEFORE UPDATE 触发器
2. 触发任何行级 UPDATE 触发器时，对行的候选集执行 UPDATE 语句
3. 触发 AFTER UPDATE 触发器

如果影响了将要在另一分支中更新的行，则合并操作期间 *target-table* 中的触发器会导致冲突。例如，假定在行 A 中执行了一项操作，从而触发了删除行 B 的触发器。但行 B 有一个为其定义的、尚未执行的操作。当无法对行执行操作时，合并操作失败，回退所有更改并返回错误。

将用多个触发器操作定义的触发器视为对同一主体将每个触发器操作都指定一次（也就是说，等效于定义单独的触发器，每个触发器具备一个触发器操作）。

有关快速实例化视图的注意事项

如果 MERGE 语句更新大量的行，可能会影响数据库服务器性能。如果要更新大量的行，请在对表执行 MERGE 语句之前首先考虑截断相关快速实例化视图中的数据。执行 MERGE 语句之后，请执行 REFRESH MATERIALIZED VIEW 语句。请参见“REFRESH MATERIALIZED VIEW 语句”一节 《SQL Anywhere 服务器 - SQL 参考》和“TRUNCATE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

有关文本索引的注意事项

如果 MERGE 语句更新大量的行，可能会影响数据库服务器性能。在对表执行 MERGE 语句之前，请考虑删除相关的文本索引。执行了 MERGE 语句后，重新创建文本索引。请参见“DROP TEXT INDEX 语句”一节 《SQL Anywhere 服务器 - SQL 参考》和“CREATE TEXT INDEX 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

示例 1

假定您拥有销售夹克衫和毛衣的小型业务。夹克衫的原料价格上升了 5%，您想要调整价格与其匹配。使用以下 CREATE TABLE 语句，创建名为 myProducts 的小表以保存所销售的夹克衫和毛衣的当前价格信息。随后的 INSERT 语句给 myProducts 填充数据。

```
CREATE TABLE myProducts (
   product_id    NUMERIC(10),
   product_name  CHAR(20),
   product_size  CHAR(20),
   product_price NUMERIC(14,2));
INSERT INTO myProducts VALUES (1, 'Jacket', 'Small', 29.99);
INSERT INTO myProducts VALUES (2, 'Jacket', 'Medium', 29.99);
INSERT INTO myProducts VALUES (3, 'Jacket', 'Large', 39.99);
INSERT INTO myProducts VALUES (4, 'Sweater', 'Small', 18.99);
INSERT INTO myProducts VALUES (5, 'Sweater', 'Medium', 18.99);
INSERT INTO myProducts VALUES (6, 'Sweater', 'Large', 19.99);
SELECT * FROM myProducts;
```

product_id	product_name	product_size	product_price
1	Jacket	Small	29.99
2	Jacket	Medium	29.99
3	Jacket	Large	39.99
4	Sweater	Small	18.99
5	Sweater	Medium	18.99
6	Sweater	Large	19.99

现在，使用以下语句创建另一名为 myPrices 的表以保存夹克衫的价格更改信息。结束时添加 SELECT 语句，以便在执行合并操作之前可以看到 myPrices 表的内容。

```
CREATE TABLE myPrices (
   product_id    NUMERIC(10),
   product_name  CHAR(20),
```

```
   product_size  CHAR(20),
   product_price NUMERIC(14,2),
   new_price     NUMERIC(14,2));
INSERT INTO myPrices (product_id) VALUES (1);
INSERT INTO myPrices (product_id) VALUES (2);
INSERT INTO myPrices (product_id) VALUES (3);
INSERT INTO myPrices (product_id) VALUES (4);
INSERT INTO myPrices (product_id) VALUES (5);
INSERT INTO myPrices (product_id) VALUES (6);
COMMIT;
SELECT * FROM myPrices;
```

product_id	product_name	product_size	product_price	new_price
1	(NULL)	(NULL)	(NULL)	(NULL)
2	(NULL)	(NULL)	(NULL)	(NULL)
3	(NULL)	(NULL)	(NULL)	(NULL)
4	(NULL)	(NULL)	(NULL)	(NULL)
5	(NULL)	(NULL)	(NULL)	(NULL)
6	(NULL)	(NULL)	(NULL)	(NULL)

使用以下 MERGE 语句将数据从 myProducts 表合并到 myPrices 表。请注意，*source-object* 是派生表，它经过过滤仅包含 product_name 为 Jacket 的行。另请注意，如果 *target-object* 和 *source-object* 的 product_id 列中的值匹配，则 ON 子句指定其中的行匹配。

```
MERGE INTO myPrices p
USING ( SELECT
     product_id,
     product_name,
     product_size,
     product_price
  FROM myProducts
  WHERE product_name='Jacket') pp
ON (p.product_id = pp.product_id)
WHEN MATCHED THEN
  UPDATE SET
    p.product_id=pp.product_id,
    p.product_name=pp.product_name,
    p.product_size=pp.product_size,
    p.product_price=pp.product_price,
    p.new_price=pp.product_price * 1.05;
SELECT * FROM myPrices;
```

product_id	product_name	product_size	product_price	new_price
1	Jacket	Small	29.99	31.49
2	Jacket	Medium	29.99	31.49
3	Jacket	Large	39.99	41.99

product_id	product_name	product_size	product_price	new_price
4	(NULL)	(NULL)	(NULL)	(NULL)
5	(NULL)	(NULL)	(NULL)	(NULL)
6	(NULL)	(NULL)	(NULL)	(NULL)

product_id 4、5 和 6 的列值都保持 NULL，因为这些产品与 myProducts 表（其产品为 (`product_name='Jacket'`)）中的任何行都不匹配。

示例 2

以下示例使用 myTargetTable 的主键值来匹配行，以此来合并 mySourceTable 表和 myTargetTable 表中的行。如果 mySourceTable 中行的值与 myTargetTable 的主键列值相同，则将该行视为匹配行。

```
MERGE INTO myTargetTable
   USING mySourceTable ON PRIMARY KEY
   WHEN NOT MATCHED THEN INSERT
   WHEN MATCHED THEN UPDATE;
```

WHEN NOT MATCHED THEN INSERT 子句指定，在 mySourceTable 中找到的、但在 myTargetTable 中找不到的行必须添加到 myTargetTable。WHEN MATCHED THEN UPDATE 子句指定，将 myTargetTable 的匹配行更新为 mySourceTable 中的值。

下面的语法与以上语法等效。它假定 myTargetTable 具有列（I1、I2、...I*n*），并且主键定义在列（I1、I2）上。此表包含三列：$N + 1$

```
MERGE INTO myTargetTable ( I1, I2, .. ., In )
   USING mySourceTable ON myTargetTable.I1 = mySourceTable.U1
      AND myTargetTable.I2 = mySourceTable.U2
   WHEN NOT MATCHED
      THEN INSERT ( I1, I2, .. In )
         VALUES ( mySourceTable.U1, mySourceTable.U2, ..., mySourceTable.Un )
   WHEN MATCHED
      THEN UPDATE SET
      myTargetTable.I1 = mySourceTable.U1,
      myTargetTable.I2 = mySourceTable.U2,
      ...
      myTargetTable.In = mySourceTable.Un;
```

使用 RAISERROR 操作

为匹配或非匹配动作指定的操作之一是 RAISERROR。如果满足 WHEN 子句的条件，RAISERROR 允许合并操作失败。

指定 RAISERROR 时，缺省条件下数据库服务器返回 SQLSTATE 23510 和 SQLCODE -1254。另外，在 RAISERROR 关键字之后，可通过指定 *error_number* 参数自定义返回的 SQLCODE。请参见“MERGE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

指定自定义 SQLCODE 对于以后尝试确定导致错误产生的特定情况会有益处。

自定义 SQLCODE 必须是大于 17000 的正整数，并且可以指定为数字或变量。

以下语句简单说明了定制自定义 SQLCODE 如何影响返回内容：

按如下所示创建表 targetTable：

```
CREATE TABLE targetTable( c1 int );
INSERT INTO targetTable VALUES( 1 );
COMMIT;
```

以下语句返回错误 SQLSTATE = '23510' 和 SQLCODE = -1254：

```
MERGE INTO targetTable
   USING (SELECT 1 c1 ) AS sourceData
   ON targetTable.c1 = sourceData.c1
   WHEN MATCHED THEN RAISERROR;
SELECT sqlstate, sqlcode;
```

以下语句返回错误 SQLSTATE = '23510' 和 SQLCODE = -17001：

```
MERGE INTO targetTable
   USING (SELECT 1 c1 ) AS sourceData
   ON targetTable.c1 = sourceData.c1
   WHEN MATCHED THEN RAISERROR 17001
   WHEN NOT MATCHED THEN RAISERROR 17002;
SELECT sqlstate, sqlcode;
```

以下语句返回错误 SQLSTATE = '23510' 和 SQLCODE = -17002：

```
MERGE INTO targetTable
   USING (SELECT 2 c1 ) AS sourceData
   ON targetTable.c1 = sourceData.c1
   WHEN MATCHED THEN RAISERROR 17001
   WHEN NOT MATCHED THEN RAISERROR 17002;
SELECT sqlstate, sqlcode;
```

使用代理表导入数据

代理表是一种本地表，它所包含的元数据可以用来像访问本地表一样访问远程数据库服务器上的表。这样便可直接导入数据。

可以在以下情况下使用代理表导入数据：

- 有权访问远程数据
- 要直接从另一个数据库导入数据

对数据库的影响

使用代理表导入数据时更改将记录在事务日志中。这意味着如果发生涉及数据库文件的介质故障，则可以从事务日志恢复关于所做更改的信息。

如何使用代理表

创建代理表，然后使用带 SELECT 子句的 INSERT 语句将数据从远程数据库插入到您的数据库的永久表中。

有关远程数据访问的详细信息，请参见“访问远程数据”第 725 页。

有关 INSERT 语句的详细信息，请参见“INSERT 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

处理导入过程中的转换错误

当从外部来源装载数据时，数据中可能有错误。例如，可能存在无效的日期和数字。使用 conversion_error 数据库选项可忽略转换错误并将无效值转换为 NULL 值。

有关设置数据库选项的详细信息，请参见“SET OPTION 语句”一节 《SQL Anywhere 服务器 - SQL 参考》和“conversion_error 选项 [兼容性]”一节 《SQL Anywhere 服务器 - 数据库管理》。

导入表

♦ **导入表**

1. 确保要在其中放置数据的表存在。
2. 在 Interactive SQL 中，从 [**数据**] 菜单中选择 [**导入**]。
3. 选择 [**文本文件中**] 并单击 [**下一步**]。
4. 在 [**文件名**] 字段中，单击 [**浏览**] 添加文件。
5. 选择 [**现有表中**]。
6. 单击 [**下一步**]。
7. 对于 ASCII 文件，指定读取 ASCII 文件的方式并单击 [**下一步**]。
8. 单击 [**导入**]。
9. 单击 [**关闭**]。

♦ **导入表（Interactive SQL [SQL 语句] 窗格）**

1. 使用 CREATE TABLE 语句创建目标表。例如：

```
CREATE TABLE GROUPO.Departments (
DepartmentID          integer NOT NULL,
DepartmentName        char(40) NOT NULL,
DepartmentHeadID      integer NULL,
CONSTRAINT DepartmentsKey PRIMARY KEY (DepartmentID) );
```

2. 执行 LOAD TABLE 语句。例如，

```
LOAD TABLE Departments
FROM 'departments.csv';
```

3. 若要在值中保留尾随空白，请在 LOAD TABLE 语句中使用 STRIP OFF 子句。缺省设置 (STRIP ON) 会在插入值之前去除值中的尾随空白。

   LOAD TABLE 语句将文件的内容添加到表中的现有行；它不替换表中的现有行。可以使用 TRUNCATE TABLE 语句从表中删除所有行。

   TRUNCATE TABLE 语句和 LOAD TABLE 语句都不触发触发器或执行参照完整性操作（例如级联删除）。

另请参见

- “CREATE TABLE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “LOAD TABLE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》

用于导入的表结构

源数据的结构不需要与目标表本身的结构相匹配。例如，列数据类型可以不同或者采用不同的顺序，或者导入数据中可能有与目标表中的列值不匹配的其它值。

重新排列表或数据

如果知道要导入数据的结构与目标表的结构不匹配，则可以：

- 在 LOAD TABLE 语句中提供一个要装载的列名称的列表
- 使用 INSERT 语句的变形形式和一个全局临时表重新排列导入数据以适合目标表的结构。
- 使用 INPUT 语句指定列的特定集合或顺序。

允许列包含 NULL 值

如果正在导入的文件包含表中列的子集的数据，或者如果列的顺序不同，则您还可以使用 LOAD TABLE 语句 DEFAULTS 选项填充空白并合并非匹配表结构。

- 如果 DEFAULTS 为 OFF，则将任何未显示在列列表中且可为空的列置为空值。如果 DEFAULTS 为 OFF 并且列列表中省略了不可为空的列，则数据库服务器会试图将空字符串转换为该列的类型。
- 如果 DEFAULTS 为 ON 并且列具有缺省值，则使用该值。

例如，可以为 Customers 表中的 City 列定义一个缺省值，然后使用 LOAD TABLE 语句从名为 new_customers.txt 的虚构文件中将新行装载到 Customers 表中，如下所示：

```
ALTER TABLE Customers
ALTER City DEFAULT 'Waterloo';
LOAD TABLE Customers ( Surname, GivenName, Street, State, Phone )
FROM 'new_customers.txt'
DEFAULTS ON;
```

由于没有为 City 列提供值，因此提供缺省值。如果早已指定 `DEFAULTS OFF`，则应已为 City 列指派空字符串。

合并不同的表结构

使用 INSERT 语句的变形形式和一个全局临时表重新排列导入数据以适合目标表的结构。

♦ **使用全局临时表装载具有不同结构的数据：**

1. 在 [**SQL 语句**] 窗格中，创建结构与输入文件相匹配的全局临时表。

   您可以使用 CREATE TABLE 语句创建该全局临时表。

2. 使用 LOAD TABLE 语句将数据装载到全局临时表中。

   关闭数据库连接时，全局临时表中的数据就会消失。但是，表定义仍然存在。可以在下次连接数据库时使用它。

3. 使用带有 SELECT 子句的 INSERT 语句从临时表中抽取数据并对其进行汇总，然后将这些数据复制到一个或多个永久数据库表中。

另请参见

- “CREATE TABLE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “LOAD TABLE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》

导入二进制文件

可以使用 xp_read_file 系统过程将二进制文件（例如 JPEG 文件、位图文件或 Microsoft Word 文件）导入数据库。请参见“插入文档和图像”一节第 500 页。

导出数据

导出数据是一项管理任务，包括将数据写出数据库。如果需要共享数据库中的大部分或依据特定的标准抽取数据库的某些部分，您会发现导出数据很有用。使用 SQL Anywhere 可执行以下操作：

- 导出单独的表、查询结果或表模式
- 创建自动执行导出的脚本，以便可以连续导出多个表
- 导出到多种不同的文件格式
- 将数据导出到客户端计算机上的文件
- 使用 BCP FORMAT 子句在 SQL Anywhere 和 Adaptive Server Enterprise 之间导出文件

在导出数据前，请确定目前拥有哪些资源，以及要从数据库导出何种类型的信息。

如果要导出整个数据库，出于性能方面的原因，请卸载数据库而不要导出数据。请参见“重建数据库”一节第 709 页。

另请参见

- “UNLOAD 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “OUTPUT 语句 [Interactive SQL]”一节 《SQL Anywhere 服务器 - SQL 参考》
- “批量操作的性能问题”一节第 678 页
- “使用 OUTPUT 语句输出 NULL”一节第 703 页
- “访问客户端计算机上的数据”一节第 706 页

使用 [导出向导] 导出数据

使用 [**导出向导**] 可将查询结果以特定格式导出到文件或数据库。

♦ 使用 Interactive SQL 导出结果集数据

1. 执行查询。
2. 在 Interactive SQL 中，选择 [**数据**] » [**导出**]。
3. 按照 [**导出向导**] 向导中的说明进行操作。

♦ 使用 Interactive SQL 将结果集数据导出到 UltraLite 数据库

1. 连接到 SQL Anywhere 示例数据库后执行以下查询。

   ```
   SELECT * FROM Employees
   WHERE State = 'GA';
   ```

   该结果集包含所有居住在乔治亚州的雇员的列表。
2. 选择 [**数据**] » [**导出**]。
3. 单击 [**数据库中**]。

4. 在 [**数据库类型**] 列表中，选择 [**UltraLite**]。
5. 在 [**用户 Id**] 字段中键入 **dba**。
6. 在 [**口令**] 字段中键入 **sql**。
7. 单击 [**数据库**] 选项卡。
8. 在 [**数据库文件**] 字段中，键入 *C:\Documents and Settings\All Users\Documents\SQL Anywhere 11\Samples\UltraLite\CustDB\custdb.udb*。
9. 单击 [**下一步**]。
10. 单击 [**新表中**]。
11. 在 [**所有者**] 字段中键入 **dba**。
12. 在 [**表名称**] 字段中，键入 **NewTable**。
13. 单击 [**导出**]。
14. 选择 [**SQL**] » [**上一个 SQL**]。

    由 [**导入向导**] 创建并使用的 IMPORT 语句出现在 [**SQL 语句**] 窗格中：

```
--  Generated by the Export Wizard
OUTPUT USING 'driver=UltraLite 11;UID=dba;PWD=sql;
DBF=C:\Documents and Settings\All Users\Documents\SQL Anywhere 11\Samples
\UltraLite\CustDB\custdb.udb'
INTO "dba"."NewTable"
CREATE TABLE ON
```

使用 OUTPUT 语句导出数据

使用 OUTPUT 语句从数据库中导出查询结果、表或视图。

涉及兼容性问题时，OUTPUT 语句十分有用，因为该语句能够以多种不同的文件格式写出 SELECT 语句的结果集。可以使用缺省输出格式，也可以在每个 OUTPUT 语句上指定文件格式。Interactive SQL 可以执行包含多个 OUTPUT 语句的命令文件。

Interactive SQL 缺省输出格式在 **Interactive SQL [选项]** 窗口（在 Interactive SQL 中选择 [**工具**] » [**选项**] 来访问）的 [**导入/导出**] 选项卡上指定。

可以在以下情况下使用 Interactive SQL OUTPUT 语句：

- 要以非文本格式导出表或视图的全部或一部分
- 希望使用命令文件自动执行导出过程

对数据库的影响

如果可以在使用 OUTPUT 语句、UNLOAD 语句或 UNLOAD TABLE 语句之间进行选择，则出于性能原因，请选择 UNLOAD TABLE 语句。

使用 OUTPUT 语句导出大量的数据会对性能产生影响。如果可能，应该在服务器所在的计算机上使用 OUTPUT 语句，从而避免通过网络发送大量的数据。

有关详细信息，请参见“OUTPUT 语句 [Interactive SQL]”一节 《SQL Anywhere 服务器 - SQL 参考》。

示例

以下示例将数据从 SQL Anywhere 示例数据库中的 Employees 表导出到名为 *Employees.txt* 的 *.txt* 文件中。

```
SELECT *
FROM Employees;
OUTPUT TO Employees.txt
FORMAT TEXT;
```

以下示例将数据从 SQL Anywhere 示例数据库的 Employees 表导出到 SQL Anywhere 数据库中名为 *mydatabase.db* 的新表中

```
SELECT *
FROM Employees;
OUTPUT USING 'driver=SQL Anywhere 11;UID=dba;PWD=sql;DBF=C:\Tobedeleted
\mydatabase.db;CON='''''
INTO "dba"."newcustomers"
CREATE TABLE ON
```

使用 UNLOAD TABLE 语句导出数据

使用 UNLOAD TABLE 语句只能以文本格式高效地导出数据。使用 UNLOAD TABLE 语句导出时，数据库表中的每行在文件中占一行，各个值之间由逗号分隔符隔开。要使重装速度更快，数据应按照主键值的顺序导出。

可以在以下情况下使用 UNLOAD TABLE 语句：

- 要以文本格式导出整个表
- 注重数据库性能
- 将数据导出到客户端计算机上的文件

对数据库的影响

当卸载表时，UNLOAD TABLE 语句在整个表上放置一个独占锁。

如果可以在使用 OUTPUT 语句、UNLOAD 语句或 UNLOAD TABLE 语句之间进行选择，则出于性能原因，请选择 UNLOAD TABLE 语句。

示例

使用 SQL Anywhere 示例数据库，可以通过执行以下命令将 Employees 表卸载到名为 *employee_data.csv* 的文本文件中：

```
UNLOAD TABLE Employees TO 'employee_data.csv';
```

因为是由数据库服务器卸载表，所以 *employee_data.csv* 指定数据库服务器计算机上的文件。

另请参见

- “访问客户端计算机上的数据”一节第 706 页
- “UNLOAD 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “OUTPUT 语句 [Interactive SQL]”一节 《SQL Anywhere 服务器 - SQL 参考》

使用 UNLOAD 语句导出数据

UNLOAD 语句类似于 OUTPUT 语句，因为这两个语句都将查询结果导出到文件中。但 UNLOAD 语句能以文本格式更高效地导出数据。使用 UNLOAD 语句导出时，数据库表中的每行在文件中占一行，各个值之间由逗号分隔符隔开。

可以在以下情况下使用 UNLOAD 语句卸载数据：

- 希望在涉及性能问题时导出查询结果
- 要以文本格式存储输出
- 要将导出命令嵌入应用程序中
- 将数据导出到客户端计算机上的文件

对数据库的影响

如果可以在使用 OUTPUT 语句、UNLOAD 语句或 UNLOAD TABLE 语句之间进行选择，则出于性能原因，请选择 UNLOAD TABLE 语句。

若要使用 UNLOAD 语句，用户必须拥有执行在 UNLOAD 语句中指定的 SELECT 子句所需的权限。

有关控制哪些用户可以使用 UNLOAD 语句的详细信息，请参见“-gl 服务器选项”一节 《SQL Anywhere 服务器 - 数据库管理》。

UNLOAD 语句在当前隔离级别上执行。

示例

使用 SQL Anywhere 示例数据库，可以通过执行以下命令将 Employees 表的子集卸载到名为 *employee_data.csv* 的文本文件中：

```
UNLOAD
SELECT * FROM Employees
WHERE State = 'GA'
TO 'employee_data.csv';
```

因为是由数据库服务器卸载结果集，所以 *employee_data.csv* 指定数据库服务器计算机上的文件。

另请参见

- “访问客户端计算机上的数据”一节第 706 页
- “UNLOAD 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “OUTPUT 语句 [Interactive SQL]”一节 《SQL Anywhere 服务器 - SQL 参考》

使用 dbunload 实用程序导出数据

使用 dbunload 实用程序可导出一个、多个或所有数据库表。可以导出表数据和表模式。要重新排列数据库表，还可以使用 dbunload 实用程序创建必要的命令文件，然后根据需要对它们进行修改。可以只卸载表结构，也可以只卸载表数据，或者既卸载表结构又卸载表数据。

还可以使用或不使用命令文件抽取一个或多个表。这些文件可用于在不同的数据库中创建相同的表。

注意
dbunload 实用程序在功能上与 Sybase Central 的 [**卸载数据库向导**] 等效。可以互换使用两者中的任何一个，它们产生的结果相同。

可以在以下情况下使用 dbunload 实用程序：

- 需要重建或抽取数据库
- 要以文本格式导出数据
- 需要快速处理大量数据
- 要求文件格式灵活

有关 dbunload 实用程序的详细信息，请参见“卸载实用程序 (dbunload)”一节 《SQL Anywhere 服务器 - 数据库管理》。

使用 [卸载数据库向导] 导出数据

使用 [**卸载数据库向导**] 将现有数据库卸载到新数据库中。

使用 [**卸载数据库向导**] 卸载数据库时，可以选择是卸载数据库中的所有对象还是卸载数据库中表的子集。只有在 [**配置所有者过滤**] 窗口中选择的用户的表才会出现在 [**卸载数据库向导**] 中。如果您要查看属于特定数据库用户的表，请右击要卸载的数据库，选择 [**配置所有者过滤**]，然后在出现的窗口中选择用户。

注意
当仅卸载表时，拥有表的用户 ID 不被卸载。重装表之前，必须在新数据库中创建拥有这些表的用户 ID。

也可以使用 [**卸载数据库向导**] 采用逗号分隔的文本格式卸载整个数据库，然后创建必要的 Interactive SQL 命令文件，以完全重建数据库。对于创建 SQL Remote 抽取或建立结构相同或稍有改动的数据库新副本来说，这可能非常有用。如果打算在 SQL Anywhere 中重新使用导出的 SQL Anywhere 文件，[**卸载数据库向导**] 非常有用。

[**卸载数据库向导**] 还会提供相应选项，让您选择重装到现有数据库或新数据库中，而不重装到重装文件中。

dbunload 实用程序在功能上与 [**卸载数据库向导**] 等效。可以互换使用两者中的任何一个，它们产生的结果相同。

注意
如果要卸载的数据库已经在运行并且启动了 [**卸载数据库向导**]，则 SQL Anywhere 插件会在卸载之前自动停止数据库。

有关卸载数据库时的特殊注意事项的信息，请参见“卸载实用程序 (dbunload)”一节 《SQL Anywhere 服务器 - 数据库管理》。

♦ **卸载数据库文件或运行中的数据库 (Sybase Central)**

1. 选择 [**工具**] » [**SQL Anywhere 11**] » [**卸载数据库**]。
2. 请按照 [**卸载数据库向导**] 中的说明进行操作。

使用 [卸载数据] 窗口导出数据

可以使用 Sybase Central 中的 [**卸载数据**] 窗口卸载数据库中的一个或多个表。使用 [**卸载数据库向导**] 或卸载实用程序 (dbunload) 也可获得此功能，但是此窗口允许您一步卸载表而不用完成整个 [**卸载数据库向导**]。

♦ **使用 [卸载数据] 窗口卸载表**

1. 以具有 DBA 权限的用户身份连接到数据库。
2. 双击 [**表**]。
3. 右击要从中导出数据的表，然后选择 [**卸载数据**]。
4. 完成 [**卸载数据**] 窗口。单击 [**确定**]。

导出查询结果

使用 [**数据**] 菜单、OUTPUT 语句 或 UNLOAD 语句可将查询（包括对视图的查询）导出到文件中。

使用 BCP FORMAT 子句可在 SQL Anywhere 和 Adaptive Server Enterprise 之间导入和导出文件。有关详细信息，请参见“Adaptive Server Enterprise 兼容性”一节第 723 页。

♦ **导出查询结果（Interactive SQL [数据] 菜单）**

1. 在 Interactive SQL 的 [**SQL 语句**] 窗格中输入查询。
2. 选择 [**SQL**] » [**执行**]。
3. 选择 [**数据**] » [**导出**]。
4. 指定结果的位置并单击 [**下一步**]。
5. 对于文本、HTML 和 XML 文件，在 [**文件名**] 字段中键入文件名并单击 [**导出**]。

   对于 ODBC 数据库：

a. 选择数据库，然后单击 [**下一步**]。

b. 选择保存数据的位置，然后单击 [**导出**]。

6. 单击 [**关闭**]。

♦ 导出查询结果（Interactive SQL OUTPUT 语句）

1. 在 Interactive SQL 的 [**SQL 语句**] 窗格中输入查询。
2. 查询结束后，键入 **OUTPUT TO '***filename***'**。例如，要将整个 Employees 表导出到文件 *employees.txt* 中，请输入以下查询：

```
SELECT *
FROM Employees;
OUTPUT TO 'employees.txt';
```

3. 要导出查询结果并将结果附加到另一个文件中，请使用 APPEND 子句。

```
SELECT * FROM Employees;
OUTPUT TO 'employees.txt'
APPEND;
```

要导出查询结果并包括消息，请使用 VERBOSE 子句。

```
SELECT * FROM Employees;
OUTPUT TO 'employees.txt'
VERBOSE;
```

4. 选择 [**SQL**] » [**执行**]。

如果导出成功，[**消息**] 选项卡将显示导出查询结果集花费的时间、导出数据的文件名和路径以及写入的行数。如果导出不成功，则将显示一条消息指示导出不成功。

有关使用 OUTPUT 语句导出查询结果的详细信息，请参见“OUTPUT 语句 [Interactive SQL]”一节 《SQL Anywhere 服务器 - SQL 参考》。

提示

您可以结合使用 APPEND 和 VERBOSE 子句将结果和消息追加到现有文件。

例如，键入 **OUTPUT TO '***filename***' APPEND VERBOSE**。

OUTPUT 语句及其 APPEND 和 VERBOSE 子句相当于早期版本 Interactive SQL 中的 >#、>>#、>& 和 >>& 运算符。您仍可以使用这些运算符重定向数据，但是新的 Interactive SQL 语句提供更精确的输出和更便于阅读的代码。

有关 APPEND 和 VERBOSE 的详细信息，请参见“OUTPUT 语句 [Interactive SQL]”一节 《SQL Anywhere 服务器 - SQL 参考》。

♦ 导出查询结果（UNLOAD 语句）：

1. 在 [**SQL 语句**] 窗格中，输入 UNLOAD 语句。例如，

```
UNLOAD
SELECT *
FROM Employees
TO 'employee_data.csv';
```


2. 选择 [**SQL**] » [**执行**]。

   如果导出成功，[**消息**] 选项卡将显示导出查询结果集花费的时间、导出数据的文件名和路径以及写入的行数。如果导出不成功，则将显示一条消息指示导出不成功。

使用 OUTPUT 语句输出 NULL

您可能要抽取数据，然后将这些数据用在其它软件产品中。由于其它软件产品可能不理解 NULL 值，所以提供两种方法指定在 Interactive SQL 中使用 OUTPUT 语句时如何显示 NULL 值：

- 使用 output_nulls 选项可指定由 OUTPUT 语句使用的输出值
- 使用 IFNULL 函数可将输出值应用到特定实例或查询

这两个选项都可用于输出特定值代替 NULL 值。通过指定如何输出 NULL 值，提高了与其它软件产品的兼容性。

♦ 指定 NULL 值输出 (Interactive SQL)

- 执行 SET OPTION 语句，更改 output_nulls 选项的值。下面的示例将 NULL 值的显示值更改为 (unknown)：

```
SET OPTION output_nulls = '(unknown)';
```

有关设置 Interactive SQL 选项的详细信息，请参见“SET OPTION 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

♦ 在 [结果] 窗格 (Interactive SQL) 中更改代替 NULL 值显示的值

1. 选择 [**工具**] » [**选项**]。
2. 单击 [**SQL Anywhere**]。
3. 单击 [**结果**] 选项卡。
4. 在 [**将空值显示为**] 字段中，键入 **Value**。
5. 单击 [**确定**]。

导出数据库

注意
如果要卸载的数据库已经在运行并且启动了 [**卸载数据库向导**]，则 SQL Anywhere 插件会在卸载之前自动停止数据库。

♦ **卸载整个或部分数据库 (Sybase Central)：**

1. 选择 [**工具**] » [**SQL Anywhere 11**] » [**卸载数据库**]。
2. 请按照 [**卸载数据库向导**] 中的说明进行操作。

♦ **卸载整个或部分数据库（命令行）：**

● 运行 dbunload 实用程序，使用 -c 选项指定连接参数。

将整个数据库卸载到服务器计算机的目录 *c:\DataFiles* 中：

```
dbunload -c "DBN=demo;UID=DBA;PWD=sql" c:\DataFiles
```

重新创建模式和重装表所需的语句写入本地当前目录中的 *reload.sql*。

若要只导出数据，请使用 -d。例如：

```
dbunload -c "DBN=demo;UID=DBA;PWD=sql" -d c:\DataFiles
```

重装表所需的语句写入本地当前目录中的 *reload.sql*。

若要只导出模式，请使用 -n。例如：

```
dbunload -c "DBN=demo;UID=DBA;PWD=sql" -n
```

重新创建模式所需的语句写入本地当前目录中的 *reload.sql*。

有关详细信息，请参见“卸载实用程序 (dbunload)”一节 《SQL Anywhere 服务器 - 数据库管理》。

导出表

还可以通过选择表中的所有数据并导出查询结果来导出表。请参见“导出查询结果”一节 第 701 页。

使用相同的过程来导出视图。

♦ **导出表（命令行）：**

● 运行以下命令：

```
dbunload -c "DBN=demo;UID=DBA;PWD=sql"
-t Employees c:\DataFiles
```

在此命令中，-c 指定数据库连接参数，-t 指定要导出的一个或多个表的名称。此 dbunload 命令将数据从 SQL Anywhere 示例数据库（假设以缺省数据库名在缺省数据库服务器上运行）卸载到服务器计算机上 *c:\DataFiles* 目录中的一组文件中。从数据文件重建表的命令文件是在本地当前目录中以缺省名称 *reload.sql* 创建的。

通过用逗号 (,) 分隔符分隔表名，您可以卸载多个表。

♦ **导出表 (SQL)：**

- 执行 UNLOAD TABLE 语句。例如，

```
UNLOAD TABLE Departments
TO 'departments.csv';
```

此语句将 Departments 表从 SQL Anywhere 示例数据库卸载到数据库服务器上当前工作目录中的 *departments.csv* 文件中。如果是在网络数据库服务器上运行，则该命令将数据卸载到服务器计算机（而不是客户端计算机）上的某个文件中。另外，将文件名作为一个字符串传递给服务器。如果目录或文件名以字母 n（\n 是换行符）或任何其它特殊字符开头，则在文件名中使用转义反斜线字符可以防止对其进行错误解释。

表的每一行都输出到输出文件的单个行上，不导出任何列名。各列之间以逗号分隔。可以使用 DELIMITED BY 子句更改分隔符。字段不是固定宽度的字段。只导出每个条目中的字符，而不是列的整个宽度。

另请参见

- “导出查询结果”一节第 701 页
- “卸载实用程序 (dbunload)”一节 《SQL Anywhere 服务器 - 数据库管理》
- “UNLOAD 语句”一节 《SQL Anywhere 服务器 - SQL 参考》

访问客户端计算机上的数据

SQL Anywhere 允许使用 SQL 语句和函数从客户端计算机上的文件中装载数据以及将数据卸载到其中，而无需将文件复制到数据库服务器计算机。为此，数据库服务器使用命令序列通讯协议 (CmdSeq) 文件处理程序启动传送。数据库服务器接收到来自客户端应用程序的请求（即需要将数据传入或传出客户端计算机）后，在发送响应之前，调用 CmdSeq 文件处理程序。文件处理程序支持随时从客户端同时和交替传送多个文件。例如，当客户端应用程序执行的语句需要时，数据库服务器启动同时传送多个文件。

使用 CmdSeq 文件处理程序实现客户端数据的传送意味着应用程序不需要任何新的专用代码，并且可以使用下面列出的 SQL 组件立即从功能中受益：

- **READ_CLIENT_FILE 函数** READ_CLIENT_FILE 函数从客户端计算机上的指定文件中读取数据，并返回表示文件内容的 LONG BINARY 值。此函数可用于 SQL 代码中可使用 BLOB 的任何地方。READ_CLIENT_FILE 函数返回的数据尽可能不在内存中进行实例化，除非语句显式导致实例化发生。例如，LOAD TABLE 语句从客户端文件中流出数据，而不对其进行实例化。将 READ_CLIENT_FILE 函数返回的值分配到连接变量导致数据库服务器检索和实例化客户端文件的内容。请参见“READ_CLIENT_FILE 函数 [String]”一节 《SQL Anywhere 服务器 - SQL 参考》。
- **WRITE_CLIENT_FILE 函数** WRITE_CLIENT_FILE 函数将数据写入客户端计算机上的指定文件中。请参见“WRITE_CLIENT_FILE 函数 [String]”一节 《SQL Anywhere 服务器 - SQL 参考》。
- **READCLIENTFILE 权限** READCLIENTFILE 权限允许从客户端计算机上的文件中读取。请参见“READCLIENTFILE 特权”一节 《SQL Anywhere 服务器 - 数据库管理》。
- **WRITECLIENTFILE 权限** WRITECLIENTFILE 权限允许向客户端计算机上的文件中写入。请参见“WRITECLIENTFILE 特权”一节 《SQL Anywhere 服务器 - 数据库管理》。
- **LOAD TABLE ...USING CLIENT FILE 子句** USING CLIENT FILE 子句允许使用位于客户端计算机上的文件中的数据装载表。例如，`LOAD TABLE ... USING CLIENT FILE 'my-file.txt';` 会从客户端计算机装载名为 *my-file.txt* 的文件。请参见“LOAD TABLE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。
- **LOAD TABLE ...USING VALUE 子句** USING VALUE 子句允许将 BLOB 表达式指定为一个值。BLOB 表达式可使用 READ_CLIENT_FILE 函数从客户端计算机上的文件中装载 BLOB。例如，`LOAD TABLE ... USING VALUE READ_CLIENT_FILE( 'my-file' )`，其中 *my-file* 是客户端计算机上的文件。请参见“LOAD TABLE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。
- **UNLOAD TABLE ...INTO CLIENT FILE 子句** INTO CLIENT FILE 子句允许指定客户端计算机上存放所卸载数据的文件。请参见“UNLOAD 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。
- **UNLOAD TABLE ...INTO VARIABLE 子句** INTO VARIABLE 子句允许指定将数据卸载到其中的变量。请参见“UNLOAD 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。
- **read_client_file 和 write_client_file 安全功能** read_client_file 和 write_client_file 安全功能控制使用某些语句，这些语句会导致从客户端文件中读取或向其写入。请参见“指定受保护的功能”

一节 《SQL Anywhere 服务器 - 数据库管理》和“-sf 服务器选项”一节 《SQL Anywhere 服务器 - 数据库管理》。

客户端数据安全性

SQL Anywhere 提供一些方法来确保传送客户端文件不会导致未授权即传送客户端计算机上的数据，而客户端计算机通过处于与数据库服务器计算机不同的位置。

为此，数据库服务器跟踪每个已执行语句的源，并确定语句是否是从客户端应用程序直接接收的。从客户机启动传送新文件时，数据库服务器会包括语句源的信息。然后 CmdSeq 文件处理程序允许传送客户端应用程序直接发送的语句的文件。如果语句未经客户端应用程序直接发送，则应用程序必须注册一个验证回调。如果未注册任何回调，则该传送被拒绝且语句失败并出现错误。

同样，直到成功建立连接后才允许传送客户端数据。此限制会防止使用连接字符串或登录过程进行未授权访问。

要防止伪装成授权用户的用户获得系统访问权限，请考虑将所传送的数据加密。

SQL Anywhere 还提供以下安全机制来控制各个级别的访问：

- **服务器级别安全性** read_client_file 和 write_client_file 受保护的功能允许禁用整个服务器范围的所有客户端传送。请参见“-sf 服务器选项”一节 《SQL Anywhere 服务器 - 数据库管理》。
- **应用程序和 DBA 级别安全性** allow_read_client_file 和 allow_write_client_file 数据库选项提供在数据库、用户或连接级别上的访问控制。例如，连接后应用程序可将此数据库选项设置为 OFF 以防止自身用于任何客户端传送。请参见“allow_read_client_file 选项 [数据库]”一节 《SQL Anywhere 服务器 - 数据库管理》。
- **用户级别安全性** READCLIENTFILE 和 WRITECLIENTFILE 权限分别为从客户端计算机读取数据和将数据写入客户端计算机提供用户级别访问控制。请参见“READCLIENTFILE 特权”一节 《SQL Anywhere 服务器 - 数据库管理》和“WRITECLIENTFILE 特权”一节 《SQL Anywhere 服务器 - 数据库管理》。

装载客户端数据时规划恢复

如果需要从事务日志中恢复 LOAD TABLE 语句，则客户端计算机上用于装载数据的文件可能不再可用于 SQL Anywhere，或者已经更改，因此原始数据不再可用。要防止这种情况发生，请确保不关闭记录。然后，装载数据时指定 WITH ROW LOGGING 或 WITH CONTENT LOGGING 子句中的任意一个。这些子句导致正在装载的数据被记录在事务日志中，以便以后恢复时可重新使用。

WITH ROW LOGGING 子句导致每个插入的行被记录为事务日志中的一个 INSERT 语句。WITH CONTENT LOGGING 导致插入的数据按块记录在事务日志中，以用于恢复过程中由数据库服务器进行处理。两种方法都适用于确保客户端数据可用于在恢复过程中进行装载。但是，将数据装载到参与同步的数据库时无法使用 WITH CONTENT LOGGING。

指定以下 LOAD TABLE 语句的任何一个但不指定记录级别时，WITH CONTENT LOGGING 为缺省行为：

- LOAD TABLE ...USING CLIENT FILE *client-filename-expression*

- LOAD TABLE ...USING VALUE *value-expression*
- LOAD TABLE ...USING COLUMN *column-expression*

有关装载操作过程中如何在事务日志中记录所装载数据的详细信息，请参见“LOAD TABLE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

重建数据库

重建数据库是一种特殊类型的导入和导出，涉及卸载和重装整个数据库。重建（卸载/装载）和抽取过程用于重建数据库、从现有数据库的一部分创建新数据库以及消除未使用的空闲页。

如果要重建数据库，使其升级为更新版本的 SQL Anywhere，请参见“升级 SQL Anywhere”一节《SQL Anywhere 11 - 更改和升级》。

可以从 Sybase Central 或使用 dbunload 实用程序重建数据库。

注意
最好在重建数据库前对其进行备份，特别是选择使用重建的数据库替换原始数据库时。

有关详细信息，请参见“备份和数据恢复” 《SQL Anywhere 服务器 - 数据库管理》。

导入和导出的结果是数据进出数据库。导入将数据读入数据库。导出将数据写出数据库。信息通常以另一个非 SQL Anywhere 数据库为来源或目的地。

如果指定了加密选项 -ek、-ep 或 -et，*reload.sql* 文件中的 LOAD TABLE 语句必须包括加密密钥。硬编码密钥会危及安全性，所以 *reload.sql* 文件中的一个参数指定加密密钥。使用 Interactive SQL 执行 *reload.sql* 文件时，必须将加密密钥指定为参数。如果未在 READ 语句中指定密钥，Interactive SQL 会提示您输入密钥。请参见“Interactive SQL 实用程序 (dbisql)”一节 《SQL Anywhere 服务器 - 数据库管理》。

装载和卸载操作会从 SQL Anywhere 数据库中提取数据和模式，然后将数据和模式放回 SQL Anywhere 数据库。卸载过程产生数据文件和一个 *reload.sql* 文件，后者包含重新创建完全相同的表所需的表定义。运行 *reload.sql* 脚本可重新创建这些表并将数据装载回这些表。

重建数据库是一项耗时的操作，并需要大量磁盘空间。同样，当卸载和重装数据库时，数据库无法使用。出于这些原因，除非目的明确，否则不建议在生产环境中重建数据库。

从一个 SQL Anywhere 数据库到另一个

重建通常将数据从 SQL Anywhere 数据库中复制出来，然后将这些数据重装回 SQL Anywhere 数据库。卸载和重装是关联的，因为通常同时执行这两个任务，而不是只执行其中的一个。

重建与导出

重建不同于导出，其不同之处在于重建过程除了导出并导入数据之外，还导出并导入表定义和模式。重建过程的卸载部分产生文本格式数据文件和包含表定义和其它定义的 *reload.sql* 文件。可以运行 *reload.sql* 脚本重新创建这些表并将数据装载到这些表中。

有关详细信息，请参见“内部与外部卸载和重装”一节 《SQL Anywhere 服务器 - 数据库管理》。

如果使用 SQL Remote 或 MobiLink，则考虑抽取数据库（从旧数据库创建新数据库）。请参见“抽取数据”一节第 716 页。

重建复制数据库

重建数据库的过程取决于该数据库是否参与复制。如果数据库参与复制，则必须在整个操作过程中保留事务日志偏移，因为 Message Agent 和 Replication Agent 需要此信息。如果数据库不参与复制，则重建过程会简单一些。

另请参见

- “重建数据库时最小化停机时间”一节第 714 页
- “重建参与同步或复制的数据库”一节第 711 页
- “重建不参与同步或复制的数据库”一节第 711 页
- “将数据库从一种归类更改为另一种归类”一节 《SQL Anywhere 服务器 - 数据库管理》
- “刷新手动视图”一节第 56 页

重建数据库的原因

有几个原因需要考虑重建数据库。如果要实现下面的目标，最好重建数据库：

- **升级数据库文件格式** 通过应用升级实用程序可以实现某些新功能，但是其它功能需要数据库文件格式升级，这是通过卸载和重装数据库完成的。要确定获得新功能是否需要卸载和重装，请参见“升级到 SQL Anywhere 11” 《SQL Anywhere 11 - 更改和升级》。

  无需升级数据库就可以使用新版本的 SQL Anywhere 数据库服务器。如果要使用需具备新系统表或数据库选项的访问权限才能使用的新版本的功能，则必须使用升级实用程序升级数据库。升级实用程序不卸载或重装任何数据。

  如果要使用依赖于数据库文件格式更改的 SQL Anywhere 新版本，则必须卸载和重装数据库。在重建数据库前应对该数据库进行备份。

  注意
  如果从版本 9 或更早版本升级，则必须重建数据库文件。如果从版本 10.0.0 及更高版本升级，可以使用升级实用程序或重建数据库。

  有关升级数据库的详细信息，请参见“升级 SQL Anywhere”一节 《SQL Anywhere 11 - 更改和升级》。

  有关在数据库镜像系统中涉及的升级 SQL Anywhere 或重建数据库的信息，请参见“在数据库镜像系统中升级 SQL Anywhere 软件和数据库”一节 《SQL Anywhere 11 - 更改和升级》。

- **回收磁盘空间** 如果删除数据，数据库不会缩小。而只是将所有空页标记为可用，以便可以重新使用这些页。除非重建数据库，否则不会从数据库中删除它们。如果从数据库中删除了大量的数据并且预期不会添加更多的数据，则重建数据库可以回收磁盘空间。
- **提高数据库性能** 重建数据库可提高性能。由于可以按照主键的顺序卸载和重装数据库，因此相关的行可能出现在相同或相邻的页上，从而使对相关信息的访问速度更快。

注意
如果检测到性能因为表碎片过多而低下，则可以重组表。请参见“REORGANIZE TABLE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

另请参见

- “升级实用程序 (dbupgrad)”一节 《SQL Anywhere 服务器 - 数据库管理》
- “卸载实用程序 (dbunload)”一节 《SQL Anywhere 服务器 - 数据库管理》

重建不参与同步或复制的数据库

只应在数据库不参与同步或复制时才使用下面的过程。

♦ **重建不参与同步或复制的数据库（命令行）**

1. 运行 dbunload 实用程序，指定以下选项之一：

要实现这一目的……	请使用此选项……	示例
重建为新数据库	-an	`dbunload -c "DBF=demo.db;UID=DBA;PWD=sql" -an DemoBackup.db`
重装到现有数据库	-ac	`dbunload -c "DBF=demo.db;UID=DBA;PWD=sql" -ac "UID=DBA;PWD=sql;DBF=NewDemo.db"`
替换现有数据库	-ar	`dbunload -c "DBF=demo.db;UID=DBA;PWD=sql" -ar`

   如果使用这些选项中的一个，则不会在磁盘上创建数据的中间副本，因此无需在命令行中指定卸载目录。这为数据提供了更高的安全性。使用 -ar 和 -an 选项时执行速度要比 Sybase Central 中的 [**卸载数据库向导**] 快，但使用 -ac 时执行速度比 [**卸载数据库向导**] 慢。

2. 先关闭数据库并将事务日志存档，然后再使用重装的数据库。

注意

-an 和 -ar 选项只适用于与个人服务器的连接或通过共享内存与网络服务器的连接。

dbunload 实用程序还提供了一些其它选项（可用来调整卸载）和连接参数选项（可用来指定运行中或非运行中数据库以及数据库参数）。

重建参与同步或复制的数据库

本节内容适用于 SQL Anywhere MobiLink 客户端（使用 dbmlsync 的客户端）、SQL Remote 和复制代理。

如果数据库正在参与同步或复制，则重建该数据库需要特别注意。同步和复制是基于事务日志中的偏移进行的。当重建数据库时，旧事务日志中的偏移不同于新事务日志中的偏移，从而使旧日志不可用。因此，良好的备份习惯在数据库参与同步或复制时尤为重要。

重建参与同步或复制的数据库有两种方法。第一种方法使用 dbunload 实用程序 -ar 选项，以一种不干扰同步或复制的方式执行卸载和重装。第二种方法是一种可执行相同任务的手动方法。

在重建参与 MobiLink 同步的数据库之前，必须同步所有预订。

♦ 重建参与同步或复制的数据库（dbunload 实用程序）

1. 关闭数据库。
2. 将数据库和事务日志文件复制到一个安全位置，从而执行完全脱机备份。
3. 运行以下 dbunload 命令重建数据库：

   ```
   dbunload -c connection-string -ar directory
   ```

   connection-string 是拥有 DBA 权限的连接，*directory* 是在复制环境中用于旧事务日志的目录。该数据库不能有任何其它连接。

   -ar 选项只适用于与个人服务器的连接或通过共享内存与网络服务器的连接。

   有关详细信息，请参见“卸载实用程序 (dbunload)”一节 《SQL Anywhere 服务器 - 数据库管理》。
4. 关闭新数据库，然后执行通常在恢复数据库之后执行的有效性检查。

   有关有效性检查的详细信息，请参见“校验数据库”一节 《SQL Anywhere 服务器 - 数据库管理》。
5. 使用任何需要的生产选项启动数据库。现在可以允许用户访问重装的数据库。

注意

dbunload 实用程序还提供了一些其它选项（可用来调整卸载）和连接参数选项（可用来指定运行中或非运行中数据库以及数据库参数）。请参见“卸载实用程序 (dbunload)”一节 《SQL Anywhere 服务器 - 数据库管理》。

如果以上过程不能满足您的需要，可以手工调整事务日志偏移。下面的过程介绍如何执行此操作。

♦ 以手工干预的方式重建参与同步或复制的数据库

1. 关闭数据库。
2. 将数据库和事务日志文件复制到一个安全位置，从而执行完全脱机备份。
3. 运行 dbtran 实用程序，以显示数据库当前事务日志文件的开始偏移和结束偏移。

   记下结束偏移，以便在第 8 步中使用。
4. 重命名当前事务日志文件，以便在卸载过程中该文件不被修改，然后将此文件放置在 dbremote 脱机日志目录中。
5. 重建数据库。

   有关此步骤的信息，请参见“重建数据库”一节第 709 页。
6. 关闭新数据库。
7. 消除新数据库的当前事务日志文件。
8. 对新数据库使用 dblog，将第 3 步中记录的结束偏移用作 -z 参数，同时将相对偏移设置为零。


```
dblog -x 0 -z 0000698242 -il -ir -is database-name.db
```

9. 在运行消息代理时，在它的命令行上为它提供原始脱机目录的位置。

10. 启动数据库。现在可以允许用户访问重装的数据库。

使用 dbunload 实用程序重建数据库

使用 dbunload 和 dbisql 实用程序可以采用逗号分隔的文本格式卸载整个数据库，然后创建必要的 Interactive SQL 命令文件，以完全重建数据库。对于创建 SQL Remote 抽取或建立结构相同或稍有变动的数据库新副本来说，这可能非常有用。如果打算在 SQL Anywhere 中重新使用导出的 SQL Anywhere 文件，此实用程序非常有用。

注意
dbunload 实用程序在功能上与 [**卸载数据库向导**] 等效。可以交替使用两者，它们产生的结果相同。

可以在以下情况下使用 dbunload 实用程序：

- 要重建数据库或从数据库抽取数据
- 要以文本格式导出数据
- 需要快速处理大量数据
- 要求文件格式灵活

有关详细信息，请参见：

- “重建不参与同步或复制的数据库”一节第 711 页
- “重建参与同步或复制的数据库”一节第 711 页

使用 UNLOAD TABLE 语句重建数据库

使用 UNLOAD TABLE 语句可以采用特殊字符编码形式高效地导出数据。若要以文本格式导出数据，请考虑使用 UNLOAD TABLE 语句重建数据库。

对数据库的影响

UNLOAD TABLE 语句在整个表上放置一个独占锁。

有关详细信息，请参见“UNLOAD 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

导出表数据或表模式

卸载实用程序提供的选项可用来仅卸载表数据或者仅卸载表模式。

这些示例中的 dbunload 命令将数据或模式从 SQL Anywhere 示例数据库表（假设以缺省数据库名在缺省数据库服务器上运行）卸载到服务器计算机上 *c:\DataFiles* 目录下的文件中。重新创建模式和重装指定表所需的语句写入本地当前目录中的 *reload.sql*。

♦ **导出表数据（命令行）**

- 运行 dbunload 命令，指定：使用 -c 选项的连接参数，想要使用 -t 选项导出数据的表格以及是否想通过指定 -d 选项仅卸载数据。

  例如，若要从 Employees 表导出数据，运行以下命令：

  ```
  dbunload -c "DBN=demo;UID=DBA;PWD=sql" -d -t Employees c:\DataFiles
  ```

  通过用逗号分隔符分隔表名，您可以卸载多个表。

♦ **导出表模式（命令行）**

- 运行 dbunload 命令，指定：使用 -c 选项的连接参数，想要使用 -t 选项导出数据的表格以及是否想通过指定 -n 选项仅卸载模式。

  例如，若要从 Employees 表导出模式，请执行下面的命令：

  ```
  dbunload -c "DBN=demo;UID=DBA;PWD=sql" -n -t Employees
  ```

  通过用逗号分隔符分隔表名，您可以卸载多个表。

重装数据库

重装包括创建一个空数据库文件，并使用 *reload.sql* 文件创建模式，然后将从另一个 SQL Anywhere 数据库卸载的所有数据插入新创建的表中。从命令行重装数据库。

♦ **重装数据库（命令行）：**

1. 运行 dbinit 实用程序创建新的空数据库文件。

   例如，以下命令创建名为 *newdemo.db* 的文件。

   ```
   dbinit newdemo.db
   ```

2. 执行 *reload.sql* 脚本。

   例如，以下命令装载并运行当前目录中的 *reload.sql* 脚本。

   ```
   dbisql -c "DBF=newdemo.db;UID=DBA;PWD=sql" reload.sql
   ```

重建数据库时最小化停机时间

以下步骤帮助您在尽量缩短停机时间的同时重建数据库。如果您的数据库全天运转，则这些步骤可能特别有用。

最好在开始实际重建之前练习运行步骤 1-4，以确定每步所需的时间。在重建过程中可能还需要在不同时间点保存文件的副本。

小心
确保没有其它调度的备份重命名生产数据库的日志。如果错误地出现了重命名的情况，则需要按照正确的顺序将这些重命名日志中的事务应用到重建的数据库。

♦ **重建过程中尽量缩短停机时间：**

1. 使用 dbbackup -r 创建数据库和日志的备份，然后重命名日志。

   有关详细信息，请参见“备份实用程序 (dbbackup)”一节 《SQL Anywhere 服务器 - 数据库管理》。
2. 在另一台计算机上重建备份的数据库。
3. 在生产服务器上再执行一次 dbbackup -r 以重命名事务日志。
4. 针对事务日志运行 dbtran，然后将事务应用于重建的服务器。

   有关详细信息，请参见“翻译日志文件实用程序 (dbtran)”一节 《SQL Anywhere 服务器 - 数据库管理》。

   至此，重建的数据库中包含截至第 3 步中的备份结束后的所有事务。
5. 关闭生产服务器并制作数据库和日志的副本。
6. 在生产服务器上复制重建的数据库。
7. 针对第 5 步中产生的日志运行 dbtran。

   这应该是一个相对较小的文件。
8. 在重建的数据库上启动服务器，但不允许用户进行连接。
9. 应用执行第 8 步后产生的事务。
10. 允许用户进行连接。

抽取数据

SQL Remote 使用数据库抽取功能。抽取过程会从 SQL Anywhere 统一数据库创建 SQL Anywhere 远程数据库。

可以使用 Sybase Central [**抽取数据库向导**] 或抽取实用程序来抽取数据库。建议使用抽取实用程序 (dbxtract) 从统一数据库创建远程数据库，以用于 SQL Remote 复制。

有关如何执行数据库抽取的更多信息，请参见：

- “抽取实用程序 (dbxtract)”一节 《SQL Remote》
- “抽取远程数据库”一节 《SQL Remote》
- “部署远程数据库”一节 《MobiLink - 客户端管理》

将数据库迁移到 SQL Anywhere

使用 sa_migrate 系统过程或 [**迁移数据库向导**] 可从以下源导入表：

- SQL Anywhere
- UltraLite
- Sybase ASE
- IBM DB/2
- Microsoft SQL Server
- Microsoft Access
- Oracle
- MySQL
- Advantage Database Server
- 连接到远程服务器的通用 ODBC 驱动程序

使用 [**迁移数据库向导**] 或 sa_migrate 系统过程集迁移数据之前，必须首先创建 **target database**。目标数据库是数据要迁移到的数据库。

有关创建数据库的信息，请参见“创建数据库”一节 《SQL Anywhere 服务器 - 数据库管理》。

使用 [迁移数据库向导]

使用 [**迁移数据库向导**] 可以创建用于连接远程数据库的远程服务器，还可以创建用于将当前用户连接到远程数据库的外部登录（如果需要）。

♦ **导入远程表 (Sybase Central)：**

1. 以具有 DBA 权限的用户身份连接到数据库。
2. 选择 [**工具**] » [**SQL Anywhere 11**] » [**迁移数据库**]。
3. 单击 [**下一步**]。
4. 选择目标数据库，然后单击 [**下一步**]。
5. 选择要用来连接远程数据库的远程服务器，然后单击 [**下一步**]。

   如果未创建远程服务器，则单击 [**立即创建远程服务器**] 并按照 [**创建远程服务器向导**] 中的说明进行操作。有关创建远程服务器的详细信息，请参见“使用 CREATE SERVER 语句创建远程服务器”一节第 730 页。

   还可以为远程服务器创建外部登录。缺省情况下，SQL Anywhere 在代表当前用户连接远程服务器时使用该用户的用户 ID 和口令。但是，如果远程服务器没有定义与当前用户具有相同用户 ID 和口令的用户，则您必须创建一个外部登录。外部登录为当前用户指派一个替代登录名和口令，以使该用户能够连接到远程服务器。
6. 选择要迁移的表，然后单击 [**下一步**]。

   您无法迁移系统表，所以此列表中不会出现任何系统表。
7. 选择在目标数据库上将拥有表的用户，然后单击 [**下一步**]。

如果未创建用户，则单击 [**立即创建用户**] 并按照 [**创建用户向导**] 中的说明进行操作。有关详细信息，请参见“创建新用户”一节 《SQL Anywhere 服务器 - 数据库管理》。

8. 选择是否要从远程表迁移数据和/或外键以及是否要保留为迁移过程创建的代理表，然后单击 [**下一步**]。
9. 单击 [**完成**]。

使用 sa_migrate 系统过程

使用 sa_migrate 系统过程可迁移远程数据。如果要删除表或外键映射，则可使用扩展方法。

使用 sa_migrate 系统过程迁移所有表

如果同时为 *table-name* 和 *owner-name* 参数提供 NULL 值，则会迁移数据库中的所有表，包括系统表。远程数据库中属于不同用户的同名的表在目标数据库中都属于同一个所有者。因此，应该一次只迁移与一个所有者关联的表。

♦ 迁移某个远程用户的所有表

1. 创建目标数据库。请参见“创建数据库”一节 《SQL Anywhere 服务器 - 数据库管理》。
2. 在 Interactive SQL 中，连接到目标数据库。
3. 创建一个远程服务器以连接到远程数据库。请参见“使用 CREATE SERVER 语句创建远程服务器”一节第 730 页。
4. 创建用于连接远程数据库的外部登录。只有当用户在目标数据库和远程数据库上具有不同的口令，或者当您在远程数据库上登录时使用的用户 ID 与在目标数据库上登录时使用的用户 ID 不同时，才需要创建外部登录。请参见“创建外部登录”一节第 738 页。
5. 创建本地用户，该用户将拥有目标数据库中迁移的表。请参见“创建新用户”一节 《SQL Anywhere 服务器 - 数据库管理》。
6. 在 [**SQL 语句**] 窗格中，运行 sa_migrate 系统过程。例如，

```
CALL sa_migrate( 'local_user1', 'rmt_server1', NULL, 'remote_user1', NULL,
1, 1, 1 );
```

此过程使用指定条件依次调用若干过程，并迁移属于用户 remote_user1 的所有远程表。

如果不想让目标数据库上的同一个用户拥有所有迁移的表，则必须通过指定 *local-table-owner* 和 *owner-name* 参数为目标数据库上的每位所有者运行一次 sa_migrate 过程。

有关详细信息，请参见“sa_migrate 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》。

使用 sa_migrate 系统过程迁移各个表

不要为 *table-name* 和 *owner-name* 参数都提供 NULL 值。这将迁移数据库中的所有表，包括系统表。远程数据库中属于不同用户的同名的表在目标数据库中都属于同一个所有者。建议您一次迁移与一个所有者关联的表。

♦ **导入远程表（具有修改）：**

1. 创建目标数据库。请参见“创建数据库”一节 《SQL Anywhere 服务器 - 数据库管理》。
2. 在 Interactive SQL 中，连接到目标数据库。
3. 创建一个远程服务器以连接到远程数据库。请参见“使用 CREATE SERVER 语句创建远程服务器”一节第 730 页。
4. 创建用于连接远程数据库的外部登录。只有当用户在目标数据库和远程数据库上具有不同的口令，或者当您在远程数据库上登录时使用的用户 ID 与在目标数据库上登录时使用的用户 ID 不同时，才需要创建外部登录。请参见“创建外部登录”一节第 738 页。
5. 创建本地用户，该用户将拥有目标数据库中迁移的表。请参见“创建新用户”一节 《SQL Anywhere 服务器 - 数据库管理》。
6. 运行 sa_migrate_create_remote_table_list 系统过程。例如，

   ```
   CALL sa_migrate_create_remote_table_list( 'rmt_server1',
        NULL, 'remote_user1', 'mydb' );
   ```

   必须为 Adaptive Server Enterprise 和 Microsoft SQL Server 数据库指定数据库名。

   此过程使用要迁移的远程表列表填充 dbo.migrate_remote_table_list 表。可以从此表中删除不想迁移的远程表的行。

   有关详细信息，请参见“sa_migrate_create_remote_table_list 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》。
7. 运行 sa_migrate_create_tables 系统过程。例如：

   ```
   CALL sa_migrate_create_tables( 'local_user1' );
   ```

   此过程从 dbo.migrate_remote_table_list 中提取远程表列表，然后为列出的每个远程表创建代理表和基表。此过程还为迁移的表创建所有主键索引。

   有关详细信息，请参见“sa_migrate_create_tables 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》。
8. 如果要将数据从远程表迁移到目标数据库上的基表中，可运行 sa_migrate_data 系统过程。例如，

   执行以下语句：

   ```
   CALL sa_migrate_data( 'local_user1' );
   ```

   此过程将数据从每个远程表迁移到由 sa_migrate_create_tables 过程创建的基表中。

   有关详细信息，请参见“sa_migrate_data 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》。

   如果不想从远程数据库中迁移外键，则可以跳至第 10 步。
9. 运行 sa_migrate_create_remote_fks_list 系统过程。例如，

   ```
   CALL sa_migrate_create_remote_fks_list( 'rmt_server1' );
   ```

   此过程使用与 dbo.migrate_remote_table_list 中列出的每个远程表关联的外键列表填充表 dbo.migrate_remote_fks_list。

您可以删除不想在本地基表上重新创建的任何外键映射。

有关详细信息，请参见“sa_migrate_create_remote_fks_list 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》。

10. 运行 sa_migrate_create_fks 系统过程。例如，

```
CALL sa_migrate_create_fks( 'local_user1' );
```

此过程在基表上创建在 dbo.migrate_remote_fks_list 中定义的外键映射。

有关详细信息，请参见“sa_migrate_create_fks 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》。

11. 如果要删除为执行迁移而创建的代理表，可运行 sa_migrate_drop_proxy_tables 系统过程。例如，

```
CALL sa_migrate_drop_proxy_tables( 'local_user1' );
```

此过程删除为迁移创建的所有代理表，整个迁移过程到此完成。

有关详细信息，请参见“sa_migrate_drop_proxy_tables 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》。

使用 SQL 命令文件

本节介绍如何处理包含一组命令的文件。**Command files** 是包含 SQL 语句的文本文件，如果您想重复运行同一个 SQL 语句，此文件将会非常有用。

创建命令文件

可以使用任何喜欢的文本编辑器来创建命令文件。可以将注释行与要执行的 SQL 语句包括在一起。命令文件通常也称为 **scripts**。请参见“注释”一节 《SQL Anywhere 服务器 - SQL 参考》。

在 Interactive SQL 中打开 SQL 命令文件

在 Windows 操作系统中，可将 Interactive SQL 作为 *.sql* 文件的缺省编辑器。当双击文件时，文件内容会出现在 Interactive SQL 的 [**SQL 语句**] 窗格中。

有关详细信息，请参见“将 Interactive SQL 设置为 .sql 文件的缺省编辑器”一节 《SQL Anywhere 服务器 - 数据库管理》。

在 Interactive SQL 中运行 SQL 命令文件

您可以通过以下任意方式执行命令文件：

- 可以在不将命令文件装载到 [**SQL 语句**] 窗格中的情况下运行命令文件。

  - **直接运行命令文件：**

  1. 在 Interactive SQL 中，选择 [**文件**] » [**运行脚本**]。
  2. 定位该文件，然后单击 [**打开**]。

     指定文件的内容立即运行。即会出现 [**状态**] 窗口并显示执行进度。

     [**运行脚本**] 菜单项等同于 READ 语句。有关 READ 语句的示例，请参见下面的内容。

- 也可以在不将命令文件装载到 [**SQL 语句**] 窗格中的情况下，使用 Interactive SQL READ 语句运行命令文件。

  - **使用 Interactive SQL READ 语句运行命令文件**

  - 在 [**SQL 语句**] 窗格中，键入以下命令：

    ```
    READ 'c:\\filename.sql';
    ```

    在此语句中，*c:\filename.sql* 是文件的路径、名称和扩展名。只有当路径包含空格时才需要使用单引号（如上所示）。

    有关详细信息，请参见“READ 语句 [Interactive SQL]”一节 《SQL Anywhere 服务器 - SQL 参考》。

- 可以提供一个命令文件作为 Interactive SQL 的命令行参数。

♦ **在批处理模式下运行命令文件（命令提示符）**

- 运行 dbisql 实用程序并提供命令文件作为命令行参数。

  例如，以下命令针对 SQL Anywhere 示例数据库运行命令文件 *myscript.sql*。

```
dbisql -c "DSN=SQL Anywhere 11 Demo" myscript.sql
```

- 可以将命令文件装载到 [**SQL 语句**] 窗格中并在那里直接执行此命令文件。

♦ **将命令从文件装载到 [SQL 语句] 窗格中**

1. 选择 [**文件**] » [**打开**]。
2. 定位该文件，然后单击 [**打开**]。

   命令显示在 [**SQL 语句**] 窗格中，其中可以对其进行读取、编辑或执行。

在 Windows 平台上，可将 Interactive SQL 作为 *.sql* 命令文件的缺省编辑器。这样便可双击文件以使其内容出现在 Interactive SQL 的 [**SQL 语句**] 窗格中。请参见“将 Interactive SQL 设置为 .sql 文件的缺省编辑器”一节 《SQL Anywhere 服务器 - 数据库管理》。

- 还可以从收藏夹中将命令文件装载到 [**SQL 语句**] 窗格中。

  请参见“使用收藏夹”一节 《SQL Anywhere 服务器 - 数据库管理》。

将数据库输出写入文件

在 Interactive SQL 中，每个命令的结果集数据都会一直保留在 [**结果**] 窗格的 [**结果**] 选项卡中，直到执行下一命令。若要记录您的数据，您可以将每个语句的输出保存到单独的文件中。如果 *statement1* 和 *statement2* 是两个 SELECT 语句，则可以将它们分别输出到 *file1* 和 *file2* 中，如下所示：

statement1; OUTPUT TO *file1*;
statement2; OUTPUT TO *file2*;

例如，以下命令将查询结果保存到名为 *Employees.txt* 的文件中：

```
SELECT * FROM Employees;
OUTPUT TO 'C:\\My Documents\\Employees.txt';
```

有关详细信息，请参见“OUTPUT 语句 [Interactive SQL]”一节 《SQL Anywhere 服务器 - SQL 参考》。

Adaptive Server Enterprise 兼容性

您可以使用 BCP FORMAT 子句在 SQL Anywhere 和 Adaptive Server Enterprise 之间导入和导出文件。只要确保 BCP 输出是分隔的文本格式即可。如果要将 BLOB 数据从 SQL Anywhere 导出，以在 Adaptive Server Enterprise 中使用，则可将 BCP Format 子句与 UNLOAD TABLE 语句结合使用。

有关 BCP 和 FORMAT 子句的详细信息，请参见“LOAD TABLE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》或“UNLOAD 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

访问远程数据

目录

通过 SQL Anywhere 远程数据访问，您可以访问其它数据源中的数据。您可以使用此功能将数据迁移到 SQL Anywhere 数据库中。您也可以使用此功能跨数据库查询数据。

利用远程数据访问，您可以：

- 通过 SQL Anywhere 使用 insert-select 将数据从一个位置移动到另一个位置。
- 访问关系数据库（例如 Sybase、Oracle 和 DB2）中的数据。
- 访问桌面数据，例如 Excel 电子表格、Microsoft Access 数据库、FoxPro 和文本文件。
- 访问支持 ODBC 接口的任何其它数据源。
- 在本地和远程数据之间执行连接，不过，性能要比所有数据都位于一个 SQL Anywhere 数据库中时低得多。
- 在不同 SQL Anywhere 数据库的表之间执行连接操作。这样操作时的性能限制与使用其它远程数据源时是相同的。

- 在通常不具有 SQL Anywhere 功能的数据源上使用 SQL Anywhere 功能。例如，您可以对存储在 Oracle 中的数据使用 Java 功能，或在电子表格上执行子查询。SQL Anywhere 将通过对检索后的数据执行操作来补偿远程数据源不支持的功能。
- 使用直通模式直接访问远程服务器。
- 对其它服务器执行远程过程调用。

SQL Anywhere 允许访问以下外部数据源：

- SQL Anywhere
- Adaptive Server Enterprise
- Advantage Database Server
- Oracle
- IBM DB2
- Microsoft SQL Server
- Microsoft Access
- MySQL
- UltraLite
- 其它 ODBC 数据源

有关平台可用性，请参见 http://www.sybase.com/detail?id=1062617。

远程表映射

SQL Anywhere 将表提供给客户端应用程序的方式就像表中的所有数据都存储在与该应用程序连接的数据库中一样。在内部，当执行涉及远程表的查询时，先确定其存储位置，然后访问远程位置以便能够检索数据。

若要让远程表对客户端显示为本地表，您可以创建映射到远程数据的本地代理表。

- **创建代理表**

1. 定义远程数据所在的服务器。此操作指定服务器的类型和远程服务器的位置。请参见“使用远程服务器”一节第 730 页。
2. 如果本地和远程这两个服务器上的登录不同，则将本地用户登录信息映射到远程服务器用户登录信息。请参见“使用外部登录”一节第 738 页。
3. 创建代理表定义。这将指定本地代理表到远程表的映射。这其中包括远程表所在的服务器、数据库名、所有者名称、表名和远程表的列名。

有关详细信息，请参见“使用代理表”一节第 740 页。

管理远程表映射

若要管理远程表映射和远程服务器定义，您可以使用 Sybase Central，也可以使用像 Interactive SQL 这样的工具执行 SQL 语句。

小心
一些远程服务器（例如 Microsoft Access、Microsoft SQL Server 和 Sybase Adaptive Server Enterprise）不会跨越 COMMIT 和 ROLLBACK 保留游标。使用这些远程服务器时，不能使用 SQL Anywhere 插件中的 [数据] 选项卡来查看或修改代理表的内容。但是，只要关闭自动提交（这是 Interactive SQL 中的缺省行为），仍可使用 Interactive SQL 来查看和编辑这些代理表中的数据。其它 RDBMS（包括 Oracle、DB/2 和 SQL Anywhere）没有此限制。

服务器类

服务器类指定用于与服务器进行交互的访问方法。为每个远程服务器指派一个服务器类。不同类型的远程服务器要求使用不同的访问方法。服务器类为 SQL Anywhere 提供详细的服务器功能信息。SQL Anywhere 根据这些功能对它与远程服务器的交互进行调整。

有两组服务器类。第一组基于 ODBC，第二组基于 JDBC。

基于 ODBC 的服务器类包括：

- **saodbc** 用于 SQL Anywhere。
- **ulodbc** 用于 UltraLite。
- **aseodbc** 用于 Sybase SQL 服务器和 Adaptive Server Enterprise（版本 10 和更高版本。
- **adsodbc** 用于 Advantage 数据库服务器。
- **db2odbc** 用于 IBM DB2。
- **mssodbc** 用于 Microsoft SQL Server。
- **oraodbc** 用于 Oracle 服务器（版本 8.0 和更高版本）。
- **mysqlodbc** 用于 MySQL。
- **msaccessodbc** 用于 Microsoft Access。
- **odbc** 用于所有其它 ODBC 数据源。

注意
当使用远程数据访问时，如果使用不支持 Unicode 的 ODBC 驱动程序，则对来自此 ODBC 驱动程序的数据不执行字符集转换。

基于 JDBC 的服务器类包括：

- **sajdbc** 用于 SQL Anywhere。
- **asejdbc** 用于 Sybase SQL 服务器和 Adaptive Server Enterprise（版本 10 和更高版本）。

注意
JDBC 类会对性能产生相当大的影响，因此应该只在无法使用 ODBC 类的情况下使用。

有关远程服务器类的完整说明，请参见“用于远程数据访问的服务器类”第 757 页。

从 PowerBuilder DataWindows 访问远程数据

连接时将 DBParm Block 参数设置为 1，可使用 PowerBuilder DataWindow 访问远程数据。

- 在设计环境中，您可以通过访问 [**Database Profile Setup**] 窗口中的 [**Transaction**] 选项卡并将 [Retrieve Blocking Factor] 设置为 1 来设置 Block 参数。
- 在连接字符串中，使用以下参数：

```
DBParm="Block=1"
```

使用远程服务器

在可以将远程对象映射到本地代理表之前，您必须定义远程对象所在的远程服务器。定义远程服务器就是在 ISYSSERVER 系统表中为远程服务器添加一个条目。

使用 CREATE SERVER 语句创建远程服务器

使用 CREATE SERVER 语句设置远程服务器定义。要使用 Sybase Central 来创建远程服务器定义，请参见“使用 Sybase Central 创建远程服务器”一节第 731 页。

对于 ODBC 连接，每个远程服务器都对应于一个 ODBC 数据源。对于某些系统（包括 SQL Anywhere），每个数据源都描述一个数据库，因此每个数据库都需要一个单独的远程服务器定义。

您必须具有 RESOURCE 权限才能创建远程服务器。

在 Unix 平台上，您还需要引用 ODBC 驱动程序管理器。

有关 CREATE SERVER 语句的完整说明，请参见“CREATE SERVER 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

示例 1

下面的语句在 ISYSSERVER 系统表中为名为 RemoteASE 的 Adaptive Server Enterprise 服务器创建一个条目：

```
CREATE SERVER RemoteASE
CLASS 'ASEJDBC'
USING 'rimu:6666';
```

- **RemoteASE** 是远程服务器的名称。
- **ASEJDBC** 是一个关键字，指示该远程服务器是 Adaptive Server Enterprise 并且与服务器的连接基于 JDBC。
- **rimu:6666** 是远程服务器所在的计算机名和 TCP/IP 端口号。

示例 2

下面的语句在 ISYSSERVER 系统表中为名为 RemoteSA 的基于 ODBC 的 SQL Anywhere 服务器创建一个条目：

```
CREATE SERVER RemoteSA
CLASS 'SAODBC'
USING 'test4';
```

- **RemoteSA** 是在此数据库中使用的远程服务器名称。
- **SAODBC** 是一个关键字，指示该服务器是 SQL Anywhere 并且与服务器的连接使用 ODBC。
- **test4** 是 ODBC 数据源名称（Data Source Name，简称 DSN）。

示例 3

在 Unix 平台上，下面的语句在 ISYSSERVER 系统表中为名为 RemoteSA 的基于 ODBC 的 SQL Anywhere 服务器创建一个条目：

```
CREATE SERVER RemoteSA
CLASS 'SAODBC'
USING 'driver=SQL Anywhere 11;dsn=my_sa_dsn';
```

- **RemoteSA** 是在此数据库中使用的远程服务器名称。
- **SAODBC** 是一个关键字，指示该服务器是 SQL Anywhere 并且与服务器的连接使用 ODBC。
- **USING** 是对 ODBC 驱动程序管理器的引用。

示例 4

在 Unix 平台上，下面的语句在 ISYSSERVER 系统表中为名为 RemoteASE 的基于 ODBC 的 Adaptive Server Enterprise 服务器创建一个条目：

```
CREATE SERVER RemoteASE
CLASS 'ASEODBC'
USING '/opt/sybase/ase_odbc_1500/DataAccess/ODBC/lib/
libsybdrvodb.so;dsn=my_ase_dsn';
```

- **RemoteASE** 是在此数据库中使用的远程服务器名称。
- **ASEODBC** 是一个关键字，指示该服务器是 Adaptive Server Enterprise 并且与服务器的连接使用 ODBC。
- **USING** 是对 ODBC 驱动程序管理器的引用。

使用 Sybase Central 创建远程服务器

♦ **创建远程服务器 (Sybase Central)**

1. 以具有 DBA 权限的用户身份连接到主机数据库。
2. 在左窗格中，双击 [**远程服务器**]。
3. 从 [**文件**] 菜单中，选择 [**新建**] » [**远程服务器**]。
4. 在 [**您要给新远程服务器指定什么名称**] 字段中键入远程服务器的名称，然后单击 [**下一步**]。
5. 选择远程服务器类型，然后单击 [**下一步**]。
6. 选择连接类型，然后在 [**连接信息是什么**] 字段中键入连接信息：
   - 对于 ODBC，提供数据源名称或指定 ODBC Driver= 参数。
   - 对于 JDBC，提供 URL，形式为 *computer-name*:*port-number*。

   数据访问方法（JDBC 或 ODBC）是 SQL Anywhere 用来访问远程数据库的方法。此方法与 Sybase Central 用来连接到您的数据库的方法无关。
7. 单击 [**下一步**]。

8. 指定是否希望远程服务器是只读的，然后单击 [**下一步**]。
9. 单击 [**为当前用户创建外部登录**]，并完成必填字段。

   缺省情况下，SQL Anywhere 在代表当前用户连接远程服务器时使用该用户的用户 ID 和口令。但是，如果远程服务器没有定义与当前用户具有相同用户 ID 和口令的用户，则您必须创建一个外部登录。该外部登录为当前用户指派一个替代登录名和口令，以使该用户能够连接到远程服务器。请参见“CREATE EXTERNLOGIN 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。
10. 单击 [**测试连接**] 以测试远程服务器连接。
11. 单击 [**完成**]。

删除远程服务器

您可以使用 Sybase Central 或 DROP SERVER 语句从 ISYSSERVER 系统表中删除远程服务器。此操作若要成功，必须已经删除在该服务器上定义的所有远程表。

- **删除远程服务器 (Sybase Central)**

1. 以具有 DBA 权限的用户身份连接到主机数据库。
2. 在左窗格中，双击 [**远程服务器**]。
3. 选择远程服务器，然后选择 [**文件**] » [**删除**]。

- **删除远程服务器 (SQL)**

1. 以具有 DBA 权限的用户身份连接到主机数据库。
2. 执行 DROP SERVER 语句。

有关详细信息，请参见“DROP SERVER 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

示例

下面的语句删除名为 RemoteSA 的服务器：

```
DROP SERVER RemoteSA;
```

变更远程服务器

对远程服务器的更改直到下次连接远程服务器之后才生效。

- **变更远程服务器的属性 (Sybase Central)**

1. 以具有 RESOURCE 权限的用户身份连接到主机数据库。
2. 在左窗格中，双击 [**远程服务器**]。
3. 选择远程服务器，然后选择 [**文件**] » [**属性**]。

4. 更改远程服务器设置，然后单击 [**确定**]。

♦ **变更远程服务器的属性 (SQL)**

1. 以具有 DBA 权限的用户身份连接到主机数据库。
2. 执行 ALTER SERVER 语句。

示例

下面的语句将名为 RemoteASE 的服务器的服务器类更改为 aseodbc。在此示例中，该服务器的数据源名是 RemoteASE。

```
ALTER SERVER RemoteASE
CLASS 'aseodbc';
```

ALTER SERVER 语句也可用于启用或禁用服务器的已知功能。请参见“ALTER SERVER 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

列出服务器上的远程表

配置 SQL Anywhere 获取特定服务器上可用的远程表的列表时，使用 sp_remote_tables 系统过程可能会有帮助。sp_remote_tables 过程返回远程服务器上表的列表。

```
sp_remote_tables(
@server-name
[, @table-name
[, @table-owner
[, @table-qualifier
[, @with-table-type ] ] ] ]
)
```

如果指定 *table-name* 或 *table-owner*，则列表中仅包含与这些参数匹配的表。

例如，要从名为 excel 的远程服务器上获取所有可用 Microsoft Excel 工作表的列表：

```
CALL sp_remote_tables  excel;
```

或者，若要获取生产数据库（在名为 asetest 的 Adaptive Server Enterprise 服务器中）中由 fred 拥有的所有表的列表：

```
CALL sp_remote_tables  asetest, null, fred, production;
```

有关详细信息，请参见“sp_remote_tables 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》。

列出远程服务器功能

sp_servercaps 系统过程显示有关远程服务器功能的信息。SQL Anywhere 使用此功能信息来确定可以向远程服务器传递多少 SQL 语句。

您还可以通过查询 SYSCAPABILITY 和 SYSCAPABILITYNAME 系统视图查看远程服务器的功能信息。在 SQL Anywhere 第一次连接到远程服务器前，这些系统视图是空的。

当使用 sp_servercaps 系统过程时，指定的 *server-name* 必须与 CREATE SERVER 语句中使用的 *server-name* 相同。

按如下方式执行存储过程 sp_servercaps：

```
CALL sp_servercaps  server-name;
```

另请参见

- “sp_servercaps 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》
- “SYSCAPABILITY 系统视图”一节 《SQL Anywhere 服务器 - SQL 参考》
- “SYSCAPABILITYNAME 系统视图”一节 《SQL Anywhere 服务器 - SQL 参考》
- “CREATE SERVER 语句”一节 《SQL Anywhere 服务器 - SQL 参考》

使用目录访问服务器

目录访问服务器是一种远程服务器，通过它可访问运行数据库服务器的计算机的本地文件结构。一旦连接到目录访问服务器，就可以使用代理表访问该计算机上的任何子目录。数据库用户必须有外部登录才能使用目录访问服务器。

目录访问服务器一旦创建便不能变更。如果需要更改目录访问服务器，则必须先删除它，然后再使用不同的设置重新创建。

创建目录访问服务器

使用 CREATE SERVER 语句或 Sybase Central 中的 [**创建目录访问服务器向导**] 创建目录访问服务器。

创建目录访问服务器时，您可以控制能够访问的子目录数以及目录访问服务器是否能够用于修改现有文件。

设置目录访问服务器必须执行以下步骤：

1. 为目录创建远程服务器（需要 DBA 权限）。
2. 为可以使用目录访问服务器的数据库用户创建外部登录（需要 DBA 权限）。
3. 创建用于访问计算机上的目录的代理表（需要 RESOURCE 权限）。

♦ 创建并配置目录访问服务器 (Sybase Central)

1. 以具有 DBA 权限的用户身份连接到数据库。
2. 在左窗格中，双击 [**目录访问服务器**]。
3. 从 [**文件**] 菜单中，选择 [**新建**] » [**目录访问服务器**]。
4. 按照 [**创建目录访问服务器向导**] 中的说明进行操作。

♦ 创建并配置目录访问服务器 (SQL)

1. 以具有 DBA 权限的用户身份连接到主机数据库。
2. 使用 CREATE SERVER 语句创建远程服务器。

   例如：

   ```
   CREATE SERVER my_dir_tree
   CLASS 'directory'
   USING 'root=c:\Program Files';
   ```

3. 使用 CREATE EXTERNLOGIN 语句创建外部登录。

   例如：

   ```
   CREATE EXTERNLOGIN DBA TO my_dir_tree;
   ```

4. 使用 CREATE EXISTING TABLE 语句为目录创建代理表。

例如：

```
CREATE EXISTING TABLE my_program_files AT 'my_dir_tree;;;.';
```

在此示例中，my_program_files 是目录名称，my_dir_tree 是目录访问服务器名称。

示例

以下语句创建一个名为 directoryserver3 的新目录访问服务器（可用于访问最多三级子目录），为 DBA 用户创建一个到该目录访问服务器的外部登录，并创建一个名为 diskdir3 的代理表。

```
CREATE SERVER directoryserver3
CLASS 'DIRECTORY'
USING 'ROOT=c:\mydir;SUBDIRS=3';
CREATE EXTERNLOGIN DBA TO directoryserver3;
CREATE EXISTING TABLE diskdir3 AT 'directoryserver3;;;.';
```

使用 sp_remote_tables 系统过程，您可以看见运行该数据库服务器的计算机上 *c:\mydir* 下的所有子目录：

```
CALL sp_remote_tables( 'directoryserver3' );
```

使用以下 SELECT 语句，您可以查看文件 *c:\mydir\myfile.txt* 的内容：

```
SELECT contents
FROM diskdir3
WHERE file_name = 'myfile.txt';
```

或者，您可以从这些目录中选择数据：

```
-- Get the list of directories in this disk directory tree.
SELECT permissions, file_name, size
FROM diskdir3
WHERE PERMISSIONS LIKE 'd%';
-- Get the list of files.
SELECT permissions, file_name, size
FROM diskdir3
WHERE PERMISSIONS NOT LIKE 'd%';
```

另请参见

- “CREATE SERVER 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “CREATE EXTERNLOGIN 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “CREATE TABLE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “CREATE EXISTING TABLE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》

删除目录访问服务器

您无法变更现有的目录访问服务器：您必须使用 DROP SERVER 语句删除现有目录访问服务器，然后创建一个新目录访问服务器。

删除目录访问服务器

♦ 删除目录访问服务器 (Sybase Central)

1. 以具有 DBA 权限的用户身份连接到数据库。
2. 在左窗格中，双击 [**目录访问服务器**]。
3. 选择目录访问服务器，然后选择 [**编辑**] » [**删除**]。

♦ 删除目录访问服务器 (SQL)

1. 以具有 DBA 权限的用户身份连接到主机数据库。
2. 执行 DROP SERVER 语句。

   例如：

   ```
   DROP SERVER my_directory_server;
   ```

删除代理表

使用 DROP TABLE 语句删除目录访问服务器使用的代理表。

♦ 删除代理表 (Sybase Central)

1. 以具有 DBA 权限的用户身份连接到数据库。
2. 在左窗格中，双击 [**目录访问服务器**]。
3. 在右窗格中，单击 [**代理表**] 选项卡。
4. 选择代理表，然后选择 [**编辑**] » [**删除**]。
5. 单击 [**是**]。

♦ 删除代理表 (SQL)

1. 以具有 DBA 权限的用户身份连接到主机数据库。
2. 执行 DROP TABLE 语句。

   例如：

   ```
   DROP TABLE my_files;
   ```

另请参见

- “DROP SERVER 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “DROP TABLE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》

使用外部登录

缺省情况下，SQL Anywhere 每次代表其客户端连接到远程服务器时都会使用这些客户端的名称和口令。但是，您可以通过创建外部登录来替换这一缺省设置。外部登录是与远程服务器通信时使用的替代登录名和口令。

有关详细信息，请参见“使用集成登录”一节 《SQL Anywhere 服务器 - 数据库管理》。

创建外部登录

使用以下过程之一创建外部登录。

- **创建外部登录 (Sybase Central)**

1. 作为具有 DBA 权限的用户或作为外部登录的所有者连接到主机数据库。
2. 在左窗格中，双击 [**远程服务器**]。
3. 选择远程服务器，然后在右窗格中单击 [**外部登录**] 选项卡。
4. 从 [**文件**] 菜单中，选择 [**新建**] » [**外部登录**]。
5. 请按照 [**创建外部登录向导**] 中的说明进行操作。

- **创建外部登录 (SQL)**

1. 作为具有 DBA 权限的用户或作为外部登录的所有者连接到主机数据库。
2. 执行 CREATE EXTERNLOGIN 语句。

示例

以下语句允许本地用户 fred 通过使用远程登录 frederick 和口令 banana 获得对服务器 RemoteASE 的访问权限。

```
CREATE EXTERNLOGIN fred
TO RemoteASE
REMOTE LOGIN frederick
IDENTIFIED BY banana;
```

有关详细信息，请参见“CREATE EXTERNLOGIN 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

删除外部登录

使用以下过程之一从 SQL Anywhere 系统表中删除外部登录。

- **删除外部登录 (Sybase Central)**

1. 作为具有 DBA 权限的用户或作为外部登录的所有者连接到主机数据库。

2. 在左窗格中，双击 [**远程服务器**]。
3. 选择远程服务器，然后在右窗格中单击 [**外部登录**] 选项卡。
4. 选择外部登录，然后选择 [**文件**] » [**删除**]。
5. 单击 [**是**]。

♦ 删除外部登录 (SQL)

1. 作为具有 DBA 权限的用户或作为外部登录的所有者连接到主机数据库。
2. 执行 DROP EXTERNLOGIN 语句。

示例

以下语句删除上面的示例中创建的本地用户 fred 的外部登录：

```
DROP EXTERNLOGIN fred TO RemoteASE;
```

另请参见

- “DROP EXTERNLOGIN 语句”一节 《SQL Anywhere 服务器 - SQL 参考》

使用代理表

远程数据的位置透明性是通过创建映射到远程对象的本地**代理表**来实现的。您可以使用代理表访问远程数据库以代理表候选对象方式导出的任何对象（包括表、视图和实例化视图）。使用以下语句之一创建代理表：

- 如果该表已存在于远程存储位置，则使用 CREATE EXISTING TABLE 语句。此语句为远程服务器上的现有表定义代理表。
- 如果该表未存在于远程存储位置，则使用 CREATE TABLE 语句。此语句在远程服务器上创建新表，并且还为该表定义代理表。

注意

处于保存点中时不能修改代理表中的数据。请参见“事务内的保存点”一节第 104 页。

当触发器在代理表上触发时，使用的权限是导致触发器触发的用户的权限，而不是代理表所有者的权限。

指定代理表位置

CREATE TABLE 和 CREATE EXISTING TABLE 都可以使用 AT 关键字定义现有对象的位置。该位置字符串由四部分组成，每部分都由句点或分号隔开。分号分隔符允许在数据库和所有者字段中使用文件名和扩展名。

AT 子句的语法是

```
... AT 'server.database.owner.table-name'
```

- **server**　这是在当前数据库中使用的服务器名称（在 CREATE SERVER 语句中指定的名称）。对于所有远程数据源，此字段是必需的。
- **database**　database 字段的含义取决于数据源。在某些情况下，此字段不适用，应该留空。但是，分隔符仍是必需的。

  如果数据源是 Adaptive Server Enterprise，则 *database* 指定表所在的数据库。例如 master 或 pubs2。

  如果数据源是 SQL Anywhere，则此字段不适用；将其留空。

  如果数据源是 Excel、Lotus Notes 或 Access，则必须包括包含表的文件的名称。如果该文件名中包含句点，请使用分号分隔符。
- **owner**　如果数据库支持所有权的概念，则此字段表示所有者名。只有当多个所有者具有使用相同名称的表时才需要此字段。
- **table-name**　此字段指定表的名称。如果是 Excel 电子表格，则这是工作簿中工作表的名称。如果将 *table-name* 留空，则假定远程表名与本地代理表名相同。

示例：

以下示例说明位置字符串的用法：

- SQL Anywhere：

```
'RemoteSA..GROUPO.Employees'
```

- Adaptive Server Enterprise：

```
'RemoteASE.pubs2.dbo.publishers'
```

- Excel：

```
'excel;d:\pcdb\quarter3.xls;;sheet1$'
```

- Access：

```
'access;\\server1\production\inventory.mdb;;parts'
```

创建代理表 (Sybase Central)

使用以下过程之一创建代理表。您不能为系统表创建代理表。

CREATE EXISTING TABLE 语句创建映射到远程服务器上现有表的代理表。SQL Anywhere 从位于远程位置的对象派生列属性和索引信息。

有关 CREATE EXISTING TABLE 语句的信息，请参见“CREATE EXISTING TABLE 语句”一节《SQL Anywhere 服务器 - SQL 参考》。

♦ 创建代理表 (Sybase Central)

1. 以具有 DBA 权限的用户身份连接到主机数据库。
2. 在左窗格中，双击 [**远程服务器**]。
3. 选择远程服务器，然后在右窗格中单击 [**代理表**] 选项卡。
4. 从 [**文件**] 菜单中，选择 [**新建**] » [**代理表**]。
5. 请按照 [**创建代理表向导**] 中的说明进行操作。

使用 CREATE EXISTING TABLE 语句创建代理表

CREATE EXISTING TABLE 语句创建映射到远程服务器上现有表的代理表。SQL Anywhere 从位于远程位置的对象派生列属性和索引信息。

♦ 使用 CREATE EXISTING TABLE 语句创建代理表 (SQL)

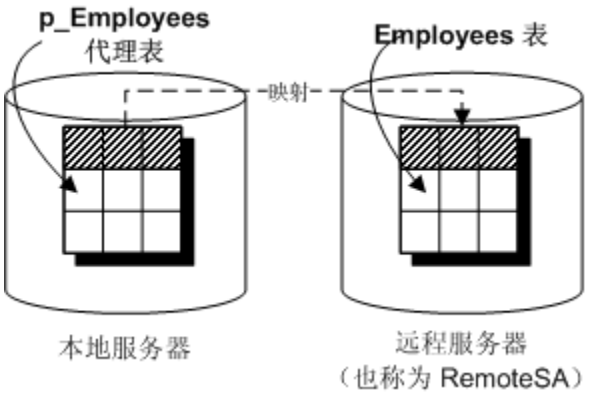
1. 以具有 DBA 权限的用户身份连接到主机数据库。
2. 执行 CREATE EXISTING TABLE 语句。

有关详细信息，请参见“CREATE EXISTING TABLE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

示例 1

若要在当前服务器上创建名为 p_Employees 的代理表以映射到 RemoteSA 服务器上名为 Employees 的远程表，请使用下面的语法：

```
CREATE EXISTING TABLE p_Employees
AT 'RemoteSA..GROUPO.Employees';
```



示例 2

以下语句将代理表 a1 映射到 Microsoft Access 文件 *mydbfile.mdb*。在本示例中，AT 关键字使用分号 (;) 作为分隔符。为 Microsoft Access 定义的服务器名为 access。

```
CREATE EXISTING TABLE a1
AT 'access;d:\mydbfile.mdb;;a1';
```

使用 CREATE TABLE 语句创建代理表

CREATE TABLE 语句在远程服务器上创建新表，如果使用 AT 选项，则可以为该表定义代理表。使用 SQL Anywhere 数据类型定义列。SQL Anywhere 会自动将数据转换为远程服务器的本机类型。

如果您使用 CREATE TABLE 语句创建本地和远程表，并随后使用 DROP TABLE 语句删除代理表，则远程表也将被删除。但是，您可以使用 DROP TABLE 语句删除使用 CREATE EXISTING TABLE 语句创建的代理表。在这种情况下，不会删除远程表。

有关详细信息，请参见“CREATE TABLE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》和“CREATE EXISTING TABLE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

示例

下面的语句在远程服务器 RemoteSA 上创建名为 Employees 的表，并创建映射到该远程表的名为 Members 的代理表：

```
CREATE TABLE Members
( membership_id INTEGER NOT NULL,
member_name CHAR( 30 ) NOT NULL,
office_held CHAR( 20 ) NULL )
AT 'RemoteSA..GROUPO.Employees';
```

列出远程表上的列

在执行 CREATE EXISTING TABLE 语句之前，获得远程表上可用列的列表可能很有帮助。sp_remote_columns 系统过程生成远程表上列的列表和这些数据类型的说明。以下是 sp_remote_columns 系统过程的语法：

```
sp_remote_columns servername, tablename [, owner ]
[, database]
```

如果给定了表名、所有者或数据库名，则列的列表仅限于那些符合条件的列。

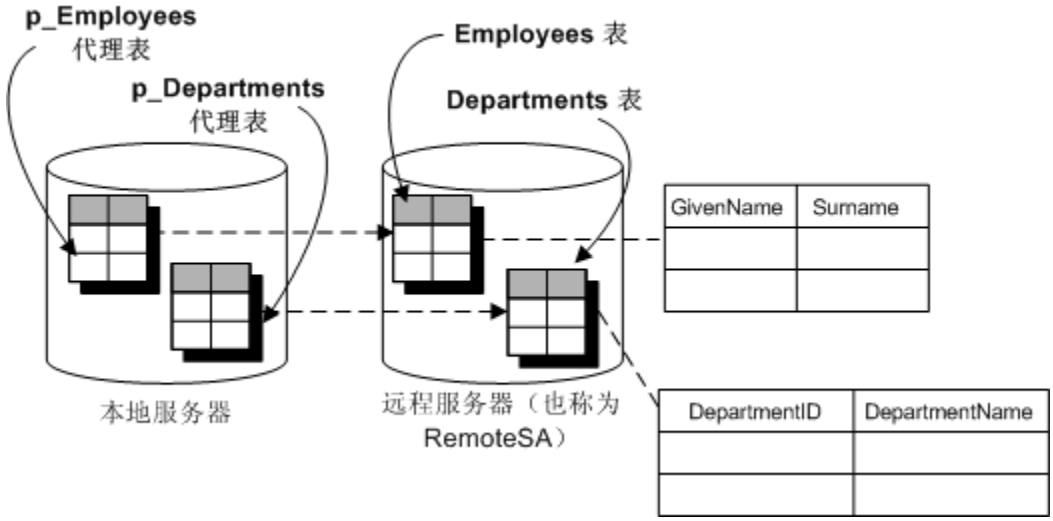
例如，以下语句返回名为 asetest 的 Adaptive Server Enterprise 服务器上 production 数据库中 sysobjects 表中列的列表：

```
CALL sp_remote_columns  asetest, sysobjects, null, production;
```

有关详细信息，请参见“sp_remote_columns 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》。

连接远程表

下图说明本地数据库服务器上的代理表映射到了远程服务器 RemoteSA 上 SQL Anywhere 示例数据库中的远程表 Employees 和 Departments。



您可以在不同 SQL Anywhere 数据库上的表之间使用连接。下面是一个简单的示例，该示例只使用一个数据库来说明这些原则。

♦ 在两个远程表之间进行连接 (SQL)

1. 创建一个名为 *empty.db* 的新数据库。

   此数据库未保存任何数据。它只用来定义远程对象和访问 SQL Anywhere 示例数据库。

2. 启动运行 *empty.db* 的数据库服务器。您可以使用以下命令行完成此步骤：

   ```
   dbeng11 empty
   ```

3. 从 Interactive SQL 以用户 DBA 身份连接到 *empty.db*。

4. 在新数据库中，创建一个名为 RemoteSA 的远程服务器。其服务器类为 saodbc，连接字符串是指 DSN SQL Anywhere 11 Demo：

   ```
   CREATE SERVER RemoteSA
   CLASS 'saodbc'
   USING 'SQL Anywhere 11 Demo';
   ```

5. 在本示例中，您在远程数据库和本地数据库上使用相同的用户 ID 和口令，因此不需要外部登录。

   在某些情况下，连接远程服务器上的数据库时必须提供用户 ID 和口令。在新数据库中，您能够创建到远程服务器的外部登录。在本例中，为简单起见，本地登录名和远程用户 ID 均为 DBA：

   ```
   CREATE EXTERNLOGIN DBA
   TO RemoteSA
   ```

```
REMOTE LOGIN DBA
IDENTIFIED BY sql;
```

6. 定义 p_Employees 代理表：

```
CREATE EXISTING TABLE p_Employees
AT 'RemoteSA..GROUPO.Employees';
```

7. 定义 p_Departments 代理表：

```
CREATE EXISTING TABLE p_Departments
AT 'RemoteSA..GROUPO.Departments';
```

8. 在 SELECT 语句中使用代理表执行连接。

```
SELECT GivenName, Surname, DepartmentName
FROM p_Employees JOIN p_Departments
ON p_Employees.DepartmentID = p_Departments.DepartmentID
ORDER BY Surname;
```

连接多个本地数据库中的表

SQL Anywhere 服务器上可能有多个本地数据库在同时运行。通过将其它本地 SQL Anywhere 数据库中的表定义为远程表，您可以执行跨数据库连接。

有关指定多个数据库的详细信息，请参见“CREATE SERVER 语句中的 USING 参数”一节 第 772 页。

示例

假设您正在使用数据库 db1，您要访问数据库 db2 的表中的数据。您需要建立指向数据库 db2 中的表的代理表定义。例如，在名为 RemoteSA 的 SQL Anywhere 服务器上，您可能有三个数据库可用：db1、db2 和 db3。

1. 如果使用 ODBC，则需要为您将访问的每个数据库创建一个 ODBC 数据源名称。
2. 连接到您将执行连接的数据库之一。例如，连接到 db1。
3. 为每个您将访问的其它本地数据库执行 CREATE SERVER 语句。这将建立到 SQL Anywhere 服务器的**回送**连接。

```
CREATE SERVER remote_db2
CLASS 'saodbc'
USING 'RemoteSA_db2';
CREATE SERVER remote_db3
CLASS 'saodbc'
USING 'RemoteSA_db3';
```

或者，使用 JDBC：

```
CREATE SERVER remote_db2
CLASS 'sajdbc'
USING 'mypc1:2638/db2';
CREATE SERVER remote_db3
CLASS 'sajdbc'
USING 'mypc1:2638/db3';
```

4. 通过执行 CREATE EXISTING TABLE 语句为您要访问的其它数据库中的表创建代理表定义。

```
CREATE EXISTING TABLE Employees
AT 'remote_db2...Employees';
```


将本机语句发送到远程服务器

使用 FORWARD TO 语句将一个或多个语句以其本机语法发送到远程服务器。可以通过两种方式使用此语句：

- 将语句发送到远程服务器。
- 让 SQL Anywhere 进入直通模式以便向远程服务器发送一系列语句。

FORWARD TO 语句可用于验证是否正确配置了服务器。如果您将语句发送到远程服务器，SQL Anywhere 未返回错误消息，则远程服务器配置正确。

不能在过程或批处理中使用 FORWARD TO 语句。

如果无法建立与指定服务器的连接，则会向用户返回一条消息。如果建立了连接，则会将任何结果转换为客户端程序可以识别的形式。

有关详细信息，请参见“FORWARD TO 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

示例 1

下面的语句通过选择版本字符串校验与名为 RemoteASE 的服务器的连接：

```
FORWARD TO RemoteASE {SELECT @@version};
```

示例 2

下面的语句显示与名为 RemoteASE 的服务器的直通会话：

```
FORWARD TO RemoteASE
SELECT * FROM titles
SELECT * FROM authors
FORWARD TO;
```

使用远程过程调用 (RPC)

SQL Anywhere 用户可以向支持此功能的远程服务器发出过程调用。

SQL Anywhere、Adaptive Server Enterprise、Oracle 和 DB2 支持此功能。发出远程过程调用与使用本地过程调用类似。

SQL Anywhere 支持读取远程过程的结果集，包括读取多个结果集。您也可以使用远程函数读取远程过程和函数的返回值。可以在 SELECT 语句的 FROM 子句中使用远程过程。

创建远程过程

使用以下过程之一发出远程过程调用。

远程过程接受长度最多 254 个字节的输入参数，并在输出变量中返回最多 254 个字符。

如果远程过程可以返回结果集，即使并不是在所有情况下都返回，本地过程定义也必须包含 RESULT 子句。

♦ 创建远程过程 (Sybase Central)

1. 以具有 DBA 权限的用户身份连接到主机数据库。
2. 在左窗格中，双击 [**远程服务器**]。
3. 选择远程服务器，然后在右窗格中单击 [**远程过程**] 选项卡。
4. 从 [**文件**] 菜单中，选择 [**新建**] » [**远程过程**]。
5. 请按照 [**创建远程过程向导**] 中的说明进行操作。

♦ 创建远程过程 (SQL)

1. 以具有 DBA 权限的用户身份连接到数据库。
2. 定义发给 SQL Anywhere 的过程。

   语法与本地过程定义相同，只是不使用 SQL 语句组成该过程的主体，而是提供一个位置字符串定义该过程所驻留的位置。

```
CREATE PROCEDURE remotewho()
AT 'bostonase.master.dbo.sp_who';
```

有关详细信息，请参见“CREATE PROCEDURE 语句（Web 服务）”一节 《SQL Anywhere 服务器 - SQL 参考》。

♦ 发布远程过程调用 (SQL)

1. 以具有 DBA 权限的用户身份连接到数据库。
2. 执行该过程，如下所示：


```
CALL remotewho();
```

示例

本示例在调用远程过程时指定一个参数：

```
CREATE PROCEDURE remoteuser ( IN uname CHAR( 30 ) )
AT 'bostonase.master.dbo.sp_helpuser';
CALL remoteuser( 'joe' );
```

远程过程的数据类型

RPC 参数可以使用以下数据类型：

- [UNSIGNED] SMALLINT
- [UNSIGNED] INT
- [UNSIGNED] BIGINT
- TINYINT
- REAL
- DOUBLE
- CHAR
- BIT
- NUMERIC 和 DECIMAL 数据类型可用于 IN 参数，但不能用于 OUT 或 INOUT 参数

删除远程过程

使用以下过程之一删除远程过程。

♦ **删除远程过程 (Sybase Central)**

1. 以具有 DBA 权限的用户身份连接到数据库。
2. 在左窗格中，双击 [**远程服务器**]。
3. 选择远程服务器，然后在右窗格中单击 [**远程过程**] 选项卡。
4. 选择远程过程，然后选择 [**文件**] » [**删除**]。
5. 单击 [**是**]。

♦ **删除远程过程 (SQL)**

1. 以具有 DBA 权限的用户身份连接到数据库。
2. 执行 DROP PROCEDURE 语句。

有关详细信息，请参见“DROP PROCEDURE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

示例

删除名为 remoteproc 的远程过程。

```
DROP PROCEDURE remoteproc;
```


事务管理和远程数据

事务提供一种对 SQL 语句进行分组的方式，这样可以将这些语句作为一个整体对待—要么将这些语句执行的所有工作都提交到数据库，要么一个也不提交。

远程表的事务管理在很大程度上与 SQL Anywhere 中本地表的事务管理相同，但也有一些差异。下一节将对它们进行讨论。

有关事务的一般性讨论，请参见“使用事务和隔离级别”第 99 页。

远程事务管理概述

管理涉及远程服务器的事务的方法使用两阶段提交协议。SQL Anywhere 执行的策略可以确保大多数情况的事务完整性。但是，当一个事务中调用了多个远程服务器时，仍然存在分布式工作单位处于不确定状态的可能性。即使使用了两阶段提交协议，也不包括恢复过程。

管理用户事务的一般逻辑如下：

1. SQL Anywhere 以 BEGIN TRANSACTION 通知向远程服务器宣布工作。
2. 当准备好提交事务时，SQL Anywhere 向已成为事务一部分的每个远程服务器发送一个 PREPARE TRANSACTION 通知。这可以确保远程服务器准备好提交事务。
3. 如果 PREPARE TRANSACTION 请求失败，则将指示所有远程服务器回退当前事务。

   如果所有 PREPARE TRANSACTION 请求都成功，则服务器将向该事务涉及的每个远程服务器发送一个 COMMIT TRANSACTION 请求。

以 BEGIN TRANSACTION 开头的任何语句都可以开始一个事务。其它语句被发送到远程服务器作为单个、远程工作单元执行。

事务管理的限制

事务管理的限制如下：

- 不将保存点传播到远程服务器。
- 如果涉及远程服务器的事务中包含嵌套的 BEGIN TRANSACTION 和 COMMIT TRANSACTION 语句，则只处理最外层的一组语句。不会将最内层的语句组（包含 BEGIN TRANSACTION 和 COMMIT TRANSACTION 语句）传输到远程服务器。

内部操作

本节介绍 SQL Anywhere 代表客户端应用程序在远程服务器上执行的基础步骤。

查询分析

当数据库服务器从客户端接收到语句时，将对该语句进行分析。如果语句不是有效的 SQL Anywhere SQL 语句，数据库服务器将报告错误。

查询规范化

将校验查询中引用的对象并检查某些数据类型兼容性。

例如，假定有以下查询：

```
SELECT *
FROM t1
WHERE c1 = 10;
```

查询规范化阶段验证系统表中是否存在具有列 c1 的表 t1。它还验证列 c1 的数据类型是否和值 10 兼容。例如，如果该列的数据类型为 datetime，则将拒绝此语句。

查询预处理

查询预处理准备对查询进行优化。它可能会更改语句的表示形式，因此，即使在语义上等效，SQL Anywhere 生成的用于传递到远程服务器的 SQL 语句在语句构成上也将不同于原始语句。

预处理执行视图扩展，以便查询可以对视图引用的表执行操作。可以对表达式进行重新排序并对子查询进行转换以提高处理效率。例如，可以将某些子查询转换为连接。

服务器功能

前面的步骤是对所有查询（本地和远程）执行的。

以下步骤取决于 SQL 语句的类型和所涉及的远程服务器的功能。

在 SQL Anywhere 中，每个远程服务器都有一组为其定义的功能。这些功能存储在 ISYSCAPABILITIES 系统表中，在第一次连接到远程服务器时初始化。

通用服务器类 odbc 严格按照从 ODBC 驱动程序返回的信息来确定这些功能。其它服务器类（例如 db2odbc）具有远程服务器类型功能的更详细的信息，并使用该信息补充驱动程序返回的内容。

一旦将服务器添加到 ISYSCAPABILITIES，就只能从该系统表检索功能信息。

由于远程服务器可能并不支持给定 SQL 语句的所有功能，所以 SQL Anywhere 必须将语句分解成较简单的组成部分，达到可以将该查询提供给远程服务器的程度。未传递给远程服务器的 SQL 功能必须由 SQL Anywhere 本身执行。

例如，某个查询可能包含一个 ORDER BY 语句。如果远程服务器不能执行 ORDER BY，则将语句发送到远程服务器时将不包括 ORDER BY，SQL Anywhere 对远程服务器返回的结果执行 ORDER BY，然后再将最终结果返回给用户。这样做的结果是用户可以使用 SQL Anywhere 支持的全部 SQL 功能，而不用担心特定后端的功能。

语句的完整直通

为提高效率，SQL Anywhere 会尽可能多地将语句传递给远程服务器。通常，这将是最初提供给 SQL Anywhere 的完整语句。

当满足以下条件时，SQL Anywhere 将传递出完整的语句：

- 语句中的所有表都驻留在同一个远程服务器上。
- 远程服务器能够处理语句中的所有语法。

在少数情况下，让 SQL Anywhere 执行某些工作实际上可能比由远程服务器来执行这些工作更高效。例如，SQL Anywhere 可能具有更好的排序算法。在这种情况下，您可以考虑使用 ALTER SERVER 语句变更远程服务器的功能。

有关详细信息，请参见“ALTER SERVER 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

语句的部分直通

如果某个语句包含对多个服务器的引用，或者使用远程服务器不支持的 SQL 功能，则会将查询分解为多个较简单的部分。

SELECT

通过删除不能传递的部分并让 SQL Anywhere 执行工作，可以对 SELECT 语句进行分解。例如，假设远程服务器不能处理以下语句中的 ATAN2 函数：

```
SELECT a,b,c
WHERE ATAN2( b, 10 ) > 3
AND c = 10;
```

发送到远程服务器的语句将被转换为：

```
SELECT a,b,c WHERE c = 10;
```

然后，SQL Anywhere 在本地将 [`WHERE ATAN2( b, 10 ) > 3`] 应用到中间结果集。

连接

当连接两个表时，将选择一个表作为外表。根据应用到外表的 WHERE 条件对外表进行扫描。对于找到的每个符合条件的行，将对另一个表（称为内部表）进行扫描以查找符合连接条件的行。

当引用远程表时将使用相同的算法。因为搜索远程表通常要比搜索本地表的开销高很多（由于网络 I/O），所以将采取一切办法让远程表成为连接中最外层的表。

UPDATE 和 DELETE

找到符合条件的行时，如果 SQL Anywhere 不能将 UPDATE 或 DELETE 语句完整地传递给远程服务器，则它必须将该语句转换成一个表扫描，该表扫描包含原始 WHERE 子句的尽可能多的部分，后跟指定 WHERE CURRENT OF *cursor-name* 的定位 UPDATE 或 DELETE 语句。

例如，如果远程服务器不支持函数 ATAN2：

```
UPDATE t1
SET a = atan2( b, 10 )
WHERE b > 5;
```

将被转换为以下内容：

```
SELECT a,b
FROM t1
WHERE  b > 5;
```

每次找到一行时，SQL Anywhere 都会计算出 a 的新值并发出：

```
UPDATE t1
SET a = 'new value'
WHERE CURRENT OF CURSOR;
```

如果 a 已有等于该新值的值，则没有必要执行定位的 UPDATE，因此也不会远程发送定位的 UPDATE。

为了处理需要表扫描的 UPDATE 或 DELETE 语句，远程数据源必须支持执行定位 UPDATE 或 DELETE (WHERE CURRENT OF *cursor-name*) 的功能。某些数据源不支持此功能。

不能更新临时表
如果需要中间临时表，则不能执行 UPDATE 或 DELETE。这种情况发生在具有 ORDER BY 的查询和某些具有子查询的查询中。

远程数据访问疑难解答

本节提供访问远程服务器疑难解答的一些提示。

不支持远程数据的功能

不支持对远程数据执行以下 SQL Anywhere 功能：

- 对远程表执行 ALTER TABLE 语句
- 在代理表上定义的触发器
- SQL Remote
- 引用远程表的外键
- READTEXT、WRITETEXT 和 TEXTPTR 函数
- 定位的 UPDATE 和 DELETE 语句
- 需要中间临时表的 UPDATE 和 DELETE 语句
- 向后滚动对远程数据打开的游标 Fetch 语句必须是 NEXT 或 RELATIVE 1
- 调用包含引用代理表的表达式的函数
- 如果远程表上的某列的名称是远程服务器上的关键字，则不能访问该列中的数据。您可以执行 CREATE EXISTING TABLE 语句并导入定义，但不能选择该列。

区分大小写

您的 SQL Anywhere 数据库的区分大小写设置应该和所访问的任何远程服务器使用的设置匹配。

缺省情况下，创建 SQL Anywhere 数据库时不区分大小写。如果使用此配置，当从区分大小写的数据库进行选择时将出现不可预知的结果。根据 ORDER BY 或字符串比较是被传递到远程服务器还是由本地 SQL Anywhere 服务器执行，将出现不同的结果。

连接测试

执行以下步骤以确定您可以连接到远程服务器：

- 在配置 SQL Anywhere 之前使用客户端工具（例如 Interactive SQL）确定您可以连接到远程服务器。
- 执行到远程服务器的简单的直通语句以检查您的连接和远程登录配置。例如：

  ```
  FORWARD TO RemoteSA {SELECT @@version};
  ```

- 打开远程跟踪以跟踪与远程服务器的交互。例如：

```
SET OPTION cis_option = 7;
```

一旦开启远程跟踪功能，跟踪信息就会出现在数据库服务器消息窗口中。通过在启动数据库服务器时指定 -o 服务器选项，您可以将此输出记录到文件中。

有关 cis_option 选项的详细信息，请参见“cis_option 选项 [数据库]”一节 《SQL Anywhere 服务器 - 数据库管理》。

有关 -o 服务器选项的详细信息，请参见“-o 服务器选项”一节 《SQL Anywhere 服务器 - 数据库管理》。

查询的一般问题

如果 SQL Anywhere 处理对远程表的查询有困难，那么了解 SQL Anywhere 如何执行该查询通常会很有帮助。您可以显示远程跟踪及查询执行计划的说明：

```
SET OPTION cis_option = 7;
```

一旦开启远程跟踪功能，跟踪信息就会出现在数据库服务器消息窗口中。通过在启动数据库服务器时指定 -o 服务器选项，您可以将此输出记录到文件中。

有关在使用远程数据访问时使用 cis_option 选项调试查询的详细信息，请参见“cis_option 选项 [数据库]”一节 《SQL Anywhere 服务器 - 数据库管理》。

有关 -o 服务器选项的详细信息，请参见“-o 服务器选项”一节 《SQL Anywhere 服务器 - 数据库管理》。

查询自身阻塞

您必须具有足够的可用线程来支持由某个查询运行的各个任务。如果未能提供所需任务数目的线程，则可能导致查询自身阻塞。请参见“事务阻塞和死锁”一节第 118 页。

管理通过 ODBC 执行的远程数据访问连接

如果您通过 ODBC 访问远程数据库，则会为到远程服务器的连接指定一个名称。可以使用该名称删除连接以取消远程请求。

连接会被命名为 ASACIS_*conn-name*，其中 *conn-name* 是本地连接的连接 ID。该连接 ID 可以从 sa_conn_info 存储过程获得。请参见“sa_conn_info 系统过程”一节 《SQL Anywhere 服务器 - SQL 参考》。

用于远程数据访问的服务器类

目录

您在 CREATE SERVER 语句中指定的服务器类决定远程连接的行为。这些服务器类为 SQL Anywhere 提供详细的服务器功能信息。SQL Anywhere 会将 SQL 语句的格式设置为适用于服务器的功能。

服务器类包含两个类别：

- 基于 ODBC 的服务器类
- 基于 JDBC 的服务器类

每一个服务器类都具有一组独特的特性，数据库管理员和程序员需要知道这些特性，才能配置用于进行远程数据访问的服务器。

您应参考关于服务器类类别（基于 JDBC 或基于 ODBC）的常规信息以及特定于各个服务器类的信息。

基于 ODBC 的服务器类

基于 ODBC 的服务器类包括：

- saodbc
- aseodbc
- db2odbc
- mssodbc
- oraodbc
- msaccessodbc
- mysqlodbc
- ulodbc
- adsodbc
- odbc

注意
当使用远程数据访问时，如果使用不支持 Unicode 的 ODBC 驱动程序，则对来自此 ODBC 驱动程序的数据不执行字符集转换。

定义 ODBC 外部服务器

定义基于 ODBC 的服务器的最常用方法是使其以 ODBC 数据源为基础。若要这样做，可使用 [ODBC 管理器] 来创建数据源。

有关详细信息，请参见“创建 ODBC 数据源”一节 《SQL Anywhere 服务器 - 数据库管理》。

定义完数据源之后，CREATE SERVER 语句中的 USING 子句应与 ODBC 数据源名称匹配。

例如，要配置名为 mydb2 的 DB2 服务器（其数据源名称也是 mydb2），请使用：

```
CREATE SERVER mydb2
CLASS 'db2odbc'
USING 'mydb2';
```

有关详细信息，请参见“CREATE SERVER 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

使用连接字符串代替数据源

有一种替代方法可以避免使用数据源，即在 CREATE SERVER 语句的 USING 子句中提供连接字符串。若要提供连接字符串，必须知道您所使用的 ODBC 驱动程序的连接参数。例如，与 SQL Anywhere 数据库的连接可能如下所示：

```
CREATE SERVER TestSA
CLASS 'saodbc'
USING 'DRIVER=SQL Anywhere 11;ENG=TestSA;DBN=sample;LINKS=tcpip()';
```

上述语句定义了与名为 TestSA 的 SQL Anywhere 数据库服务器的连接，其中数据库是 sample，使用的协议是 TCP-IP。

另请参见

有关特定的 ODBC 服务器类的信息，请参见：

- “服务器类 saodbc”一节第 759 页
- “服务器类 ulodbc”一节第 759 页
- “服务器类 aseodbc”一节第 759 页
- “服务器类 db2odbc”一节第 762 页
- “服务器类 oraodbc”一节第 764 页
- “服务器类 mssodbc”一节第 765 页
- “服务器类 msaccessodbc”一节第 768 页
- “服务器类 mysqlodbc”一节第 767 页
- “服务器类 adsodbc”一节第 761 页
- “服务器类 odbc”一节第 769 页

服务器类 saodbc

服务器类为 saodbc 的服务器是 SQL Anywhere 数据库服务器。对 SQL Anywhere 数据源的配置没有任何特殊要求。

要访问支持多个数据库的 SQL Anywhere 数据库服务器，可创建一个 ODBC 数据源名称，用以定义与每个数据库的连接。对创建的每个 ODBC 数据源名称发出 CREATE SERVER 语句。请参见“CREATE SERVER 语句中的 USING 参数”一节第 772 页。

服务器类 ulodbc

服务器类为 ulodbc 的服务器是 UltraLite 数据库。创建一个 ODBC 数据源名称，用以定义与 UltraLite 数据库的连接。对该 ODBC 数据源名称发出 CREATE SERVER 语句。

UltraLite 和 SQL Anywhere 数据类型之间存在一一对应关系，因为 UltraLite 支持一部分 SQL Anywhere 中可用的数据类型。请参见“UltraLite 中的数据类型”一节 《UltraLite - 数据库管理和参考》。

服务器类 aseodbc

服务器类为 aseodbc 的服务器是 Sybase SQL Server 和 Adaptive Server Enterprise（版本 10 及更高版本）数据库服务器。SQL Anywhere 需要安装 Adaptive Server Enterprise ODBC 驱动程序和 Open Client 连接库，才能连接到服务器类为 aseodbc 的远程 Adaptive Server Enterprise 服务器，但连接后的性能要优于服务器类为 asejdbc 的远程 Adaptive Server Enterprise 服务器。

注意

- Open Client 的版本应为 11.1.1, EBF 7886 或更高。在安装 ODBC 并配置 SQL Anywhere 之前，请先安装 Open Client 并验证与 Adaptive Server Enterprise 服务器的连接。Sybase ODBC 驱动程序的版本应为 11.1.1, EBF 7911 或更高。

- quoted_identifiers 选项的本地设置控制是否对 Adaptive Server Enterprise 使用加引号的标识符。例如，如果在本地将 quoted_identifiers 选项设置为 [关闭]，则会为 Adaptive Server Enterprise 关闭加引号的标识符。
- 在 [**Configuration Manager**] 中配置用户数据源的以下属性：
  - **[常规] 选项卡** 在 [**Data Source Name**] 中键入任意值。此值将在 CREATE SERVER 语句的 USING 子句中使用。

    服务器名应匹配 Sybase 接口文件中的服务器的名称。

    有关接口文件的详细信息，请参见“接口文件”一节 《SQL Anywhere 服务器 - 数据库管理》。
  - **[Advanced] 选项卡** 选择 [**Application Using Threads**] 和 [**Enable Quoted Identifiers**] 选项。
  - **[Connection] 选项卡** 设置字符集字段，使其与您的 SQL Anywhere 字符集匹配。

    将语言字段设置为您的用于显示错误消息的首选语言。
  - **[Performance] 选项卡** 将 [**Prepare Method**] 设置为 [**2-Full**]。

    将 [**Fetch Array Size**] 设置得尽可能大，以获得最佳性能。这会增加内存要求，因为该值是必须被高速缓存到内存中的行数。Adaptive Server Enterprise 建议使用值 100。

    将 [**Select Method**] 设置为 [**0-Cursor**]。

    将 [**Packet Size**] 设置为尽可能大的值。Adaptive Server Enterprise 建议使用值 -1。

    将 [**Connection Cache**] 设置为 1。

数据类型转换：ODBC 和 Adaptive Server Enterprise

当您发出 CREATE TABLE 语句后，SQL Anywhere 会自动将数据类型转换为对应的 Adaptive Server Enterprise 数据类型。下表列出了 SQL Anywhere 与 Adaptive Server Enterprise 之间的数据类型转换关系。

SQL Anywhere 数据类型	Adaptive Server Enterprise 缺省数据类型
BIT	bit
TINYINT	tinyint
SMALLINT	smallint
INT	int
INTEGER	integer
DECIMAL [缺省值 p=30, s=6]	numeric(30.6)
DECIMAL(128,128)	不支持
NUMERIC [缺省值 p=30 s=6]	numeric(30.6)

SQL Anywhere 数据类型	Adaptive Server Enterprise 缺省数据类型
NUMERIC(128,128)	不支持
FLOAT	real
REAL	real
DOUBLE	float
SMALLMONEY	numeric(10.4)
MONEY	numeric(19.4)
DATE	datetime
TIME	datetime
TIMESTAMP	datetime
SMALLDATETIME	datetime
DATETIME	datetime
CHAR(*n*)	varchar(*n*)
CHARACTER(*n*)	varchar(*n*)
VARCHAR(*n*)	varchar(*n*)
CHARACTER VARYING(*n*)	varchar(*n*)
LONG VARCHAR	text
TEXT	text
BINARY(*n*)	binary(*n*)
LONG BINARY	image
IMAGE	image
BIGINT	numeric(20,0)

服务器类 adsodbc

当您发出 CREATE TABLE 语句后，SQL Anywhere 会使用以下数据类型转换关系自动将数据类型转换为对应的 Advantage Database Server 数据类型。

SQL Anywhere 数据类型	ADS 缺省数据类型
BIT	Logical
TINYINT、SMALLINT、INT、INTEGER	Integer
BIGINT	Numeric(32)
DECIMAL(p,s)、NUMERIC(p,s)	Numeric(p+3)
DATE	Date
TIME	Time
DATETIME、TIMESTAMP	TimeStamp
MONEY、SMALLMONEY	Money
FLOAT、REAL	Double
CHAR(*n*)、VARCHAR(*n*)、LONG VARCHAR	Char(*n*)
BINARY(*n*)、VARBINARY(*n*)、LONG BINARY	Blob

服务器类 db2odbc

一种具有 db2odbc 服务器类的服务器是 IBM DB2。

注意

- 经 Sybase 认证，可以使用 IBM 的 DB2 Connect 版本 5（安装有补丁程序包 WR09044）。根据该产品的说明进行配置，并测试您的 ODBC 配置。SQL Anywhere 对 DB2 数据源的配置没有任何特殊要求。
- 以下是 CREATE EXISTING TABLE 语句的示例，该语句适用于一个 DB2 服务器，其 ODBC 数据源名为 mydb2：

```
CREATE EXISTING TABLE ibmcol
AT 'mydb2..sysibm.syscolumns';
```

数据类型转换：DB2

当您发出 CREATE TABLE 语句时，SQL Anywhere 会自动将数据类型转换为对应的 DB2 数据类型。下表列出了 SQL Anywhere 与 DB2 数据类型之间的转换关系。

SQL Anywhere 数据类型	DB2 缺省数据类型
BIT	smallint

SQL Anywhere 数据类型	DB2 缺省数据类型
TINYINT	smallint
SMALLINT	smallint
INT	int
INTEGER	int
BIGINT	decimal(20,0)
CHAR(1-254)	varchar(*n*)
CHAR(255-4000)	varchar(*n*)
CHAR(4001-32767)	long varchar
CHARACTER(1-254)	varchar(*n*)
CHARACTER(255-4000)	varchar(*n*)
CHARACTER(4001-32767)	long varchar
VARCHAR(1-4000)	varchar(*n*)
VARCHAR(4001-32767)	long varchar
CHARACTER VARYING(1-4000)	varchar(*n*)
CHARACTER VARYING(4001-32767)	long varchar
LONG VARCHAR	long varchar
TEXT	long varchar
BINARY(1-4000)	varchar for bit data
BINARY(4001-32767)	对于位数据为 long varchar
LONG BINARY	对于位数据为 long varchar
IMAGE	对于位数据为 long varchar
DECIMAL [缺省值 p=30, s=6]	decimal(30.6)
NUMERIC [缺省值 p=30 s=6]	decimal(30.6)
DECIMAL(128, 128)	不支持

SQL Anywhere 数据类型	DB2 缺省数据类型
NUMERIC(128, 128)	不支持
REAL	real
FLOAT	float
DOUBLE	float
SMALLMONEY	decimal(10,4)
MONEY	decimal(19.4)
DATE	date
TIME	time
SMALLDATETIME	timestamp
DATETIME	timestamp
TIMESTAMP	timestamp

服务器类 oraodbc

一种具有 oraodbc 服务器类的服务器是 Oracle 8.0 版或更高版本。

注意

- 经 Sybase 认证，可以使用 Oracle 8.0.03 版本的 ODBC 驱动程序。根据该产品的说明进行配置，并测试您的 ODBC 配置。
- 以下是名为 myora 的 Oracle 服务器的 CREATE EXISTING TABLE 语句的示例：

```
CREATE EXISTING TABLE employees
AT 'myora.database.owner.employees';
```

- 由于 Oracle ODBC 驱动程序的限制，您不能对系统表发出 CREATE EXISTING TABLE 语句。否则将返回一则消息，说明无法找到表或列。

数据类型转换：Oracle

当您发出 CREATE TABLE 语句后，SQL Anywhere 会使用以下数据类型转换关系自动将数据类型转换为对应的 Oracle 数据类型。

SQL Anywhere 数据类型	Oracle 数据类型
BIT	number(1,0)

SQL Anywhere 数据类型	Oracle 数据类型
TINYINT	number(3,0)
SMALLINT	number(5,0)
INT	number(11,0)
BIGINT	number(20,0)
DECIMAL(prec, scale)	number(prec, scale)
NUMERIC(prec, scale)	number(prec, scale)
FLOAT	float
REAL	real
SMALLMONEY	numeric(13,4)
MONEY	number(19,4)
DATE	date
TIME	date
TIMESTAMP	date
SMALLDATETIME	date
DATETIME	date
CHAR(*n*)	如果 (n > 255)，则为 long，否则为 varchar(*n*)
VARCHAR(*n*)	如果 (n > 2000)，则为 long，否则为 varchar(*n*)
LONG VARCHAR	long 或 clob
BINARY(*n*)	如果 (n > 255)，则为 long raw，否则为 raw(*n*)
VARBINARY(*n*)	如果 (n > 255)，则为 long raw，否则为 raw(*n*)
LONG BINARY	long raw

服务器类 mssodbc

一种具有 mssodbc 服务器类的服务器是 Microsoft SQL Server 6.5 版（安装 Service Pack 4）。

注意

- 经 Sybase 认证，可以使用 Microsoft SQL Server 的 ODBC 驱动程序 3.60.0319 版（包含在 MDAC 2.0 发行版本中）。根据该产品的说明进行配置，并测试您的 ODBC 配置。
- 以下是名为 mymssql 的 Microsoft SQL Server 的 CREATE EXISTING TABLE 语句示例：

```
CREATE EXISTING TABLE accounts,
AT 'mymssql.database.owner.accounts';
```

- quoted_identifiers 选项的本地设置控制是否对 Microsoft SQL Server 使用加引号的标识符。例如，如果在本地将 quoted_identifiers 选项设置为 [关闭]，则会为 Microsoft SQL Server 关闭加引号的标识符。

数据类型转换：Microsoft SQL Server

当您发出 CREATE TABLE 语句后，SQL Anywhere 会使用以下数据类型转换关系自动将数据类型转换为对应的 Microsoft SQL Server 数据类型。

SQL Anywhere 数据类型	Microsoft SQL Server 缺省数据类型
BIT	bit
TINYINT	tinyint
SMALLINT	smallint
INT	int
BIGINT	numeric(20,0)
DECIMAL [defaults p=30, s=6]	decimal(prec, scale)
NUMERIC [defaults p=30 s=6]	numeric(prec, scale)
FLOAT	if (prec) float(prec) else float
REAL	real
SMALLMONEY	smallmoney
MONEY	money
DATE	datetime
TIME	datetime
TIMESTAMP	datetime
SMALLDATETIME	datetime
DATETIME	datetime

SQL Anywhere 数据类型	Microsoft SQL Server 缺省数据类型
CHAR(*n*)	如果 (length > 255)，则为 text，否则为 varchar(length)
CHARACTER(*n*)	char(*n*)
VARCHAR(*n*)	如果 (length > 255)，则为 text，否则为 varchar(length)
LONG VARCHAR	text
BINARY(*n*)	如果 (length > 255)，则为 image，否则为 binary(length)
LONG BINARY	image
DOUBLE	float
UNIQUEIDENTIFIERSTR	uniqueidentifier

服务器类 mysqlodbc

当您发出 CREATE TABLE 语句后，SQL Anywhere 会使用以下数据类型转换关系自动将数据类型转换为对应的 MySQL 数据类型。

SQL Anywhere 数据类型	MySQL 缺省数据类型
BIT	bit(1)
TINYINT	tinyint unsigned
SMALLINT	smallint
INT、INTEGER	int
BIGINT	bigint
DECIMAL(p,s)、NUMERIC(p,s)	decimal(p,s)
DATE	date
TIME	time
DATETIME、TIMESTAMP	datetime
MONEY	decimal(19.4)
SMALLMONEY	decimal(10,4)

SQL Anywhere 数据类型	MySQL 缺省数据类型
FLOAT	float
REAL	real
CHAR(*n*)	如果 *n* 小于 254，则为 char(*n*) 如果 *n* 大于或等于 254 但小于 4000，则为 varchar(*n*) 如果 *n* 大于或等于 4000，则为 longtext
VARCHAR(*n*)	如果 *n* 小于 4000，则为 varchar(*n*) 如果 *n* 大于或等于 4000，则为 longtext
LONG VARCHAR	longtext
BINARY(*n*)、VARBINARY(*n*)	如果 *n* 小于 4000，则为 varbinary(*n*) 如果 *n* 大于或等于 4000，则为 longblob
LONG BINARY	longblob

服务器类 msaccessodbc

Access 数据库存储在 *.mdb* 文件中。使用 ODBC 管理器创建一个 ODBC 数据源，并将该数据源映射到一个这类文件。可通过 ODBC 管理器创建新的 *.mdb* 文件。如果在通过 SQL Anywhere 创建表时没有指定其它缺省文件，此数据库文件将成为缺省文件。

假定有一个名为 access 的 ODBC 数据源，则可以使用以下任何一个语句访问数据：

- ```
  CREATE TABLE tab1 (a int, b char(10))
  AT 'access...tab1';
  ```
- ```
  CREATE TABLE tab1 (a int, b char(10))
  AT 'access;d:\pcdb\data.mdb;;tab1';
  ```
- ```
  CREATE EXISTING TABLE tab1
  AT 'access;d:\pcdb\data.mdb;;tab1';
  ```

Access 不支持所有者名称限定，将其保留为空。

数据类型转换：Microsoft Access

SQL Anywhere 数据类型	Microsoft Access 缺省数据类型
BIT、TINYINT	TINYINT
SMALLINT	SMALLINT
INT、INTEGER	INTEGER
  ```

SQL Anywhere 数据类型	Microsoft Access 缺省数据类型
BIGINT	DECIMAL(19,0)
DECIMAL(p,s)、NUMERIC(p,s)	DECIMAL(p,s)
DATE、TIME、DATETIME、TIMESTAMP	DATETIME
MONEY、SMALLMONEY	MONEY
FLOAT	FLOAT
REAL	REAL
CHAR(*n*)、VARCHAR(*n*)	如果 *n* 小于 254，则为 CHARACTER(*n*) 如果 *n* 大于或等于 254，则为 TEXT
LONG VARCHAR	TEXT
BINARY、VARBINARY	如果 *n* 小于 4000，则为 BINARY(*n*) 如果 *n* 大于或等于 4000，则为 IMAGE
LONG BINARY	IMAGE

服务器类 odbc

没有自己的服务器类的 ODBC 数据源使用 **odbc** 服务器类。可以使用任何 ODBC 驱动程序。经 Sybase 认证，可以使用以下 ODBC 数据源：

- “Microsoft Excel (Microsoft 3.51.171300)”一节第 769 页
- “Microsoft FoxPro (Microsoft 3.51.171300)”一节第 770 页
- “Lotus Notes SQL 2.0”一节第 770 页

可通过 Microsoft Download Center 上的 Microsoft 数据访问组件（Microsoft Data Access Components，简称 MDAC）获取最新版本的 Microsoft ODBC 驱动程序。以上列出的 Microsoft 驱动程序版本是 MDAC 2.0 的一部分。

Microsoft Excel (Microsoft 3.51.171300)

对于 Excel，可以将每一个 Excel 工作簿在逻辑上视为包含若干个表的数据库。数据库中的表可以映射到 Excel 工作簿中的工作表。当您在 ODBC 驱动程序管理器中配置 ODBC 数据源名时，可以指定与该数据源相关联的缺省工作簿名称。不过，当您发出 CREATE TABLE 语句时，可以替换该缺省值并在位置字符串中指定工作簿名称。这样，您就可以只使用一个 ODBC DSN 来访问所有 Excel 工作簿。

在本示例中，创建了一个名为 excel 的 ODBC 数据源。要创建名为 *work1.xls* 的工作簿，其中包含称为 mywork 的工作表（或表），请执行以下语句：

```
CREATE TABLE mywork (a int, b char(20))
AT 'excel;d:\work1.xls;;mywork';
```

要创建第二个工作表（或表），请执行下面的语句：

```
CREATE TABLE mywork2 (x float, y int)
AT 'excel;d:\work1.xls;;mywork2';
```

您可以使用 CREATE EXISTING 语句将现有工作簿导入到 SQL Anywhere 中，前提是电子表格的第一行包含列名。

```
CREATE EXISTING TABLE mywork
AT'excel;d:\work1;;mywork';
```

如果 SQL Anywhere 报告未找到该表，则可能需要明确说明您想要映射到的列和行范围。例如：

```
CREATE EXISTING TABLE mywork
AT 'excel;d:\work1;;mywork$';
```

如果在工作表名中添加 $，则表示应选择整个工作表。

请注意，在 AT 指定的位置字符串中，使用分号而不是句点作为字段分隔符。这是因为句点出现在文件名中。Excel 不支持所有者名字段，因此请不要填写该字段。

Excel 也不支持删除操作。此外，由于 Excel 驱动程序不支持定位更新，因此可能无法进行某些更新。

Microsoft FoxPro (Microsoft 3.51.171300)

您可以将多个 FoxPro 表一起存储在一个 FoxPro 数据库文件 (*.dbc*) 中；也可以将每个表存储在各自的 *.dbf* 文件中。使用 *.dbf* 文件时，请确保文件名已填充到位置字符串中；否则，将使用 SQL Anywhere 的启动目录。

```
CREATE TABLE fox1 (a int, b char(20))
AT 'foxpro;d:\pcdb;;fox1';
```

如果您在 ODBC 驱动程序管理器中选择了 [**Free Table Directory**] 选项，则此语句会创建一个名为 *d:\pcdb\fox1.dbf* 的文件。

Lotus Notes SQL 2.0

可从 Lotus Web 站点 http://www.lotus.com/ 获得此驱动程序。请阅读附随的文档，以了解有关 Notes 数据与关系表映射关系的说明。可以轻松地将 SQL Anywhere 表映射到 Notes 表单。

下面介绍如何设置 SQL Anywhere 来访问 Address 示例文件。

- 使用 NotesSQL 驱动程序创建 ODBC 数据源。数据库将是示例名称文件：*c:\notes\data\names.nsf*。应启用 [**Map Special Characters**] 选项。对于本示例，[**数据源名**] 是 *my_notes_dsn*。

- 在 SQL Anywhere 中创建服务器：

```
CREATE SERVER names
CLASS 'odbc'
USING 'my_notes_dsn';
```

- 将 Person 表单映射到 SQL Anywhere 表中：

```
CREATE EXISTING TABLE Person
AT 'names...Person';
```

- 查询该表

```
SELECT * FROM Person;
```

避免出现要求输入口令的提示

Lotus Notes 不支持通过 ODBC API 发送用户名和口令。如果您尝试使用有口令保护的 ID 来访问 Lotus Notes，则运行 SQL Anywhere 的计算机上会出现一个窗口，提示您输入口令。在多用户服务器环境中应避免此行为。

要以无人照管的方式访问 Lotus Notes，而不会收到要求输入口令的提示，则必须使用不带口令保护的 ID。可以通过清除口令（选择 [**文件**] » [**工具**] » [**用户 ID**] » [**清除口令**]）来取消对 ID 的口令保护，除非 Domino 管理员要求在创建 ID 时设置口令。在此情况下，您将无法清除口令。

基于 JDBC 的服务器类

当 SQL Anywhere 在内部使用 Java 虚拟机和 jConnect 5.5 连接远程服务器时，使用的便是基于 JDBC 的服务器类。基于 JDBC 的服务器类如下：

- **sajdbc** SQL Anywhere。
- **asejdbc** Sybase SQL Server 和 Adaptive Server Enterprise（版本 10 和更高版本）。

有关基于 JDBC 的类的配置说明

在访问用基于 JDBC 的类定义的远程服务器时，请考虑以下因素：

- 为获得最优性能，建议采用基于 ODBC 的类（saodbc 或 aseodbc）。
- 必须将使用 asejdbc 或 sajdbc 服务器类进行访问的任何远程服务器都设置为可以处理基于 jConnect 6.x 的客户端。
- 如果 JDBC 远程服务器连接断开或丢失，则您只会在尝试使用 JDBC 远程服务器来访问代理对象（例如，代理表或代理过程）时，发现服务器不可用。ODBC 则没有此限制。

服务器类 sajdbc

服务器类为 sajdbc 的服务器是 SQL Anywhere 服务器。对于 SQL Anywhere 数据源，并没有任何特殊的配置要求。

CREATE SERVER 语句中的 USING 参数

必须对每个要访问的 SQL Anywhere 数据库单独运行 CREATE SERVER 语句。例如，如果名为 TestSA 的 SQL Anywhere 服务器在计算机 banana 上运行，并且该服务器拥有三个数据库（db1、db2、db3），则应这样配置本地 SQL Anywhere 数据库服务器：

```
CREATE SERVER TestSAdb1
CLASS 'sajdbc'
USING 'banana:2638/db1'
CREATE SERVER TestSAdb2
CLASS 'sajdbc'
USING 'banana:2638/db2'
CREATE SERVER TestSAdb3
CLASS 'sajdbc'
USING 'banana:2638/db3';
```

如果不指定 */database-name* 值，则远程连接使用远程 SQL Anywhere 缺省数据库。

有关详细信息，请参见“CREATE SERVER 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

服务器类 asejdbc

服务器类为 asejdbc 的服务器是 Sybase SQL Server 和 Adaptive Server Enterprise（版本 10 和更高版本）服务器。对 Adaptive Server Enterprise 数据源的配置没有任何特殊要求。

注意

- quoted_identifiers 选项的本地设置控制是否对 Adaptive Server Enterprise 使用加引号的标识符。例如，如果在本地将 quoted_identifiers 选项设置为 [关闭]，则会为 Adaptive Server Enterprise 关闭加引号的标识符。

数据类型转换：JDBC 和 Adaptive Server Enterprise

当您发出 CREATE TABLE 语句后，SQL Anywhere 会使用以下数据类型转换关系自动将数据类型转换为对应的 Adaptive Server Enterprise 数据类型。

SQL Anywhere 数据类型	Adaptive Server Enterprise 缺省数据类型
BIT	bit
TINYINT	tinyint
SMALLINT	smallint
INT	int
INTEGER	integer
DECIMAL [缺省值 p=30, s=6]	numeric(30.6)
DECIMAL(128,128)	不支持
NUMERIC [缺省值 p=30 s=6]	numeric(30.6)
NUMERIC(128,128)	不支持
FLOAT	real
REAL	real
DOUBLE	float
SMALLMONEY	numeric(10.4)
MONEY	numeric(19.4)
DATE	datetime
TIME	datetime

SQL Anywhere 数据类型	Adaptive Server Enterprise 缺省数据类型
TIMESTAMP	datetime
SMALLDATETIME	datetime
DATETIME	datetime
CHAR(*n*)	varchar(*n*)
CHARACTER(*n*)	varchar(*n*)
VARCHAR(*n*)	varchar(*n*)
CHARACTER VARYING(*n*)	varchar(*n*)
LONG VARCHAR	text
TEXT	text
BINARY(*n*)	binary(*n*)
LONG BINARY	image
IMAGE	image
BIGINT	numeric(19,0)

存储过程和触发器

本节介绍如何使用 SQL 存储过程和触发器在数据库中生成逻辑。在数据库中存储逻辑会使该逻辑可以自动供所有应用程序使用，从而带来一致性、性能和安全性等方面的益处。本节还介绍如何使用 SQL Anywhere 调试程序—一种用于调试各种类型逻辑的功能强大的工具。

使用过程、触发器和批处理

目录

过程和触发器概述

过程和触发器在数据库中存储过程 SQL 语句，以供所有应用程序使用。它们包括允许 SQL 语句的重复执行（LOOP 语句）和条件执行（IF 语句和 CASE 语句）的控制语句。批处理是作为一组提交到数据库服务器的 SQL 语句的集合。在过程和触发器中提供的许多功能（例如控制语句）在批处理中也提供。

过程通过 CALL 语句来调用，它们使用参数来接受值并将值返回给调用环境。通过将过程名包括在 SELECT 语句的 FROM 子句中，SELECT 语句也可以对过程结果集进行操作。

过程可以将结果集返回给调用者、调用其它过程或触发触发器。例如，用户定义的函数是一种将单个值返回到调用环境的存储过程。用户定义的函数不修改传递给它们的参数，而是拓宽了查询和其它 SQL 语句中可以使用的函数的范围。

触发器与特定数据库表相关联。只要有人插入、更新或删除关联表的行，触发器就会自动触发。触发器可以调用过程和触发其它触发器，但它们不具有任何参数并且无法由 CALL 语句调用。

SQL Anywhere 调试程序

可以使用 SQL Anywhere 调试程序来调试存储过程和触发器。请参见“调试过程、函数、触发器和事件”第 827 页。

可以在 Sybase Central 中分析存储过程，以便分析性能特性。请参见“使用系统过程进行过程分析”一节第 189 页。

过程和触发器的优点

过程和触发器增强了数据库的安全性，提高了效率并促进了标准化。

过程和触发器的定义在数据库中提供，与任何一个数据库应用程序相分离。这一分离具有多个优点。

标准化

过程和触发器使多个应用程序所执行的操作实现标准化。通过对操作进行一次编码并将编码存储于数据库中以供将来使用，应用程序只需调用过程或触发触发器即可反复获得预期结果。此外，因为更改只需在一个地点进行，所以，如果操作的实现方式发生更改，则使用该操作的所有应用程序都可以自动获取新功能。

高效

在网络数据库服务器环境中使用的过程和触发器无需通过网络通信即可访问数据库中的数据。这意味着，与在某一客户机的应用程序中实现的过程和触发器相比，数据库服务器环境中的过程和触发器的执行速度更快，对网络性能的影响更小。

当创建过程或触发器时，系统自动检查过程或触发器的语法是否正确，然后将它们存储在系统表中。任何应用程序第一次调用过程或触发触发器时，将过程或触发器从系统表编译到服务器的虚拟内存中，然后从那里执行它们。由于过程或触发器的一个副本在第一次执行后将保留在内存中，因此，可以立即重复执行同一过程或触发器。此外，多个应用程序可以同时使用某一过程或触发器，同一应用程序也可以递归地使用某一过程或触发器。

安全性

通过授予用户对表中他们不能直接检查或修改的数据具有一定限制的访问权限，过程和触发器提供了安全性。

例如，要执行触发器必须要有关联表所有者的表权限，但要触发触发器，具有在该表中插入、更新或删除行的权限的任何用户都可以做到。同样，过程（包括用户定义的函数）根据过程所有者的权限执行，但授予权限的任何用户都可以调用这些过程。这意味着过程和触发器可以具有（并通常具有）与调用它们的用户 ID 不同的权限。

过程简介

创建过程

在 Sybase Central 中，[**创建过程向导**] 提供了使用过程模板的选项。此外，还可以使用 Interactive SQL 执行 CREATE PROCEDURE 语句来创建过程。必须拥有 DBA 或 RESOURCE 权限才能创建过程。

♦ 创建新过程 (Sybase Central)

1. 以具有 DBA 或资源权限的用户身份连接到数据库。
2. 在左窗格中，双击 [**过程和函数**]。
3. 选择 [**文件**] » [**新建**] » [**过程**]。
4. 请按照 [**创建过程向导**] 中的说明进行操作。
5. 在右窗格中，单击 [**SQL**] 选项卡以填写过程代码。

   新过程即出现在 [**过程和函数**] 中。

示例

以下简单示例创建了过程 NewDepartment，用以对 SQL Anywhere 示例数据库的 Departments 表执行 INSERT 操作，从而创建新的部门。

```
CREATE PROCEDURE NewDepartment(
   IN id INT,
   IN name CHAR(35),
   IN head_id INT )
BEGIN
   INSERT
   INTO Departments ( DepartmentID,
       DepartmentName, DepartmentHeadID )
   VALUES ( id, name, head_id );
END;
```

过程的主体是一个复合语句。复合语句以 BEGIN 语句起始，以 END 语句结束。在 NewDepartment 的例子中，复合语句是包括在 BEGIN 和 END 语句之间的单个 INSERT。

过程的参数可以标记为 IN、OUT 或 INOUT 之一。缺省情况下，参数是 INOUT 参数。NewDepartment 过程的所有参数都是 IN 参数，因为该过程并不更改它们。如果参数不用来向调用者返回值，则应将参数设置为 IN。

临时过程

要创建临时过程，必须使用 CREATE TEMPORARY PROCEDURE 语句，该语句是 CREATE PROCEDURE 语句的扩展。临时过程不会永久存储在数据库中。它们会在连接结束或有明确删除指示时（以最先发生的为准）被删除。请参见“CREATE PROCEDURE 语句（Web 服务）”一节《SQL Anywhere 服务器 - SQL 参考》。

远程过程

要创建远程过程，必须至少拥有一个远程服务器。请参见：

- “创建远程过程”一节第 748 页
- “使用 Sybase Central 创建远程服务器”一节第 731 页

另请参见

- “SQL Anywhere 数据库连接” 《SQL Anywhere 服务器 - 数据库管理》
- “CREATE PROCEDURE 语句（Web 服务）”一节 《SQL Anywhere 服务器 - SQL 参考》
- “ALTER PROCEDURE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “使用复合语句”一节第 801 页
- “创建远程过程”一节第 748 页

变更过程

您可以使用 Sybase Central 或 Interactive SQL 修改现有过程。您必须具有 DBA 权限或者是过程的所有者。

在 Sybase Central 中，您不能直接重命名现有过程，而是必须用新名称创建一个新过程，将以前的代码复制到这一新过程中，然后删除旧过程。

在 Interactive SQL 中，可以执行 ALTER PROCEDURE 语句来修改现有过程。必须在此语句中包括完整的新过程（语法与用来创建该过程的 CREATE PROCEDURE 语句的语法相同）。

♦ 变更过程的代码 (Sybase Central)

1. 以具有 DBA 或资源权限的用户身份连接到数据库。
2. 在左窗格中，双击 [**过程和函数**]。
3. 选择某个过程。
4. 使用以下方法之一编辑该过程：
   - 在右窗格中，单击 [**SQL**] 选项卡。
   - 右击该过程并选择 [**在新建窗口中编辑**]。

     提示
     可以为每个过程单独打开一个窗口，并在过程之间复制代码。

   - 要添加或编辑过程注释，右击该过程并选择 [**属性**]。

     如果使用**数据库文档生成器**生成 SQL Anywhere 数据库文档，则可以选择在输出中包括这些注释。请参见“记录数据库”一节 《SQL Anywhere 服务器 - 数据库管理》。

另请参见

- “设置数据库对象的属性”一节第 14 页
- “授予针对过程的权限”一节 《SQL Anywhere 服务器 - 数据库管理》
- “撤消用户权限和特权”一节 《SQL Anywhere 服务器 - 数据库管理》
- “ALTER PROCEDURE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “CREATE PROCEDURE 语句（Web 服务）”一节 《SQL Anywhere 服务器 - SQL 参考》
- “创建过程”一节第 780 页
- “使用 Sybase Central 转换存储过程”一节第 632 页

调用过程

CALL 语句可调用过程。过程可以由一个应用程序调用，也可以由其它过程和触发器调用。

以下语句调用 NewDepartment 过程来插入 Eastern Sales 部门：

```
CALL NewDepartment( 210, 'Eastern Sales', 902 );
```

此调用结束后，可能要检查 Departments 表，以查看该新部门是否已经添加。

已被授予该过程的 EXECUTE 权限的所有用户都可以调用 NewDepartment 过程，即使他们对 Departments 表不具有任何权限。

调用返回结果集的过程的另一种方法是在查询中调用它。您可以对过程的结果集执行查询，并应用 WHERE 子句和其它 SELECT 功能来限制结果集。

```
SELECT t.ID, t.QuantityOrdered AS q
FROM ShowCustomerProducts( 149 ) t;
```

另请参见

- “数据库权限和特权概述”一节 《SQL Anywhere 服务器 - 数据库管理》
- “CALL 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “GRANT 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “FROM 子句”一节 《SQL Anywhere 服务器 - SQL 参考》

在 Sybase Central 中复制过程

在 Sybase Central 中，要在数据库间复制过程，请在左窗格中选择过程，然后将其拖到另一连接的数据库的 [**过程和函数**] 中。这样即创建了一个新的过程，并且原过程的代码已复制到其中。

只将过程代码复制到新过程中，并不复制其它过程属性（权限等）。只要为过程赋予新名称，过程也可以复制到同一数据库中。

删除过程

过程一旦创建即会一直保留在数据库中，直到有人显式将其删除。只有过程的所有者或具有 DBA 权限的用户才可以将其从数据库中删除。

♦ **删除过程 (Sybase Central)：**

1. 以 DBA 用户或过程所有者的身份连接到数据库。
2. 在左窗格中，双击 [**过程和函数**]。
3. 选择过程，然后选择 [**编辑**] » [**删除**]。
4. 单击 [**是**]。

♦ **删除过程 (SQL)**

1. 作为具有 DBA 权限的用户或作为过程的所有者连接到数据库。
2. 执行 DROP PROCEDURE 语句。

示例

以下语句从数据库中删除过程 NewDepartment：

```
DROP PROCEDURE NewDepartment;
```

另请参见

- “SQL Anywhere 数据库连接” 《SQL Anywhere 服务器 - 数据库管理》
- “DROP PROCEDURE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》

在参数中返回过程结果

过程通过以下方式之一将结果返回到调用环境：

- 单独的值作为 OUT 或 INOUT 参数返回。
- 结果集可被返回。
- 过程可以使用 RETURN 语句返回单个结果。

♦ **创建并运行此过程并显示其输出 (SQL)**

1. 使用 Interactive SQL，作为 DBA 连接到 SQL Anywhere 示例数据库。
2. 在 [SQL 语句] 窗格中，键入以下内容以创建过程 (AverageSalary)，该过程用于将雇员的平均薪水作为 OUT 参数返回：

```
CREATE PROCEDURE AverageSalary( OUT avgsal NUMERIC(20,3) )
BEGIN
   SELECT AVG( Salary )
   INTO avgsal
   FROM Employees;
END;
```

3. 创建用于保存过程输出值的变量。在此例中，输出变量为数值型，具有三位小数，因此按如下所示创建变量：

```
CREATE VARIABLE Average NUMERIC(20,3);
```

4. 使用所创建的用来保存结果的变量调用过程：

```
CALL AverageSalary( Average );
```

如果该过程已正确创建并运行，则 Interactive SQL [消息] 选项卡不会显示任何错误。

5. 要检查该变量的值，请执行以下语句：

```
SELECT Average;
```

查看输出变量 Average 的值。[**结果**] 窗格的 [**结果**] 选项卡会显示此变量的值 49988.623，这是雇员的平均薪水。

另请参见

- “SQL Anywhere 数据库连接” 《SQL Anywhere 服务器 - 数据库管理》

在结果集中返回过程结果

除了在单独的参数中将结果返回给调用环境外，过程还可以结果集的形式返回信息。结果集通常是查询的结果。以下过程返回包含指定部门中每个雇员的薪水的结果集：

```
CREATE PROCEDURE SalaryList( IN department_id INT )
RESULT ( "Employee ID" INT, Salary NUMERIC(20,3) )
BEGIN
   SELECT EmployeeID, Salary
    FROM Employees
    WHERE Employees.DepartmentID = department_id;
END;
```

如果 Interactive SQL 调用此过程，则 RESULT 子句中的姓名与查询的结果对应，用作所显示结果中的列标题。

要从 Interactive SQL 测试此过程，可以利用 CALL 语句调用该过程，并指定公司的某一部门。在 Interactive SQL 中，结果会出现在 [**结果**] 窗格的 [**结果**] 选项卡中。

示例

要列出研发部门（部门 ID 为 100）中雇员的薪水，请键入以下命令：

```
CALL SalaryList( 100 );
```

Employee ID	Salary
102	45700.000
105	62000.000
160	57490.000
243	72995.000
...	...

只有在 [**选项**] 窗口的 [**结果**] 选项卡上启用返回多个结果集选项，Interactive SQL 才能返回多个结果集。每个结果集在 [**结果**] 窗格中均出现在单独的选项卡上。

另请参见

- “从过程返回多个结果集”一节第 808 页

用户定义的函数简介

用户定义的函数是一种将单个值返回给调用环境的过程。本节介绍如何创建、使用和删除用户定义的函数。

注意
关于用户定义的函数是否是线程安全的，SQL Anywhere 不进行任何假设。这是应用程序开发人员的职责。

创建用户定义的函数

使用 CREATE FUNCTION 语句创建用户定义的函数。要执行此语句，必须具有 RESOURCE 权限。

以下简单示例创建一个函数，该函数连接两个字符串（用空格连在一起）以根据名字和姓氏建立全名。

```
CREATE FUNCTION FullName( FirstName CHAR(30),
   LastName CHAR(30) )
RETURNS CHAR(61)
BEGIN
   DECLARE name CHAR(61);
   SET name = FirstName || ' ' || LastName;
   RETURN ( name );
END;
```

CREATE FUNCTION 的语法与 CREATE PROCEDURE 语句的语法稍有不同。它们存在以下差异：

- 不需要 IN、OUT 或 INOUT 关键字，因为所有参数都是 IN 参数。
- 需要 RETURNS 子句来指定所返回的数据类型。
- 需要 RETURN 语句来指定所返回的值。

还可以通过 Sybase Central 创建用户定义的函数。

♦ 创建用户定义的函数 (Sybase Central)

1. 以具有 DBA 或资源权限的用户身份连接到数据库。
2. 在左窗格中，单击 [**过程和函数**]。
3. 选择 [**文件**] » [**新建**] » [**函数**]。
4. 请按照 [**创建函数向导**] 中的说明进行操作。
5. 在右窗格中，单击 [**SQL**] 选项卡以填写过程代码。

   新函数即出现在 [**过程和函数**] 中。

另请参见

- “CREATE FUNCTION 语句（Web 服务）”一节 《SQL Anywhere 服务器 - SQL 参考》

调用用户定义的函数

在您要使用内置的非集合函数的任何地方，都可以根据权限使用用户定义的函数。

以下语句在 Interactive SQL 中从包含名字和姓氏的两列返回全名：

```
SELECT FullName(GivenName, Surname)
 AS "Full Name"
 FROM Employees;
```

Full Name
Fran Whitney
Matthew Cobb
Philip Chin
...

以下语句在 Interactive SQL 中从提供的名字和姓氏返回全名：

```
SELECT FullName('Jane', 'Smith')
 AS "Full Name";
```

Full Name
Jane Smith

已被授予函数的 EXECUTE 权限的任何用户都可以使用 FullName 函数。

示例

以下用户定义的函数阐释变量的局部声明。

Customers 表包括加拿大和美国客户。用户定义的函数 Nationality 根据 Country 列生成一个 3 个字母组成的国家/地区代码。

```
CREATE FUNCTION Nationality( CustomerID INT )
RETURNS CHAR( 3 )
BEGIN
    DECLARE nation_string CHAR(3);
    DECLARE nation country_t;
    SELECT DISTINCT Country INTO nation
    FROM Customers
    WHERE ID = CustomerID;
    IF nation = 'Canada' THEN
            SET nation_string = 'CDN';
    ELSE IF nation = 'USA' OR nation = ' ' THEN
            SET nation_string = 'USA';
        ELSE
            SET nation_string = 'OTH';
        END IF;
    END IF;
RETURN ( nation_string );
END;
```

此示例声明 nation_string 变量来保存国籍字符串，使用 SET 语句来设置该变量的值，并且将 nation_string 字符串的值返回给调用环境。

以下查询列出 Customers 表中的所有加拿大客户：

```
SELECT *
FROM Customers
WHERE Nationality(ID) = 'CDN';
```

游标和异常的声明将在后面几节中讨论。

注意

尽管此函数适合于概念阐述，但在涉及多行的 SELECT 中使用时，其执行效果会非常差。例如，如果在某张表的查询的 SELECT 列表中使用该函数，该表包含 100,000 行，其中 10,000 行将被返回，则该函数将被调用 10,000 次。如果在同一查询的 WHERE 子句中使用该函数，则该函数会被调用 100,000 次。

删除用户定义的函数

用户定义的函数一旦创建即会一直保留在数据库中，直到有人显式地将其删除。只有函数的所有者或具有 DBA 权限的用户才可以将其从数据库中删除。

以下语句从数据库中删除 FullName 函数：

```
DROP FUNCTION FullName;
```

执行用户定义的函数的权限

用户定义的函数的所有权属于创建它的用户，因此该用户无需权限即可执行它。用户定义的函数的所有者可以使用 GRANT EXECUTE 命令向其他用户授予权限。

例如，函数 FullName 的创建者可以通过以下语句允许另一个用户使用 FullName：

```
GRANT EXECUTE ON Nationality TO BobS;
```

以下语句会撤消函数使用权限：

```
REVOKE EXECUTE ON Nationality FROM BobS;
```

请参见“授予针对过程的权限”一节 《SQL Anywhere 服务器 - 数据库管理》。

有关用户定义的函数的高级信息

SQL Anywhere 将所有用户定义的函数视为**等幂**函数，除非它们被声明为 NOT DETERMINISTIC。等幂函数为相同的参数返回一致的结果，并且没有副作用。以相同参数连续两次调用等幂函数会返回相同的结果，并且对查询的语义不会导致任何不良的副作用。

有关非确定性函数和确定性函数的详细信息，请参见“函数高速缓存”一节第 564 页。

触发器简介

触发器是一种特殊形式的存储过程，它在修改数据的语句执行时自动执行。只要参照完整性和其它声明性约束不足够，就可以使用触发器。请参见“确保数据完整性”第 73 页和“CREATE TABLE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

您可能要实施涉及更详细检查、形式更为复杂的参照完整性，或者对新数据实施检查，但允许旧数据违反约束。触发器的另一个用途是记录与数据库表有关的活动，而与使用数据库的应用程序无关。

注意
有三个特殊语句，触发器不在其后触发：LOAD TABLE、TRUNCATE 和 WRITETEXT。请参见“LOAD TABLE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》、“TRUNCATE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》和“WRITETEXT 语句 [T-SQL]”一节 《SQL Anywhere 服务器 - SQL 参考》。

触发器执行权限

要执行触发器必须要有关联表或视图所有者的权限，发出使触发器触发的操作的用户 ID 不能执行触发器。触发器可以修改用户无法直接修改的表中各行。

可以通过指定 -gf 服务器选项或设置 fire_triggers 选项防止触发触发器。请参见：

- “-gf 服务器选项”一节 《SQL Anywhere 服务器 - 数据库管理》
- “fire_triggers 选项 [兼容性]”一节 《SQL Anywhere 服务器 - 数据库管理》

触发器类型

SQL Anywhere 支持以下触发器类型：

- **BEFORE 触发器** BEFORE 触发器在执行触发操作之前触发。可为表定义 BEFORE 触发器，但不能为视图定义。
- **AFTER 触发器** AFTER 触发器在完成触发操作后触发。可为表定义 AFTER 触发器，但不能为视图定义。
- **INSTEAD OF 触发器** INSTEAD OF 触发器是替代触发操作触发的条件触发器。可为表和视图（实例化视图除外）定义 INSTEAD OF 触发器。请参见“INSTEAD OF 触发器”一节 第 795 页。

有关定义触发器的语法的完整说明，请参见“CREATE TRIGGER 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

触发事件

可以对以下一种或多种触发事件定义触发器：

操作	说明
INSERT	只要有新行插入到与触发器关联的表中，就调用触发器。

操作	说明
DELETE	只要删除了关联表中的行，就调用触发器。
UPDATE	只要更新了关联表中的行，就调用触发器。
UPDATE OF *column-list*	只要更新了关联表中的行，因而修改了 *column-list* 中的列，就调用触发器。

您可以为需要处理的每个事件分别编写触发器，如果您有一些共享操作和一些取决于事件的操作，也可以为所有事件创建一个触发器并使用 IF 语句辨别所发生的操作。请参见“触发器操作条件”一节 《SQL Anywhere 服务器 - SQL 参考》。

触发时间

触发器可以是**行级别**或**语句级别**：

- 行级别触发器为所更改的每一行执行一次。行级别触发器在更改行之前或之后执行。

  触发器通过变量获得受影响行新旧映像的列值。
- 语句级别触发器在整个触发语句完成后执行。触发器通过表示受触发语句影响行的新旧映像的临时表来获得这些行。SQL Anywhere 不支持语句级 BEFORE 触发器。

触发器执行时间灵活这一特点对于依赖于在执行时执行（或不执行）参照完整性操作（例如级联更新或删除）的触发器而言十分有用。

如果在触发器执行时发生错误，则触发该触发器的操作将失败。INSERT、UPDATE 和 DELETE 是原子操作。在这些操作失败时，该语句的所有结果（包括触发器的结果以及这些触发器调用的任何过程的结果）都恢复为其操作前状态。请参见“原子复合语句”一节第 801 页。

创建触发器

使用 Sybase Central 或 Interactive SQL 创建触发器。在 Sybase Central 中，您可以使用向导提供必要的信息。在 Interactive SQL 中，可以使用 CREATE TRIGGER 语句。对于这两个工具，必须具有 DBA 或 RESOURCE 权限才能够创建触发器，而且必须对与触发器关联的表具有 ALTER 权限。

触发器的主体由一个复合语句组成：一组用 BEGIN 和 END 括起来的 SQL 语句，中间用分号分隔。

不能在触发器内使用 COMMIT 和 ROLLBACK 以及某些 ROLLBACK TO SAVEPOINT 语句。

♦ 为给定表创建触发器 (Sybase Central)：

1. 以具有 DBA 或资源权限的用户身份连接到数据库。
2. 在左窗格中，单击 [**触发器**]。
3. 选择 [**文件**] » [**新建**] » [**触发器**]。
4. 请按照 [**创建触发器向导**] 中的说明进行操作。
5. 要填写代码，请在右窗格中单击 [**SQL**] 选项卡。

♦ **为给定表创建触发器 (SQL)：**

1. 以具有 DBA 权限的用户身份连接到数据库。还必须对与触发器关联的表具有 ALTER 权限。
2. 执行 CREATE TRIGGER 语句。

示例 1：行级 INSERT 触发器

以下触发器是一个行级 INSERT 触发器的示例。它检查为新雇员输入的出生日期是否合理：

```
CREATE TRIGGER check_birth_date
   AFTER INSERT ON Employees
REFERENCING NEW AS new_employee
FOR EACH ROW
BEGIN
     DECLARE err_user_error EXCEPTION
   FOR SQLSTATE '99999';
   IF new_employee.BirthDate > 'June 6, 2001' THEN
         SIGNAL err_user_error;
   END IF;
END;
```

注意
在 SQL Anywhere 示例数据库中，可能已经有一个名为 check_birth_date 的触发器。如果是这样，当尝试运行上述 SQL 语句时，将返回一个错误，指示触发器定义与现有触发器冲突。

此触发器在有任何行插入 Employees 表之后触发。它会检测并禁止对应出生日期晚于 2001 年 6 月 6 日的任何新行。

短语 REFERENCING NEW AS new_employee 使用别名 new_employee 以允许触发器代码中的语句引用新行中的数据。

发出错误信号将导致撤消触发语句以及之前所有的触发器操作结果。

对于将许多行添加到 Employees 表的 INSERT 语句，check_birth_date 触发器为每一新行触发一次。如果该触发器对任何行触发失败，则 INSERT 语句的所有结果都回退。

您可以通过将示例的第二行更改为以下内容，指定触发器在插入行之前而非插入行之后触发：

```
BEFORE INSERT ON Employees
```

REFERENCING NEW 子句引用该行的插入值；它与触发器的计时（之前或之后）无关。

您会发现，在某些情况下，使用声明的参照完整性或 CHECK 约束（而不是触发器）实施约束会更容易。例如，使用列检查约束来实现上例会更简洁、更高效：

```
CHECK (@col <= 'June 6, 2001')
```

示例 2：行级 DELETE 触发器示例

以下 CREATE TRIGGER 语句定义行级别 DELETE 触发器：

```
CREATE TRIGGER mytrigger
BEFORE DELETE ON Employees
REFERENCING OLD AS oldtable
FOR EACH ROW
BEGIN
   ...
END;
```

REFERENCING OLD 子句与触发器的计时（之前或之后）无关，它使删除触发器代码使用别名 oldtable 来引用所删除的行中的值。

示例 3：语句级 UPDATE 触发器示例

以下 CREATE TRIGGER 语句适合于语句级别 UPDATE 触发器：

```
CREATE TRIGGER mytrigger AFTER UPDATE ON Employees
REFERENCING NEW AS table_after_update
            OLD AS table_before_update
FOR EACH STATEMENT
BEGIN
   ...
END;
```

REFERENCING NEW 和 REFERENCING OLD 子句允许 UPDATE 触发器代码引用所更新的行的旧值和新值。表别名 table_after_update 引用新行中的列，表别名 table_before_update 引用旧行中的列。

对于语句级别触发器和行级别触发器，REFERENCING NEW 和 REFERENCING OLD 子句在意义上稍有不同。对于语句级别触发器，REFERENCING OLD 或 REFERENCING NEW 别名是表别名，而在行级别触发器中，它们引用所变更的行。

另请参见

- “SQL Anywhere 数据库连接” 《SQL Anywhere 服务器 - 数据库管理》
- “COMMIT 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “ROLLBACK TO SAVEPOINT 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “CREATE TRIGGER 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “使用复合语句”一节第 801 页

执行触发器

只要对触发器中指定的表执行了 INSERT、UPDATE 或 DELETE 操作，触发器就会自动执行。行级别触发器对受影响的每一行触发一次，而语句级别触发器则对整个语句触发一次。

在 INSERT、UPDATE 或 DELETE 触发触发器时，根据触发器类型（BEFORE 或 AFTER），操作的顺序如下所示：

1. 在触发器触发前。
2. 执行操作本身。
3. 执行参照操作。
4. 在触发器触发后。

> **注意**
> 使用 CREATE TRIGGER 语句创建触发器时，如果未指定触发器类型，则缺省类型为 AFTER。

如果其中任何步骤遇到在过程或触发器内未处理的错误，则前面的步骤被撤消，不执行随后的步骤，触发触发器的操作将失败。

变更触发器

可以使用 Sybase Central 或 Interactive SQL 修改现有触发器。但必须是触发器所对应表的所有者或 DBA 用户，或者必须对该表具有 ALTER 权限并具有 RESOURCE 权限。

在 Sybase Central 中，不能直接将现有触发器重命名，而是必须用新名称创建新的触发器，将以前的代码复制到这一新触发器中，然后删除旧触发器。

或者，也可以使用 ALTER TRIGGER 语句修改现有触发器。您必须在此语句中包括完整的新触发器（语法与用来创建该触发器的 CREATE TRIGGER 语句的语法相同）。

♦ 变更触发器的代码 (Sybase Central)

1. 作为具有 DBA 授权的用户或作为触发器的所有者连接到数据库。
2. 在左窗格中，双击 [**触发器**]。
3. 选择某个触发器。
4. 使用以下方法之一变更该触发器：
   - 在右窗格中，单击 [**SQL**] 选项卡。
   - 右击该触发器并选择 [**在新建窗口中编辑**]。

     提示
     可以为每个过程单独打开一个窗口，并在触发器之间复制代码。
   - 要添加或编辑过程注释，请右击该触发器并选择 [**属性**]。

     如果使用**数据库文档生成器**生成 SQL Anywhere 数据库文档，则可以选择在输出中包括这些注释。请参见“记录数据库”一节 《SQL Anywhere 服务器 - 数据库管理》。

♦ 变更触发器的代码 (SQL)：

1. 作为具有 DBA 授权的用户或作为触发器的所有者连接到数据库。
2. 执行 ALTER TRIGGER 语句。在此语句中包括完整的新触发器。

另请参见

- “设置数据库对象的属性”一节第 14 页
- “SQL Anywhere 数据库连接” 《SQL Anywhere 服务器 - 数据库管理》
- “使用 Sybase Central 转换存储过程”一节第 632 页
- “ALTER TRIGGER 语句”一节 《SQL Anywhere 服务器 - SQL 参考》

删除触发器

触发器一旦创建即会一直保留在数据库中，直到有人显式将其删除。您必须对与触发器关联的表具有 ALTER 权限，才可以删除该触发器。

♦ **删除触发器 (Sybase Central)：**

1. 作为具有 DBA 授权的用户或作为触发器的所有者连接到数据库。
2. 在左窗格中，双击 [**触发器**]。
3. 选择触发器，然后选择 [**编辑**] » [**删除**]。
4. 单击 [**是**]。

♦ **删除触发器 (SQL)：**

1. 作为具有 DBA 权限的用户或作为触发器的所有者连接到数据库。
2. 执行 DROP TRIGGER 语句。

示例

以下语句从数据库中删除 mytrigger 触发器：

```
DROP TRIGGER mytrigger;
```

另请参见

- “SQL Anywhere 数据库连接” 《SQL Anywhere 服务器 - 数据库管理》
- “DROP TRIGGER 语句”一节 《SQL Anywhere 服务器 - SQL 参考》

触发器执行权限

您不能授予触发器的执行权限，因为用户不能执行触发器：SQL Anywhere 会作为数据库操作的响应来触发触发器。不过，触发器确实有执行期间的相关权限，用于定义其执行某些操作的权利。

触发器使用定义了这些权限的表的所有者的权限（而不是导致触发器触发的用户的权限，并且不是创建该触发器的用户的权限）执行。

如果触发器引用某张表，在不显式指定所有者名称的情况下，触发器会使用表创建者的组成员资格来查找表。例如，如果 user_1.Table_A 上的触发器引用 Table_B，并且不指定 Table_B 的所有者，则 Table_B 必须是由 user_1 创建的，或者 user_1 必须（直接或间接地）是 Table_B 所有者组的成员。如果这两个条件都不具备，在该触发器触发时，数据库服务器将返回一条消息，指示未找到表。

此外，user_1 必须具有执行该触发器中所指定的操作的权限。

另请参见

- “数据库权限和特权概述”一节 《SQL Anywhere 服务器 - 数据库管理》

有关触发器的高级信息

关于触发器难以理解的一个方面是当多个触发器被同一触发操作所影响时，这些触发器的触发顺序。是否触发了触发器竞争以及它们被触发的顺序取决于两件事情：触发器类型（BEFORE、INSTEAD OF 或 AFTER）和触发器范围（行级或语句级）。

对于行级触发器，BEFORE 触发器早于 INSTEAD OF 触发器触发，而 INSTEAD OF 触发器早于 AFTER 触发器触发。给定行的所有行级触发器在任何触发器为后继行触发之前触发。

对于语句级触发器，INSTEAD OF 触发器早于 AFTER 触发器触发。不支持语句级 BEFORE 触发器。

如果有竞争的语句级 AFTER 触发器和行级 AFTER 触发器，则语句级 AFTER 触发器在所有行级触发器完成后触发。

如果有竞争的语句级 INSTEAD OF 触发器和行级 INSTEAD OF 触发器，则行级触发器不触发。

INSTEAD OF 触发器

INSTEAD OF 触发器不同于 BEFORE 和 AFTER 触发器，因为当 INSTEAD OF 触发器触发时，会跳过触发操作而执行指定的操作。

INSTEAD OF 触发器独有的功能和限制列表如下：

- 对于给定表中的每个触发事件，只能有一个 INSTEAD OF 触发器。
- 可为表或视图定义 INSTEAD OF 触发器。但不能对实例化视图定义 INSTEAD OF 触发器，因为在实例化视图上不能执行 DML 操作（例如 INSERT、DELETE 和 UPDATE 语句）。
- 当定义 INSTEAD OF 触发器时不能指定 ORDER 或 WHEN 子句。
- 不能为 UPDATE OF *column-list* 触发事件定义 INSTEAD OF 触发器。请参见“CREATE TRIGGER 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。
- INSTEAD OF 触发器是否执行递归取决于该触发器的目标是基表还是视图。对于视图，发生递归；对于基表，则不发生。就是说，如果 INSTEAD OF 触发器在定义触发器的基表上执行 DML 操作，这些操作不会使触发器触发（包括 BEFORE 或 AFTER 触发器）。如果目标为视图，则在该视图上执行的操作会触发所有触发器。
- 如果表中定义了 INSTEAD OF 触发器，则不能对该表执行带有 ON EXISTING 子句的 INSERT 语句。尝试此操作将返回 SQLE_INSTEAD_TRIGGER 错误。
- 不能对使用 WITH CHECK OPTION 定义（或嵌套在以此方式定义的其它视图中）的视图，以及包含对其定义的 INSTEAD OF INSERT 触发器的视图执行 INSERT 语句。对于 UPDATE 和 DELETE 语句，这点也同样适用。尝试此操作将返回 SQLE_CHECK_TRIGGER_CONFLICT 错误。
- 如果由于执行了定位更新、定位删除、PUT 语句或宽插入操作导致 INSTEAD OF 触发器被触发，则返回 SQLE_INSTEAD_TRIGGER_POSITIONED 错误。

使用 INSTEAD OF 触发器更新不可更新的视图

INSTEAD OF 触发器允许对非固有可更新视图执行 INSERT、UPDATE 或 DELETE 语句。触发器的主体定义执行相应 INSERT、UPDATE 或 DELETE 语句的意义。例如，假定您创建了以下视图：

```
CREATE VIEW V1 ( Surname, GivenName, State )
   AS SELECT DISTINCT Surname, GivenName, State
        FROM Contacts;
```

不能从 V1 中删除行，因为 DISTINCT 关键字使得 V1 为非固有可更新。换言之，该数据库服务器无法明确地确定从 V1 中删除行的意义。但可定义实现 V1 上的删除操作的 INSTEAD OF DELETE 触发器。例如，当从 V1 中删除该行时，下面的触发器将从 Contacts 中删除具有给定 Surname、GivenName 和 State 的所有行：

```
CREATE TRIGGER V1_Delete
  INSTEAD OF DELETE ON V1
  REFERENCING OLD AS old_row
  FOR EACH ROW
BEGIN
    DELETE FROM Contacts
      WHERE Surname = old_row.Surname
        AND GivenName = old_row.GivenName
        AND State = old_row.State
END;
```

定义 V1_Delete 触发器后，就可以从 V1 中删除行了。还可以定义其它 INSTEAD OF 触发器，以允许对 V1 执行 INSERT 和 UPDATE 语句。

如果具有 INSTEAD OF DELETE 触发器的视图嵌套在其它视图中，则为了检查 DELETE 的可更新性，它会象基表一样被处理。对于 INSERT 和 UPDATE 操作，这点也同样适用。继续先前的示例，创建另一个视图：

```
CREATE VIEW V2 ( Surname, GivenName ) AS
  SELECT Surname, GivenName from V1;
```

没有 V1_Delete 触发器，不能从 V2 中删除行，因为 V1 不是固有可更新的，因此 V2 也不是。但是，如果对 V1 定义 INSTEAD OF DELETE 触发器，则可以从 V2 中删除行。每个从 V2 中删除的行都会导致一个行被从 V1 中删除，从而触发 V1_Delete 触发器。

对嵌套视图定义 INSTEAD OF 触发器时要谨慎，因为这些触发器的触发可能产生意外的结果。要使目标行为变为显式，则对引用该嵌套视图的任何视图定义 INSTEAD OF 触发器。

可对 V2 定义以下触发器以产生 DELETE 语句的所需行为：

```
CREATE TRIGGER V2_Delete
  INSTEAD OF DELETE ON V2
  REFERENCING OLD AS old_row
  FOR EACH ROW
BEGIN
    DELETE FROM Contacts
      WHERE Surname = old_row.Surname
        AND GivenName = old_row.GivenName
END;
```

V2_Delete 触发器可确保对 V2 的删除操作保持不变，即使 V1 上的 INSTEAD OF DELETE 触发器被删除或更改。

批处理简介

批处理是一起提交并作为一个组依次执行的 SQL 语句组。过程中使用的控制语句（CASE、IF、LOOP 等）也可以在批处理中使用。如果批处理由用 BEGIN/END 括起来的复合语句组成，则它还可以包含主机变量以及变量、游标、临时表或异常的本地声明。允许在批处理中引用主机变量，但具有以下限制：

- 批处理中只有一个语句可以引用主机变量
- 使用主机变量的语句之前不能出现返回结果集的语句

正在使用批处理时，建议使用 BEGIN/END 进行清晰地指示。

批处理内的语句可以使用分号分隔，此种情况下批处理符合 Watcom-SQL 方言。不能使用分号分隔语句的多语句批处理符合 Transact-SQL 方言。批处理的方言决定在批处理中允许使用哪些语句，还决定如何处理批处理中错误。有关 Transact-SQL 批处理的详细信息，请参见“Transact-SQL 批处理概述”一节第 631 页。

在许多方面，批处理类似于存储过程；但是，存在一些差异：

- 批处理没有名称
- 批处理不接收参数
- 批处理不是永久存储在数据库中
- 批处理不能由不同的连接共享

简单的批处理由一组不带分隔符的 SQL 语句组成，后面跟有只含 go 一词的单独一行。以下示例会创建 Eastern Sales 部门并将所有销售代表从麻萨诸塞州转移到该部门。以下是一个 Transact-SQL 批处理示例。

```
INSERT
INTO Departments ( DepartmentID, DepartmentName )
VALUES ( 220, 'Eastern Sales' )

UPDATE Employees
SET DepartmentID = 220
WHERE DepartmentID = 200
AND State = 'MA'

COMMIT
go
```

go 一词由 Interactive SQL 识别，让其将前面的语句作为单个批处理发送到服务器。请参见“执行多条 SQL 语句”一节 《SQL Anywhere 服务器 - 数据库管理》。

以下示例尽管看起来与上面的很相似，但其实 Interactive SQL 会以截然不同的方式进行处理。此示例不使用 Transact-SQL 方言。每个语句以分号分隔。Interactive SQL 将每个以分号分隔的语句分别发送到服务器。它不会被视为批处理。

```
INSERT
INTO Departments ( DepartmentID, DepartmentName )
VALUES ( 220, 'Eastern Sales' );

UPDATE Employees
```



```
SET DepartmentID = 220
WHERE DepartmentID = 200
AND State = 'MA';

COMMIT;
```

要使 Interactive SQL 将其视为批处理，可将其改为使用 BEGIN ... END 的复合语句。以下是前一个示例的修订版。复合语句中的三个语句会作为批处理发送给服务器。

```
BEGIN
  INSERT
  INTO Departments ( DepartmentID, DepartmentName )
  VALUES ( 220, 'Eastern Sales' );

  UPDATE Employees
  SET DepartmentID = 220
  WHERE DepartmentID = 200
  AND State = 'MA';

  COMMIT;
END
```

在这个特殊的示例中，不论服务器执行批处理还是单个语句，最终结果都没有区别。但也存在结果不一样的情况。请看以下示例。

```
DECLARE @CurrentID INTEGER;
SET @CurrentID = 207;
SELECT Surname FROM Employees
  WHERE EmployeeID=@CurrentID;
```

如果使用 Interactive SQL 执行此示例，则数据库服务器会返回一个错误，指示无法找到变量。发生这种情况是因为 Interactive SQL 将三个语句分别发送到服务器。它们不是作为批处理来执行。正如您已经看到的那样，这一问题的解决方法是使用复合语句来强制 Interactive SQL 将这些语句作为批处理发送给服务器。以下示例实现了这一目的。

```
BEGIN
  DECLARE @CurrentID INTEGER;
  SET @CurrentID = 207;
  SELECT Surname FROM Employees
    WHERE EmployeeID=@CurrentID;
END
```

用 BEGIN 和 END 将一组语句括起来可强制 Interactive SQL 将它们视作批处理。

IF 语句是复合语句的另一个例子。Interactive SQL 将以下语句作为单个批处理发送给服务器。

```
IF EXISTS(   SELECT *
             FROM SYSTAB
             WHERE table_name='Employees' )
THEN
   SELECT    Surname AS LastName,
             GivenName AS FirstName
   FROM Employees;
   SELECT Surname, GivenName
   FROM Customers;
   SELECT Surname, GivenName
   FROM Contacts;
ELSE
   MESSAGE 'The Employees table does not exist'
   TO CLIENT;
END IF
```


使用其它技术准备和执行 SQL 语句时不会发生这种情况。例如，使用 ODBC 的应用程序可以将一系列用分号分隔的语句当作批处理来准备和执行。

对服务器混用 Interactive SQL 语句和 SQL 语句时必须小心。以下的示例说明混用 Interactive SQL 语句和 SQL 语句会引发问题。在此示例中，由于 Interactive SQL OUTPUT 语句嵌入到复合语句中，因此它与所有其它语句一起作为批处理发送给服务器，结果导致语法错误。

```
IF EXISTS(   SELECT *
             FROM SYSTAB
             WHERE table_name='Employees' )
THEN
   SELECT    Surname AS LastName,
             GivenName AS FirstName
   FROM Employees;
   SELECT Surname, GivenName
   FROM Customers;
   SELECT Surname, GivenName
   FROM Contacts;
   OUTPUT TO 'c:\\temp\\query.txt';
ELSE
   MESSAGE 'The Employees table does not exist'
   TO CLIENT;
END IF
```

OUTPUT 语句的正确用法如下所示。

```
IF EXISTS(   SELECT *
             FROM SYSTAB
             WHERE table_name='Employees' )
THEN
   SELECT    Surname AS LastName,
             GivenName AS FirstName
   FROM Employees;
   SELECT Surname, GivenName
   FROM Customers;
   SELECT Surname, GivenName
   FROM Contacts;
ELSE
   MESSAGE 'The Employees table does not exist'
   TO CLIENT;
END IF;
OUTPUT TO 'c:\\temp\\query.txt';
```

控制语句

一些控制语句可以用在过程或触发器的主体或在批处理中来实现逻辑流和决策。可用的控制语句包括：

控制语句	语法

复合语句
请参见“BEGIN 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。
	`BEGIN [ ATOMIC ]`
	`  Statement-list`
	`END`

条件执行：IF
请参见“IF 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。
	`IF condition THEN`
	`  Statement-list`
	`ELSEIF condition THEN`
	`  Statement-list`
	`ELSE`
	`  Statement-list`
	`END IF`

条件执行：CASE
请参见“CASE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。
	`CASE expression`
	`WHEN value THEN`
	`  Statement-list`
	`WHEN value THEN`
	`  Statement-list`
	`ELSE`
	`  Statement-list`
	`END CASE`

重复：WHILE、LOOP
请参见“LOOP 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。
	`WHILE condition LOOP`
	`  Statement-list`
	`END LOOP`

重复：FOR 游标循环
请参见“FOR 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。
	`FOR loop-name`
	`  AS cursor-name CURSOR FOR`
	`  select-statement`
	`DO`
	`  Statement-list`
	`END FOR`

中断：LEAVE 请参见“LEAVE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。	`LEAVE label`
CALL 请参见“CALL 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。	`CALL procname( arg, ... )`

使用复合语句

复合语句以关键字 BEGIN 起始，以关键字 END 结束。过程或触发器的主体是**复合语句**。复合语句还可以用于批处理。复合语句可以嵌套，可以与其它控制语句组合在一起，以定义过程和触发器中或批处理中的执行流。

复合语句允许将一组 SQL 语句组合在一起作为一个单元处理。复合语句内的 SQL 语句以分号分隔。

有关复合语句的详细信息，请参见“BEGIN 语句”一节 《SQL Anywhere 服务器 - SQL 参考》。

复合语句中的声明

复合语句中的局部声明紧随在 BEGIN 关键字之后。这些局部声明只存在于复合语句内。在复合语句内，您可以声明：

- 变量
- 游标
- 临时表
- 异常（错误标识符）

局部声明可由该复合语句中的任何语句引用，或者可由该复合语句内嵌套的任何复合语句中的任何语句引用。局部声明对于从复合语句调用的其它过程而言是不可见的。

原子复合语句

原子语句是或者完全执行，或者根本不执行的语句。例如，更新数千行的 UPDATE 语句在更新许多行后可能会遇到错误。如果该语句没有完成，则所有更改的行都恢复为其初始状态。UPDATE 语句即是原子语句。

所有非复合 SQL 语句都是原子语句。可以通过在 BEGIN 关键字后添加 ATOMIC 关键字来使复合语句成为原子语句。

```
BEGIN ATOMIC
   UPDATE Employees
   SET ManagerID = 501
   WHERE EmployeeID = 467;
    UPDATE Employees
   SET BirthDate = 'bad_data';
END
```

在此示例中，两个更新语句都属于原子复合语句。它们要么成功，要么失败，二者只能居其一。第一个更新语句将是成功的。第二个语句会引发数据转换错误，因为指派给 BirthDate 列的值无法转换为日期。

该原子复合语句会失败，因此两个 UPDATE 语句的结果都会被撤消。即使当前执行的事务最终被提交，原子复合语句中的这两个语句也都不会生效。

如果原子复合语句成功，则在复合语句中进行的更改直到当前执行的事务被提交的时候才会生效。如果原子复合语句成功而包含该语句的事务却回退，则原子复合语句也回退。在原子复合语句开始时建立一个保存点。语句中的任何错误都会导致回退到该保存点。

在自动提交（非链接）模式下执行原子复合语句时，直到完成语句执行后提交模式才变为到手动（链接）。在手动模式中，在原子复合语句中执行的 DML 语句不会引起快速 COMMIT 或 ROLLBACK。如果原子复合语句成功完成，将执行 COMMIT 语句；否则执行 ROLLBACK 语句。有关自动提交行为的详细信息，请参见“设置自动提交或手工提交模式”一节 《SQL Anywhere 服务器 - 编程》和“控制自动提交行为”一节 《SQL Anywhere 服务器 - 编程》。

在原子复合语句中不能使用 COMMIT 和 ROLLBACK 以及某些 ROLLBACK TO SAVEPOINT 语句。请参见“过程和触发器中的事务和保存点”一节第 822 页。

有一种情况下会只执行原子复合语句中的某些语句（而不是全部语句）。那就是当复合语句中的异常处理程序处理错误的时候。

有关详细信息，请参见“在过程和触发器中使用异常处理程序”一节第 818 页。

过程和触发器的结构

过程或触发器的主体由“使用复合语句”一节第 801 页中所讨论的复合语句组成。复合语句由 BEGIN 和 END 组成，它们将一组 SQL 语句括起来，每个语句以分号分隔。

为过程声明参数

过程参数在 CREATE PROCEDURE 语句中以列表形式出现。参数名必须符合其它数据库标识符（如列名）的规则。它们必须具有有效数据类型（请参见“SQL 数据类型”《SQL Anywhere 服务器 - SQL 参考》），并且可以用关键字 IN、OUT 或 INOUT 作为前缀。缺省情况下，参数是 INOUT 参数。这些关键字具有以下含义：

- **IN** 此参数是一个为过程提供值的表达式。
- **OUT** 此参数是一个可由过程为其赋值的变量。
- **INOUT** 此参数是一个为过程提供值的变量，并且可以由过程为其赋予新值。

可以在 CREATE PROCEDURE 语句中将缺省值指派给过程参数。缺省值必须是常量，可以是空值。例如，以下过程对一个 IN 参数使用 NULL 缺省值，来避免执行没有任何意义的查询：

```
CREATE PROCEDURE CustomerProducts(
      IN customer_ID
                  INTEGER DEFAULT NULL )
RESULT ( product_ID INTEGER,
         quantity_ordered INTEGER )
BEGIN
   IF customer_ID IS NULL THEN
      RETURN;
   ELSE
      SELECT    Products.ID,
                sum( SalesOrderItems.Quantity )
        FROM    Products,
                  SalesOrderItems,
                SalesOrders
      WHERE SalesOrders.CustomerID = customer_ID
      AND SalesOrders.ID = SalesOrderItems.ID
      AND SalesOrderItems.ProductID = Products.ID
      GROUP BY Products.ID;
   END IF;
END;
```

以下语句指派 DEFAULT NULL，当遇到该值时，过程会返回而不执行查询。

```
CALL CustomerProducts();
```

将参数传递给过程

您可以通过 CALL 语句的两种形式之一利用存储过程参数的缺省值的优越性。

位于 CREATE PROCEDURE 语句参数列表末尾的可选参数可以在 CALL 语句中省略。例如，假定有以下具有三个 INOUT 参数的过程：

```
CREATE PROCEDURE SampleProcedure(
       INOUT var1 INT DEFAULT 1,
                  INOUT var2 int DEFAULT 2,
                  INOUT var3 int DEFAULT 3 )
...
```

此示例假定调用环境已设置了三个变量来保存传递给过程的值：

```
CREATE VARIABLE V1 INT;
CREATE VARIABLE V2 INT;
CREATE VARIABLE V3 INT;
```

调用过程 SampleProcedure 时可以只提供第一个参数，如下所示：

```
CALL SampleProcedure( V1 );
```

这时，会对 *var2* 和 *var3* 使用缺省值。

以可选参数调用过程的更灵活方法是按名称传递这些参数。可按如下所示调用 SampleProcedure 过程：

```
CALL SampleProcedure( var1 = V1, var3 = V3 );
```

或按如下所示调用该过程：

```
CALL SampleProcedure( var3 = V3, var1 = V1 );
```

将参数传递给函数

用户定义的函数不通过 CALL 语句进行调用，而是采用与内置函数同样的方式使用。例如，以下语句使用在“创建用户定义的函数”一节第 786 页中所定义的 FullName 函数来检索雇员的姓名：

♦ **列出所有雇员的姓名：**

● 在 Interactive SQL 中，键入以下内容：

```
SELECT FullName( GivenName, Surname ) AS Name
   FROM Employees;
```

Name
Fran Whitney
Matthew Cobb
Philip Chin
Julie Jordan
...

注意

- 在调用函数中可以使用缺省参数。但不能按名称将参数传递给函数。
- 参数是按值而不是按引用传递的。即使函数更改了参数的值，这一更改也不会返回给调用环境。
- 不能在用户定义的函数中使用输出参数。
- 用户定义的函数不能返回结果集。

从过程返回结果

过程可以以一行或多行数据的形式返回结果。由单行数据组成的结果可作为参数传递回过程。由多行数据组成的结果作为结果集传递回过程。过程也可以返回在 RETURN 语句中给定的单个值。

有关如何从过程返回结果的简单示例，请参见“过程简介”一节第 780 页。

使用 RETURN 语句返回值

RETURN 语句会将单个整数值返回给调用环境，并导致程序立即从过程退出。RETURN 语句采用以下形式：

```
RETURN expression
```

所提供表达式的值被返回给调用环境。要将返回值保存在变量中，请使用 CALL 语句的扩展：

```
CREATE VARIABLE returnval INTEGER;
returnval = CALL myproc();
```

作为过程参数返回结果

过程可以将结果作为作用于该过程的参数返回到调用环境中。

在过程内，可以采用以下方式为参数和变量指派值：

- SET 语句。
- 具有 INTO 子句的 SELECT 语句。

使用 SET 语句

以下过程使用 SET 语句将值返回到指派的 OUT 参数中：

```
CREATE PROCEDURE greater(
   IN a INT,
   IN b INT,
   OUT c INT )
BEGIN
   IF a > b THEN
      SET c = a;
   ELSE
      SET c = b;
   END IF ;
END;
```

使用单行 SELECT 语句

单行查询会从数据库中最多检索一行数据。此类型的查询将 SELECT 语句与 INTO 子句一起使用。INTO 子句跟随在选择列表之后，位于 FROM 子句之前。它包含一组变量，用于为每一选择列表项检索值。变量的数目必须与选择列表项的数目相同。

在执行 SELECT 语句时，服务器会检索 SELECT 语句的结果并将这些结果放置在变量中。如果查询结果包含多行，则服务器返回错误。对于返回多行的查询，必须使用游标。有关从过程返回多行的信息，请参见“从过程返回结果集”一节第 807 页。

如果通过查询没有选择任何一行，则会返回警告。

以下过程在过程参数中返回单行 SELECT 语句的结果。

♦ **返回给定客户下的订单的数目：**

● 键入以下语句：

```
CREATE PROCEDURE OrderCount(
   IN customer_ID INT,
   OUT Orders INT )
BEGIN
   SELECT COUNT(SalesOrders.ID)
      INTO Orders
   FROM Customers
      KEY LEFT OUTER JOIN SalesOrders
   WHERE Customers.ID = customer_ID;
END;
```

可以使用以下语句在 Interactive SQL 中测试此过程，将会显示 ID 为 102 的客户所下订单的数目：

```
CREATE VARIABLE orders INT;
CALL OrderCount ( 102, orders );
SELECT orders;
```

注意

- customer_ID 参数被声明为 IN 参数。此参数会保存传递给过程的客户 ID。
- Orders 参数被声明为 OUT 参数。用于保存返回给调用环境的订单数变量的值。
- Orders 变量不需要 DECLARE 语句，因为它是在过程参数列表中声明的。
- SELECT 语句返回单个行并将其放置在变量 Orders 中。

从过程返回结果集

结果集允许过程将多行结果返回到调用环境中。

以下过程会返回已下订单的客户及其所下订单总值的列表。此过程不会列出未下订单的客户。

```
CREATE PROCEDURE ListCustomerValue()
RESULT ("Company" CHAR(36), "Value" INT)
BEGIN
   SELECT CompanyName,
      CAST( sum(  SalesOrderItems.Quantity *
                  Products.UnitPrice)
                  AS INTEGER ) AS value
   FROM Customers
      INNER JOIN SalesOrders
      INNER JOIN SalesOrderItems
      INNER JOIN Products
   GROUP BY CompanyName
```

```
    ORDER BY value DESC;
END;
```

- 键入以下语句：

```
CALL ListCustomerValue ( );
```

Company	Value
The Hat Company	5016
The Igloo	3564
The Ultimate	3348
North Land Trading	3144
Molly's	2808
...	...

注意

- RESULT 列表中变量的数目必须与 SELECT 列表项的数目相匹配。如果数据类型不匹配，会尽可能执行自动的数据类型转换。
- RESULT 子句是 CREATE PROCEDURE 语句的一部分，并且不具有命令分隔符。
- SELECT 列表项的名称不必与 RESULT 列表中的那些名称相匹配。
- 在测试此过程时，缺省情况下 Interactive SQL 只显示第一个结果集。可以通过在 [**选项**] 窗口的 [**结果**] 选项卡上设置 [**显示多个结果集**] 选项，将 Interactive SQL 配置为可显示多个结果集。
- 除非过程结果集是从视图生成的，否则可以对它们进行修改。调用该过程的用户要求对基表具有适当的权限，以修改过程结果。这与通常的过程执行权限（要求过程所有者必须对该表具有权限）不同。请参见“在 Interactive SQL 中编辑结果集”一节 《SQL Anywhere 服务器 - 数据库管理》 。
- 如果存储过程或用户定义的函数返回一个结果集，则不能同时设置输出参数或返回一个返回值。

从过程返回多个结果集

要使 Interactive SQL 可以返回多个结果集，需要先在 [**选项**] 窗口的 [**结果**] 选项卡上启用此选项。缺省情况下，此选项被禁用。此设置的更改会在新创建的连接（例如新窗口）中生效。

♦ 启用多个结果集功能 (Sybase Central)

1. 以具有 DBA 权限的用户身份连接到数据库。
2. 在左窗格中，选择数据库，然后选择 [**文件**] » [**打开 Interactive SQL**]。
3. 在 Interactive SQL 中，选择 [**工具**] » [**选项**]。

4. 单击 [**SQL Anywhere**]。
5. 在 [**结果**] 选项卡上，选择 [**显示所有结果集**]。
6. 单击 [**确定**]。

启用此选项后，过程可以将多个结果集返回给调用环境。如果采用 RESULT 子句，结果集必须是兼容的：项目数必须与 SELECT 列表中的项目数相同，数据类型必须全都可以自动转换为 RESULT 列表中所列的数据类型。

示例

以下过程列出在数据库中列出的所有雇员、客户和联系人的姓名：

```
CREATE PROCEDURE ListPeople()
RESULT ( Surname CHAR(36), GivenName CHAR(36) )
BEGIN
   SELECT Surname, GivenName
   FROM Employees;
   SELECT Surname, GivenName
   FROM Customers;
   SELECT Surname, GivenName
   FROM Contacts;
END;
```

要在 Interactive SQL 中测试此过程并查看多个结果集，请在 [SQL 语句] 窗格中输入以下语句：

```
CALL ListPeople ();
```

从过程返回可变结果集

RESULT 子句在过程中是可选的。通过省略 RESULT 子句，可以编写返回不同结果集的过程，结果集中可以有不同数目或类型的列，具体情况取决于过程的执行方式。

如果不使用可变结果集功能，则出于性能方面的原因，应使用 RESULT 子句。

例如，如果输入变量为 Y，以下过程返回两列，否则只返回一列。

```
CREATE PROCEDURE Names( IN formal char(1) )
BEGIN
   IF formal = 'y' THEN
      SELECT Surname, GivenName
      FROM Employees
   ELSE
      SELECT GivenName
      FROM Employees
   END IF
END;
```

过程中可变结果集的使用受到某些限制，具体限制取决于客户端应用程序所使用的接口。

- **嵌入式 SQL**　要获取正确形式的结果集，必须在打开用于结果集的游标之后、但在返回任何行之前对过程调用执行 DESCRIBE。

  有关 DESCRIBE 语句的详细信息，请参见“DESCRIBE 语句 [Interactive SQL]”一节 《SQL Anywhere 服务器 - SQL 参考》。

- **ODBC** ODBC 应用程序可以使用可变结果集过程。SQL Anywhere ODBC 驱动程序会对可变结果集进行适当的描述。
- **Open Client 应用程序** Open Client 应用程序可以使用可变结果集过程。SQL Anywhere 会对可变结果集进行适当的描述。

在过程和触发器中使用游标

游标会逐行检索通过结果集中含多行数据的查询或存储过程所获得的数据行，一次一行。游标是查询或过程的句柄或标识符，也是结果集内的当前位置的句柄或标识符。

游标管理概述

管理游标类似于通过编程语言管理文件。游标的管理步骤如下：

1. 使用 DECLARE 语句为特殊 SELECT 语句或过程声明游标。
2. 使用 OPEN 语句打开游标。
3. 使用 FETCH 语句从游标一次检索一行结果。
4. 警告 [**未找到行**] 说明已到达结果集的结尾。
5. 使用 CLOSE 语句关闭游标。

缺省情况下，游标会在事务结尾处（COMMIT 或 ROLLBACK 语句上）自动关闭。而使用 WITH HOLD 子句所打开的游标对随后的事务也会保持打开状态，直至被显式关闭。

有关游标定位的详细信息，请参见“游标定位”一节 《SQL Anywhere 服务器 - 编程》。

在过程中在 SELECT 语句上使用游标

以下过程在 SELECT 语句上使用游标。该过程基于在“从过程返回结果集”一节第 807 页所述 ListCustomerValue 过程中使用的同一查询，说明了存储过程语言的几个特征。

```
CREATE PROCEDURE TopCustomerValue(
      OUT TopCompany CHAR(36),
      OUT TopValue INT )
BEGIN
   -- 1. Declare the "row not found" exception
   DECLARE err_notfound
      EXCEPTION FOR SQLSTATE '02000';
   -- 2.  Declare variables to hold
   --     each company name and its value
   DECLARE ThisName CHAR(36);
   DECLARE ThisValue INT;
   -- 3.  Declare the cursor ThisCompany
   --     for the query
   DECLARE ThisCompany CURSOR FOR
   SELECT CompanyName,
         CAST( sum( SalesOrderItems.Quantity *
               Products.UnitPrice ) AS INTEGER )
         AS value
   FROM Customers
      INNER JOIN SalesOrders
      INNER JOIN SalesOrderItems
      INNER JOIN Products
   GROUP BY CompanyName;
   -- 4. Initialize the values of TopValue
```

```
    SET TopValue = 0;
    -- 5. Open the cursor
    OPEN ThisCompany;
    -- 6. Loop over the rows of the query
    CompanyLoop:
    LOOP
       FETCH NEXT ThisCompany
          INTO ThisName, ThisValue;
       IF SQLSTATE = err_notfound THEN
          LEAVE CompanyLoop;
       END IF;
       IF ThisValue > TopValue THEN
          SET TopCompany = ThisName;
          SET TopValue = ThisValue;
       END IF;
    END LOOP CompanyLoop;
    -- 7. Close the cursor
    CLOSE ThisCompany;
END;
```

注意

TopCustomerValue 过程具有以下显著的功能：

- 声明了 [未找到行] 异常。在过程的后期，此异常会在针对查询结果的循环结束时通知用户。

  有关异常的详细信息，请参见“过程和触发器中的错误和警告”一节第 814 页。
- 声明两个局部变量 ThisName 和 ThisValue，以保存来自该查询的每一行的结果。
- 声明了游标 ThisCompany。SELECT 语句会生成公司名称及公司所下订单总值的列表。
- TopValue 的值被设置为初始值 0，以便以后在循环中使用。
- ThisCompany 游标打开。
- LOOP 语句对该查询的每一行执行循环，依次将每一公司名称及订单值放置在变量 ThisName 和 ThisValue 中。如果 ThisValue 大于当前的最高值，则将 TopCompany 和 TopValue 重置为 ThisName 和 ThisValue。
- 该游标在过程结束后关闭。
- 还可以通过在 SELECT 语句中添加 "ORDER BY value DESC" 子句来编写不含循环语句的此过程。那样就只需获取游标的第一行。

TopCompanyValue 过程中的 LOOP 构造是标准形式，在最后一行处理完毕后退出。您可以使用 FOR 循环，以更紧凑的形式重新编写此过程。FOR 语句将上述过程的若干方面合并到单个语句中。

```
CREATE PROCEDURE TopCustomerValue2(
      OUT TopCompany CHAR(36),
      OUT TopValue INT )
BEGIN
   -- 1. Initialize the TopValue variable
   SET TopValue = 0;
   -- 2. Do the For Loop
   FOR CompanyFor AS ThisCompany
      CURSOR FOR
      SELECT CompanyName AS ThisName,
         CAST( sum( SalesOrderItems.Quantity *
               Products.UnitPrice ) AS INTEGER )
         AS ThisValue
```



```
      FROM Customers
         INNER JOIN SalesOrders
         INNER JOIN SalesOrderItems
         INNER JOIN Products
      GROUP BY ThisName
   DO
      IF ThisValue > TopValue THEN
         SET TopCompany = ThisName;
         SET TopValue = ThisValue;
      END IF;
   END FOR;
END;
```

在存储过程中更新游标

以下过程在 SELECT 语句上使用可更新的游标。它说明了如何使用存储过程语言在某一行上执行 UPDATE。

```
CREATE PROCEDURE UpdateSalary(
  IN employeeIdent INT,
  IN salaryIncrease NUMERIC(10,3) )
BEGIN
-- Procedure to increase (or decrease) an employee's salary
  DECLARE err_notfound
      EXCEPTION FOR SQLSTATE '02000';
  DECLARE oldSalary NUMERIC(20,3);
  DECLARE employeeCursor
    CURSOR FOR SELECT Salary from Employees
               WHERE EmployeeID = employeeIdent
    FOR UPDATE;
  OPEN employeeCursor;
  FETCH employeeCursor INTO oldSalary FOR UPDATE;
  IF SQLSTATE = err_notfound THEN
    MESSAGE 'No such employee' TO CLIENT;
  ELSE
    UPDATE Employees SET Salary = oldSalary + salaryIncrease
      WHERE CURRENT OF employeeCursor;
  END IF;
  CLOSE employeeCursor;
END;
```

以下语句调用上述存储过程：

```
CALL UpdateSalary( 105, 220.00 );
```

过程和触发器中的错误和警告

应用程序在执行某个 SQL 语句后，可以检查**状态代码**。此状态码（即返回代码）指示执行的语句是成功还是失败，并给出失败的原因。可以使用相同的机制来指示对过程执行的 CALL 语句是成功还是失败。

错误报告使用 SQLCODE 或 SQLSTATE 状态说明。有关 SQLCODE 和 SQLSTATE 错误以及警告值及其含义的完整说明，请参见错误消息。

只要执行 SQL 语句，就会在称作 SQLSTATE 和 SQLCODE 的特殊过程变量中出现值。特殊值指示在执行语句时是否遇到了任何意外情况。可以在执行 SQL 语句后通过 IF 语句来检查 SQLSTATE 或 SQLCODE 的值，并根据语句的成功与否采取措施。

例如，可使用 SQLSTATE 变量来指示是否成功地获取了某行。“在过程中在 SELECT 语句上使用游标”一节第 811 页一节中介绍的 TopCustomerValue 过程使用 SQLSTATE 测试来检测是否已经处理了 SELECT 语句的所有行。

过程和触发器中缺省的错误处理

本节介绍 SQL Anywhere 在您没有在过程中内置错误处理方式的情况下会如何处理过程执行期间所发生的错误。

要改变行为方式，可以使用在“在过程和触发器中使用异常处理程序”一节第 818 页中介绍的异常处理程序。

警告与错误的处理方式稍有不同：有关说明，请参见“过程和触发器中警告的缺省处理”一节第 817 页。

有两种无需使用显式错误处理即可处理错误的方法：

- **缺省的错误处理** 过程或触发器失败，并且将错误代码返回到调用环境。
- **ON EXCEPTION RESUME** 如果 ON EXCEPTION RESUME 子句出现在 CREATE PROCEDURE 语句中，则过程将在出现错误后继续执行，在导致错误的语句之后的下一语句恢复。

  使用 ON EXCEPTION RESUME 的过程的精确行为由 on_tsql_error 选项设置来规定。请参见“on_tsql_error 选项 [兼容性]”一节 《SQL Anywhere 服务器 - 数据库管理》。

缺省的错误处理

通常，如果过程或触发器中的 SQL 语句失败，该过程或触发器会停止执行，并将控制权及对 SQLSTATE 值和 SQLCODE 值的适当设置返回给应用程序。即使错误发生在从第一个过程或触发器直接或间接调用的过程或触发器中，上述情况也同样适用。对于触发器，导致该触发器触发的操作也将撤消，错误被返回给应用程序。

以下演示过程说明了在发生下面情况时的结果：应用程序调用过程 OuterProc，OuterProc 又调用过程 InnerProc，然后遇到错误。

```
CREATE PROCEDURE OuterProc()
BEGIN
   MESSAGE 'Hello from OuterProc.' TO CLIENT;
```



```
   CALL InnerProc();
   MESSAGE 'SQLSTATE set to ',
      SQLSTATE,' in OuterProc.' TO CLIENT
END;
CREATE PROCEDURE InnerProc()
   BEGIN
      DECLARE column_not_found
         EXCEPTION FOR SQLSTATE '52003';
      MESSAGE 'Hello from InnerProc.' TO CLIENT;
      SIGNAL column_not_found;
   MESSAGE 'SQLSTATE set to ',
      SQLSTATE, ' in InnerProc.' TO CLIENT;
END;

CALL OuterProc();
```

Interactive SQL [消息] 选项卡会显示以下内容：

```
Hello from OuterProc.
Hello from InnerProc.
```

InnerProc 中的 DECLARE 语句为与服务器已知的错误条件相关联的某一预定义的 SQLSTATE 值声明符号名称。

MESSAGE 语句将消息发送到 Interactive SQL [消息] 选项卡。

SIGNAL 语句从 InnerProc 过程内生成错误条件。

InnerProc 中跟随在 SIGNAL 语句之后的任何语句都不会执行：InnerProc 会立即将控制权传递回调用环境，在本例中调用环境即为 OuterProc 过程。OuterProc 中没有任何跟随在 CALL 语句之后的语句执行。错误条件返回到调用环境中，并在那里进行处理。例如，Interactive SQL 通过显示描述该错误的消息窗口来处理该错误。

TRACEBACK 函数会提供在发生错误时正执行的语句的列表。可以输入以下语句以从 Interactive SQL 使用 TRACEBACK 函数：

```
SELECT TRACEBACK();
```

使用 ON EXCEPTION RESUME 处理错误

如果 ON EXCEPTION RESUME 子句出现在 CREATE PROCEDURE 语句中，则该过程会在发生错误时检查随后的语句。如果该语句处理错误，则过程继续执行，在导致错误的语句之后的下一语句恢复。当发生错误时，它不会将控制权返回给调用环境。

可以通过 on_tsql_error 选项设置修改使用 ON EXCEPTION RESUME 的过程的行为。请参见“on_tsql_error 选项 [兼容性]”一节 《SQL Anywhere 服务器 - 数据库管理》。

错误处理语句包括以下这些：

- IF
- SELECT @variable =
- CASE
- LOOP
- LEAVE
- CONTINUE
- CALL
- EXECUTE
- SIGNAL
- RESIGNAL
- DECLARE
- SET VARIABLE

以下演示过程说明了在发生下面情况时的结果：应用程序调用过程 OuterProc，OuterProc 又调用过程 InnerProc，然后遇到错误。这些演示过程基于在本节前面部分中所使用的过程：

```
DROP PROCEDURE OuterProc;
DROP PROCEDURE InnerProc;

CREATE PROCEDURE OuterProc()
ON EXCEPTION RESUME
BEGIN
   DECLARE res CHAR(5);
   MESSAGE 'Hello from OuterProc.' TO CLIENT;
   CALL InnerProc();
   SET res=SQLSTATE;
   IF res='52003' THEN
      MESSAGE 'SQLSTATE set to ',
         res, ' in OuterProc.' TO CLIENT;
   END IF
END;

CREATE PROCEDURE InnerProc()
ON EXCEPTION RESUME
BEGIN
   DECLARE column_not_found
      EXCEPTION FOR SQLSTATE '52003';
   MESSAGE 'Hello from InnerProc.' TO CLIENT;
   SIGNAL column_not_found;
   MESSAGE 'SQLSTATE set to ',
   SQLSTATE, ' in InnerProc.' TO CLIENT;
END;

CALL OuterProc();
```

Interactive SQL [消息] 选项卡随后会显示以下内容：

```
Hello from OuterProc.
Hello from InnerProc.
SQLSTATE set to 52003 in OuterProc.
```

使用的执行路径如下所示：

1. OuterProc 执行并调用 InnerProc。

2. 在 InnerProc 中，SIGNAL 语句指示一个错误。
3. MESSAGE 语句不是错误处理语句，因此控制将被传递回 OuterProc 并且不显示该消息。
4. 在 OuterProc 中，跟随在错误后的语句将 SQLSTATE 值指派给名为 **res** 的变量。这是错误处理语句，因此过程的执行会继续并显示 OuterProc 消息。

过程和触发器中警告的缺省处理

错误和警告的处理方式各不相同。对错误的缺省操作是为 SQLSTATE 和 SQLCODE 变量设置值，并在遇到错误时将控制权返回给调用环境；而对警告的缺省操作是设置 SQLSTATE 和 SQLCODE 值，并继续执行过程。

以下演示过程说明缺省情况下警告的处理方式。这些演示过程基于在“过程和触发器中缺省的错误处理”一节第 814 页中使用的那些过程。

在本例中，SIGNAL 语句生成 [**未找到行**] 条件，这是警告，而非错误。

```
DROP PROCEDURE OuterProc;
DROP PROCEDURE InnerProc;

CREATE PROCEDURE OuterProc()
BEGIN
   MESSAGE 'Hello from OuterProc.' TO CLIENT;
   CALL InnerProc();
   MESSAGE 'SQLSTATE set to ',
      SQLSTATE,' in OuterProc.' TO CLIENT;
END;
CREATE PROCEDURE InnerProc()
BEGIN
   DECLARE row_not_found
      EXCEPTION FOR SQLSTATE '02000';
   MESSAGE 'Hello from InnerProc.' TO CLIENT;
   SIGNAL row_not_found;
   MESSAGE 'SQLSTATE set to ',
   SQLSTATE, ' in InnerProc.' TO CLIENT;
END;

CALL OuterProc();
```

Interactive SQL [消息] 选项卡随后会显示以下内容：

```
Hello from OuterProc.
Hello from InnerProc.
SQLSTATE set to 02000 in InnerProc.
SQLSTATE set to 00000 in OuterProc.
```

通过由警告 (02000) 设置的 SQLSTATE，这两个过程都在生成警告后继续执行。

执行 InnerProc 中的第二个 MESSAGE 语句时会重置该警告。任何 SQL 语句的成功执行都会将 SQLSTATE 重置为 00000，将 SQLCODE 重置为 0。如果某一过程需要保存错误状态，它必须在引发错误或警告的语句执行后立即指派状态值。

在过程和触发器中使用异常处理程序

对于某些类型的错误，通常需要进行拦截并在过程或触发器内处理，而不是将其传递回调用环境。这是通过使用**异常处理程序**完成的。

使用复合语句的 EXCEPTION 部分来定义异常处理程序。请参见“使用复合语句”一节第 801 页。

只要在复合语句中发生错误，异常处理程序就会执行。与错误不同，警告不会导致执行异常处理代码。如果错误出现在嵌套的复合语句中，或者出现在复合语句内任何地方所调用的过程或触发器中，异常处理代码也会执行。

中断错误 SQL_INTERRUPT、SQLSTATE 57014 的异常处理程序应只包含不可中断的语句，如 ROLLBACK 和 ROLLBACK TO SAVEPOINT 等。如果异常处理程序包含连接中断时调用的可中断语句，数据库服务器将在第一个可中断语句处停止异常处理程序，并返回中断错误。

用来说明异常处理情况的这一演示过程基于在“过程和触发器中缺省的错误处理”一节第 814 页中所使用的过程。

在本例中，附加代码会在 InnerProc 过程中处理 [**未找到列**] 错误。

```
DROP PROCEDURE OuterProc;
DROP PROCEDURE InnerProc;

CREATE PROCEDURE OuterProc()
BEGIN
   MESSAGE 'Hello from OuterProc.' TO CLIENT;
   CALL InnerProc();
   MESSAGE 'SQLSTATE set to ',
      SQLSTATE,' in OuterProc.' TO CLIENT
END;
CREATE PROCEDURE InnerProc()
BEGIN
   DECLARE column_not_found
    EXCEPTION FOR SQLSTATE '52003';
   MESSAGE 'Hello from InnerProc.' TO CLIENT;
  SIGNAL column_not_found;
  MESSAGE 'Line following SIGNAL.' TO CLIENT;
   EXCEPTION
      WHEN column_not_found THEN
      MESSAGE 'Column not found handling.' TO CLIENT;
        WHEN OTHERS THEN
      RESIGNAL ;
END;

CALL OuterProc();
```

Interactive SQL [**消息**] 选项卡随后会显示以下内容：

```
Hello from OuterProc.
Hello from InnerProc.
Column not found handling.
SQLSTATE set to 00000 in OuterProc.
```

EXCEPTION 子句声明异常处理程序。EXCEPTION 后的各行直到错误发生时才会执行。每一 WHEN 子句都指定一个异常名称（用 DECLARE 语句声明）以及在遇到该异常时要执行的语句。WHEN OTHERS THEN 子句指定当所发生的异常没有出现在之前的 WHEN 子句时要执行的语句。

在此示例中，RESIGNAL 语句将该异常传递给更高级别的异常处理程序。如果在异常处理程序中不指定 WHEN OTHERS THEN，则 RESIGNAL 是缺省操作。

附加说明

- 在 InnerProc 中，执行 EXCEPTION 处理程序，而不是执行 SIGNAL 语句之后的那些行。
- 因为遇到的错误是 [**未找到列**] 错误，因此执行其中用来处理该错误的 MESSAGE 语句，SQLSTATE 重置为零（指示没有发生任何错误）。
- 在异常处理代码执行后，控制被传递回 OuterProc，就像没有遇到任何错误一样继续执行。
- 不应将 ON EXCEPTION RESUME 与显式异常处理一起使用。如果包括 ON EXCEPTION RESUME，将不会执行异常处理代码。
- 如果 [**未找到列**] 异常的错误处理代码只是一个 RESIGNAL 语句，控制权会传递回 OuterProc 过程，SQLSTATE 仍然设置为值 52003。这就像是 InnerProc 中没有错误处理代码。由于在 OuterProc 中没有错误处理代码，因此该过程将失败。

异常处理和原子复合语句

在复合语句内处理异常时，复合语句结束时不会有活动的异常，出现异常前所做的更改也不被回退。即使对于原子复合语句，上述情况也同样适用。如果错误在原子复合语句内发生并被显式处理，则该原子复合语句中的某些语句（而非全部语句）会执行。

嵌套的复合语句和异常处理程序

只有 ON EXCEPTION RESUME 子句出现在过程定义中，跟随在导致出现错误的语句之后的代码才会执行。

您可以使用嵌套的复合语句来增强相关控制，以控制哪些语句在出现错误之后仍执行，哪些语句不执行。

以下示例说明如何使用嵌套的复合语句来控制程序流。此过程基于在“过程和触发器中缺省的错误处理”一节第 814 页中用作示例的过程。

```
DROP PROCEDURE OuterProc;
DROP PROCEDURE InnerProc;

CREATE PROCEDURE InnerProc()
BEGIN
   BEGIN
      DECLARE column_not_found
      EXCEPTION FOR SQLSTATE VALUE '52003';
      MESSAGE 'Hello from InnerProc' TO CLIENT;
      SIGNAL column_not_found;
      MESSAGE 'Line following SIGNAL' TO CLIENT
      EXCEPTION
         WHEN column_not_found THEN
            MESSAGE 'Column not found handling' TO
            CLIENT;
         WHEN OTHERS THEN
            RESIGNAL;
   END;
   MESSAGE 'Outer compound statement' TO CLIENT;
```

```
END;

CALL InnerProc();
```

Interactive SQL [消息] 选项卡随后会显示以下内容：

```
Hello from InnerProc
Column not found handling
Outer compound statement
```

在遇到导致错误的 SIGNAL 语句时，控制权会传递给复合语句的异常处理程序，并输出 [`Column not found handling`] 消息。控制权然后传递回外部复合语句，并输出 [`Outer compound statement`] 消息。

如果在内部复合语句中遇到了 [**未找到列**] 以外的其它错误，则异常处理程序会执行 RESIGNAL 语句。RESIGNAL 语句将控制权直接传递回调用环境，不再执行其余的外部复合语句。

在过程中使用 EXECUTE IMMEDIATE 语句

EXECUTE IMMEDIATE 语句使用文字字符串（在引号中）和变量的组合，允许在过程内构建语句。例如，以下过程包含创建表的 EXECUTE IMMEDIATE 语句。

```
CREATE PROCEDURE CreateTableProcedure(
      IN tablename char(128) )
BEGIN
   EXECUTE IMMEDIATE 'CREATE TABLE '
   || tablename
   || '(column1 INT PRIMARY KEY)'
END;
```

EXECUTE IMMEDIATE 语句可以与返回结果集的查询一起使用。例如：

```
CREATE PROCEDURE DynamicResult(
   IN Columns LONG VARCHAR,
   IN TableName CHAR(128),
   IN Restriction LONG VARCHAR DEFAULT NULL )
BEGIN
    DECLARE Command LONG VARCHAR;
    SET Command = 'SELECT ' || Columns || ' FROM ' || TableName;
    IF ISNULL( Restriction,'') <> '' THEN
         SET Command = Command || ' WHERE ' || Restriction;
    END IF;
    EXECUTE IMMEDIATE WITH RESULT SET ON Command;
END;
```

以下语句调用此过程：

```
CALL DynamicResult(
   'table_id,table_name',
   'SYSTAB',
   'table_id <= 10');
```

table_id	table_name
1	ISYSTAB
2	ISYSTABCOL
3	ISYSIDX
...	...

在原子复合语句中，不能使用导致 COMMIT（提交）的 EXECUTE IMMEDIATE 语句，因为在该上下文中不允许 COMMIT。

请参见“EXECUTE IMMEDIATE 语句 [SP]”一节 《SQL Anywhere 服务器 - SQL 参考》。

过程和触发器中的事务和保存点

过程或触发器中的 SQL 语句是当前事务的一部分。可以在一个事务内调用若干过程，或在一个过程中具有若干事务。

不允许将 COMMIT 和 ROLLBACK 用在任何原子语句中。请注意，由于触发触发器的 INSERT、UPDATE 或 DELETE 是原子语句，所以不允许将 COMMIT 和 ROLLBACK 用在触发器或触发器调用的任何过程中。

保存点可在过程或触发器内使用，但 ROLLBACK TO SAVEPOINT 语句永远不可以引用在原子操作开始前的保存点。此外，在完成原子操作后，释放该原子操作内的所有保存点。

另请参见

- “使用事务和隔离级别”第 99 页
- “原子复合语句”一节第 801 页
- “事务内的保存点”一节第 104 页

编写过程的提示

本节提供与开发过程有关的一些提示。

检查是否需要更改命令分隔符

编写过程时，不必在 Interactive SQL 或 Sybase Central 中更改命令分隔符。但是，如果使用其它浏览工具创建和测试过程和触发器，则可能需要将命令分隔符从分号更改为其它字符。

过程内的每个语句均以分号结束。为使某些浏览应用程序能够分析 CREATE PROCEDURE 语句本身，需要采用分号以外的其它命令分隔符。

如果使用需要更改命令分隔符的应用程序，最好将两个分号作为命令分隔符 (;;)；如果系统不允许使用多字符分隔符，则使用问号 (?)。

记住在过程内分隔语句

在过程中应使用分号结束每个语句。尽管对于语句列表中的最后一个语句可以省略分号，但最好还是在每个语句后都使用分号。

CREATE PROCEDURE 语句本身包含 RESULT 规范以及构成其主体的复合语句。在 BEGIN 或 END 关键字后，或在 RESULT 子句后，不需要分号。

在过程中为表使用全限定名

如果某一过程具有对该过程中包含的表的引用，则应始终以该表的所有者（创建者）的名称作为表名称的开始部分。

当过程引用某一表时，会使用过程创建者的组成员资格来查找没有显式指定所有者名称的表。例如，如果 user_1 所创建的过程引用 Table_B，并且不指定 Table_B 的所有者，则 Table_B 必须是由 user_1 创建的，或者 user_1 必须（直接或间接地）是 Table_B 所有者组的成员。如果这两个条件都不具备，调用该过程时将出现消息 [**未找到表**]。

通过在语句内使用相关名来提供用于该表的方便使用的名称，可以尽量减小使用长全限定名所带来的不便。相关名在“FROM 子句”一节 《SQL Anywhere 服务器 - SQL 参考》中加以介绍。

在过程中指定日期和时间

将日期和时间从过程发送到数据库时，它们将作为字符串发送。该字符串的日期部分根据 date_order 数据库选项的当前设置进行解释。由于其它连接可能会将该选项设置为不同的值，因此某些字符串可能会错误地转换为日期，或者数据库可能无法将字符串转换为日期。

在过程中使用日期字符串时，应该使用明确的日期格式 *yyyy-mm-dd* 或 *yyyy/mm/dd*。服务器会将这些字符串明确解释为日期，而不论 date_order 数据库选项如何设置。请参见“日期和时间数据类型”一节 《SQL Anywhere 服务器 - SQL 参考》。

验证正确传递了过程输入参数

验证输入参数的一个方法是使用 MESSAGE 语句在 Interactive SQL [消息] 选项卡中显示参数的值。例如，以下过程只显示输入参数 *var* 的值：

```
CREATE PROCEDURE message_test( IN var char(40) )
BEGIN
   MESSAGE var TO CLIENT;
END;
```

也可以使用调试程序来验证是否正确传递了过程输入参数。请参见“第 2 课：调试存储过程”一节 第 830 页。

过程、触发器、事件和批处理中允许使用的语句

除了以下语句，所有 SQL 语句在批处理中都是可接受的：

- ALTER DATABASE（语法 3 和 4）
- CONNECT
- CREATE DATABASE
- CREATE DECRYPTED FILE
- CREATE ENCRYPTED FILE
- DISCONNECT
- DROP CONNECTION
- DROP DATABASE
- FORWARD TO
- Interactive SQL 命令，例如 INPUT 或 OUTPUT
- PREPARE TO COMMIT
- STOP ENGINE

可以在过程、触发器、事件和批处理中使用 COMMIT、ROLLBACK 和 SAVEPOINT 语句，但有一些限制。请参见“过程和触发器中的事务和保存点”一节第 822 页。

有关详细信息，请参见“SQL 语句”《SQL Anywhere 服务器 - SQL 参考》中每个 SQL 语句的用法。

在批处理中使用 SELECT 语句

可以在批处理中包括一个或多个 SELECT 语句。例如：

```
IF EXISTS( SELECT *
            FROM SYSTAB
            WHERE table_name='Employees' )
THEN
   SELECT   Surname AS LastName,
            GivenName AS FirstName
   FROM Employees;
   SELECT Surname, GivenName
   FROM Customers;
   SELECT Surname, GivenName
   FROM Contacts;
END IF;
```

只需要在第一个 SELECT 语句中指定结果集的别名，因为服务器使用批处理中的第一个 SELECT 语句来描述结果集。

在每一查询后需要使用 RESUME 语句，以检索下一个结果集。

隐藏过程、函数、触发器和视图的内容

在某些情况下，当您分发应用程序和数据库时，可能不希望透露过程、函数、触发器和视图中包含的逻辑。作为附加的安全手段，您可以使用 ALTER PROCEDURE、ALTER FUNCTION、ALTER TRIGGER 和 ALTER VIEW 语句的 SET HIDDEN 子句来隐藏这些对象的内容。

SET HIDDEN 子句会对关联对象的内容进行模糊处理，使之难以理解，但这些对象仍然可以使用。还可以卸载对象并将其重新装入其它数据库。

修改是无法撤消的，会删除对象的原始文本。因此需要在数据库外保留对象的原始数据源。

使用调试程序进行调试时不会显示过程定义，对过程进行分析时也不会显示数据源。

在已经隐藏的对象上运行上面任何一个语句不会产生任何结果。

要隐藏特定类型的所有对象的文本，您可以使用类似于下面的循环：

```
BEGIN
    FOR hide_lp as hide_cr cursor FOR
        SELECT proc_name, user_name
        FROM SYS.SYSPROCEDURE p, SYS.SYSUSER u
        WHERE p.creator = u.user_id
        AND p.creator NOT IN (0,1,3)
    DO
        MESSAGE 'altering ' || proc_name;
        EXECUTE IMMEDIATE 'ALTER PROCEDURE "' ||
            user_name || '"."' || proc_name
            || '" SET HIDDEN'
    END FOR
END;
```

另请参见

- “ALTER FUNCTION 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “ALTER PROCEDURE 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “ALTER TRIGGER 语句”一节 《SQL Anywhere 服务器 - SQL 参考》
- “ALTER VIEW 语句”一节 《SQL Anywhere 服务器 - SQL 参考》

调试过程、函数、触发器和事件

目录

SQL Anywhere 调试程序简介

您可以在开发 SQL 存储过程、触发器、事件处理程序和用户定义函数的过程中使用 SQL Anywhere 调试程序。

使用 SQL Anywhere 调试程序可以执行许多任务，包括：

- **调试存储过程和触发器** 可以对 SQL 存储过程或触发器进行调试。
- **调试事件处理程序** 事件处理程序是 SQL 存储过程的扩展。本章中有关调试存储过程的内容也同样适用于调试事件处理程序。
- **浏览存储过程和类** 您可以浏览 SQL 过程的源代码。
- **跟踪执行** 逐行执行存储过程的代码。也可以监视被调用的函数堆栈。
- **设置断点** 可以让代码运行到断点处，并在该处停止运行。
- **设置中断条件** 断点包括代码行，但也可以指定中断执行代码的条件。例如，可以在第十次执行某行代码时停止执行，或仅在某个变量具有特定的值时停止执行。
- **检查和修改局部变量** 在断点处停止执行后，可以检查和变更局部变量的值。
- **在表达式处中断执行及检查** 在断点处停止执行后，可以检查各种表达式的值。
- **检查和修改行变量** 行变量是行级触发器的 OLD 和 NEW 值。您可以检查和修改这些值。
- **执行查询** 当执行在 SQL 过程中的断点处停止时，可以执行查询。这样，您就可以查看临时表中保存的中间结果、检查基表中的值以及查看查询执行计划。

提示
缺省情况下，SOAP 连接超时前的时间为 60 秒。您可以指定 -xs http(kto=0)，以便在尝试调试 SOAP 函数和过程时连接不会超时。请参见“-xs 服务器选项”一节 《SQL Anywhere 服务器 - 数据库管理》。

使用调试程序的要求

若要使用调试程序，您必须具有 DBA 权限或被授予 SA_DEBUG 组中的权限。创建所有数据库时，都会将该组添加到数据库中。每次只能有一个用户调试某个数据库。

教程：调试程序入门

本教程介绍如何连接到数据库，如何启动调试程序以及如何调试一个简单的存储过程。

第 1 课：连接到数据库并启动调试程序

♦ 启动调试程序

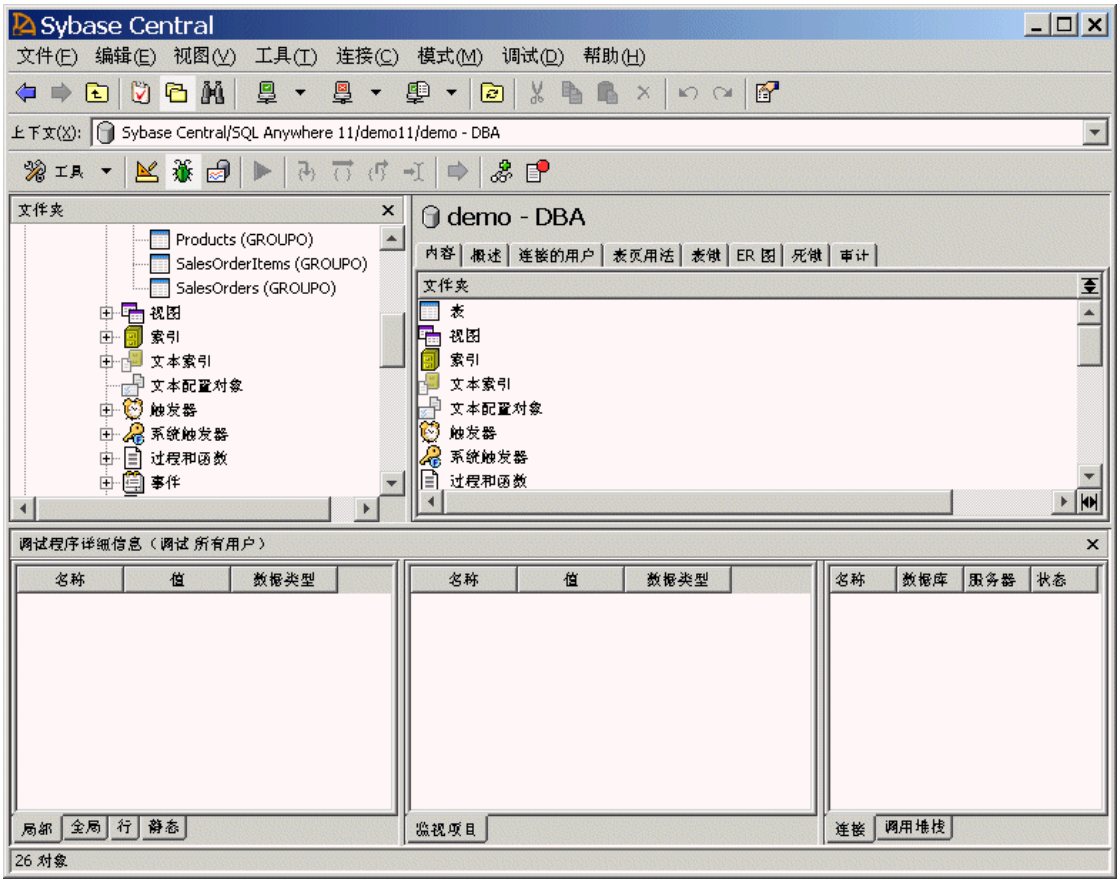
1. 创建一个目录以存放将在本教程中使用的示例数据库的副本，例如 *c:\demodb*。
2. 将示例数据库从 *samples-dir\demo.db* 复制到 *c:\demodb*，并重命名。

   有关 *samples-dir* 的信息，请参见“示例目录”一节 《SQL Anywhere 服务器 - 数据库管理》。
3. 单击 [**开始**] » [**程序**] » [**SQL Anywhere 11**] » [**Sybase Central**]。
4. 选择 [**连接**] » [**使用 SQL Anywhere 11 连接**]。
5. 在 [**ODBC 数据源名称**] 字段中，键入示例数据库的名称。
6. 单击 [**确定**]。
7. 选择 [**模式**] » [**调试**]。
8. 在 [**您希望调试哪个用户？**] 字段中，键入 *****，然后单击 [**确定**]。

   如果要调试不同的用户，必须退出调试模式，然后重新进入调试模式。

   Sybase Central 的底部会显示 [**调试程序详细信息**] 窗格，Sybase Central 工具栏会显示一组调试程序工具。

   提供用户名称时，将捕获具有该用户名称的连接的信息并出现在 [**连接**] 选项卡中。



第 2 课：调试存储过程

本课说明如何使用调试程序来确定存储过程中的错误。若要设置该阶段，您可以故意在 debugger_tutorial（它是 SQL Anywhere 示例数据库的一部分）中引入一个错误。

debugger_tutorial 过程应该返回一个结果集，该结果集中包含签订了最高价值订单的公司的名称及其订单价值。它通过循环遍历列出公司和订单的查询结果集来计算这些值。（通过使用 SELECT FIRST 查询，不在过程中添加逻辑就可以获得此结果。该过程用于创建方便的示例。）该过程中有一个有意安排的错误。在本教程中，您要诊断并修正该错误。

运行 debugger_tutorial 过程

debugger_tutorial 过程应该返回包含排在第一位的公司及其订购的产品价值的结果集。由于错误所致，它不会返回此结果集。在本课中，您要运行该存储过程。

♦ **运行 debugger_tutorial 存储过程**

1. 在 Sybase Central 的左窗格中，双击 [**过程和函数**]。
2. 右击 [**Debugger_Tutorial (GROUPO)**]，选择 [**从 Interactive SQL 执行**]。

   Interactive SQL 将打开并显示以下结果集：

top_company	top_value
(NULL)	(NULL)

   这是错误的结果。教程的剩余部分将诊断生成此结果的错误。
3. 关闭 Interactive SQL。

诊断错误

要诊断过程中的错误，请在过程中设置断点并单步执行代码，同时在执行过程时监视变量的值。

下面，您要在过程中的第一个可执行语句处设置断点。

♦ **诊断错误**

1. 选择 [**模式**] » [**调试**]。
2. 在右窗格中，双击 [**Debugger_Tutorial (GROUPO)**]。
3. 在右窗格中，定位以下语句：

```
OPEN cursor_this_customer;
```

4. 要添加断点，单击语句左侧灰色竖区。断点会显示为红色圆圈。
5. 在左窗格中，右击 [**Debugger_Tutorial (GROUPO)**]，选择 [**从 Interactive SQL 执行**]。

   在 Sybase Central 的 [**连接**] 选项卡中，将出现一个表示断点的黄色箭头。
6. 在 [**调试程序详细信息**] 窗口中，单击 [**局部**] 选项卡，以显示该过程中的局部变量的列表以及这些变量的当前值和数据类型。top_company、top_value、this_value 和 this_company 变量均未初始化，因此为 NULL。
7. 按 F11 键以滚动浏览该过程。当您到达下面这一行时，变量的值会改变：

```
IF this_value > top_value THEN
```

8. 再一次按 F11 键，确定执行采用了哪个分支。黄色箭头移回到下面的文本处：

```
customer_loop: loop
```

   IF 测试未返回 true。该测试之所以会失败，是因为任何值与 NULL 的比较都返回 NULL。值为 NULL 会导致测试失败并且不执行 IF...END IF 语句中的代码。

   此时，您可能会意识到问题在于 top_value 未初始化这一事实上。

确认诊断并修正错误

您可以测试 [问题在于调试程序中缺少对 top_value 的初始化] 这一假设，而不更改过程代码。

♦ 测试假设

1. 在 [**调试程序详细信息**] 窗口中，单击 [**局部**] 选项卡。
2. 单击 **Top_Value** 变量，并在 [**值**] 字段中键入 **3000**。
3. 重复按 F11，直至 **This_Value** 变量的 [**值**] 字段大于 3000。
4. 单击断点以使其变为灰色。
5. 按 F5 键以执行该过程。

   Interactive SQL 窗口会再次出现。它会显示正确的结果。

top_company	top_value
Chadwicks	8076

假设已经得到确认。问题在于没有初始化 top_value。

♦ 修正错误

1. 选择 [**模式**] » [**设计**]。
2. 在右窗格中，定位以下语句：

   ```
   OPEN cursor_this_customer;
   ```
3. 键入用于初始化 top_value 变量的新行：

   ```
   SET top_value = 0;
   ```
4. 选择 [**文件**] » [**保存**]。
5. 再次执行过程，并确认 Interactive SQL 显示了正确的结果。

您现在已经完成了本课。关闭任何打开的 Interactive SQL 窗口。

使用断点

调试程序中断执行源代码时的断点控制。

在 [调试] 模式下运行且连接运行到断点处时，相应的行为会视所选择的连接的不同而发生变化：

- 如果未选择任何连接，则自动选择连接并显示该过程的源以对连接进行调试。
- 如果已选择了一个连接，且该连接是运行到断点处的连接，则显示该过程的源以对连接进行调试。
- 如果已选择了一个连接，但该连接不是运行到断点处的连接，则出现一个窗口，提示您将连接更改为遇到断点的连接。

设置断点

断点指示调试程序在指定行中断执行。缺省情况下，断点应用于所有连接。

♦ 设置断点

1. 以具有 DBA 权限的用户身份连接到数据库。
2. 在左窗格中，双击 [**过程和函数**]。
3. 选择某个过程。
4. 选择 [**模式**] » [**调试**]。
5. 在 [**您希望调试哪个用户？**] 字段中，键入 * 对所有用户进行调试，或键入要调试的数据库用户的名称。
6. 在右窗格中，单击要插入断点的那一行。

   在单击的行中会出现一个光标
7. 按 F9。

   一个红色圆圈即会出现在代码行的左侧。

♦ 设置断点（[调试] 菜单）

1. 选择 [**调试**] » [**断点**]。
2. 单击 [**新建**]。
3. 在 [**过程**] 列表中，选择一个过程。
4. 根据需要，完成 [**条件**] 和 [**计数**] 字段。

   条件是 SQL 表达式，其计算结果必须为 true，断点才能中断执行。例如，通过输入以下条件，您可以设置断点以应用于指定用户建立的连接：

   ```
   CURRENT USER = 'user-name'
   ```

计数是断点使执行停止前命中断点的次数。值为 0 表示断点始终停止执行。

5. 单击 [**确定**]。断点会设置在过程中的第一个可执行语句上。

禁用和启用断点

您可以从 Sybase Central 右窗格或者从 [**断点**] 窗口更改断点的状态。

♦ **更改断点的状态**

1. 以具有 DBA 权限的用户身份连接到数据库。
2. 在左窗格中，双击 [**过程和函数**]。
3. 选择某个过程。
4. 选择 [**模式**] » [**调试**]。
5. 在右窗格中，单击要编辑的行左侧的断点指示符。断点从活动状态变为非活动状态。

♦ **更改断点的状态（[断点] 窗口）**

1. 以具有 DBA 权限的用户身份连接到数据库。
2. 在左窗格中，双击 [**过程和函数**]。
3. 选择某个过程。
4. 选择 [**模式**] » [**调试**]。
5. 选择 [**调试**] » [**断点**]。
6. 选择断点，单击 [**编辑**]、[**禁用**] 或 [**删除**]。
7. 单击 [**关闭**]。

编辑断点条件

您可以为断点添加条件，指示调试程序仅在满足特定条件或计数时在该断点处中断执行。对于过程和触发器，必须是 SQL 搜索条件。

例如，若要使断点只应用于特定连接，请设置断点条件。

♦ **设置断点条件或计数**

1. 以具有 DBA 权限的用户身份连接到数据库。
2. 在左窗格中，双击 [**过程和函数**]。
3. 选择某个过程。
4. 选择 [**模式**] » [**调试**]。

5. 选择 [**调试**] » [**断点**]。
6. 选择要编辑的断点，然后单击 [**编辑**]。
7. 在 [**条件**] 列表中，单击一个条件。例如，要设置断点使其仅应用于特定用户 ID 的连接，请输入下面的条件：

   ```
   CURRENT USER='user-name'
   ```

   在此条件中，*user-name* 是将为其激活断点的用户 ID。
8. 单击 [**确定**]，然后单击 [**关闭**]。

使用变量

使用调试程序，您可以在单步执行代码时查看并编辑变量的行为。调试程序提供了 [**调试程序详细信息**] 窗格，以显示在存储过程中使用的各种变量。当 Sybase Central 在 [调试] 模式下运行时，[**调试程序详细信息**] 窗格显示在 Sybase Central 的底部。

查看变量值

♦ **查看变量值**

1. 以具有 DBA 权限的用户身份连接到数据库。
2. 在左窗格中，双击 [**过程和函数**]。
3. 选择某个过程。
4. 选择 [**模式**] » [**调试**]。
5. 在 [**您希望调试哪个用户？**] 字段中，键入 * 对所有用户进行调试，或键入要调试的数据库用户的名称。
6. 在右窗格中，单击要插入断点的那一行。

   在单击的行中会出现一个光标
7. 按 F9。

   一个红色圆圈即会出现在代码行的左侧。
8. 在 [**调试程序详细信息**] 窗格中，单击 [**局部**] 选项卡。
9. 在左窗格中，右击该过程并选择 [**从 Interactive SQL 执行**]。变量和变量的值将显示在 [**局部**] 选项卡中。

查看全局变量

全局变量是由 SQL Anywhere 定义的，它包含有关当前连接、数据库和其它设置的信息。它们显示在 [**全局**] 选项卡的 [**调试程序详细信息**] 窗格中。

有关全局变量的列表，请参见“全局变量”一节 《SQL Anywhere 服务器 - SQL 参考》。

行变量用在触发器中，保存受触发语句影响的行的值。它们显示在 [**行**] 选项卡的 [**调试程序详细信息**] 窗格中。

有关触发器的详细信息，请参见“触发器简介”一节第 789 页。

静态变量用在 Java 类中。它们显示在 [**静态**] 选项卡上。

显示调用堆栈

在调试嵌套过程时，检查已执行的调用顺序是很有用的。您可以在 [**调用堆栈**] 选项卡中查看过程的列表。

♦ **显示调用堆栈**

1. 以具有 DBA 权限的用户身份连接到数据库。
2. 在左窗格中，双击 [**过程和函数**]。
3. 选择某个过程。
4. 选择 [**模式**] » [**调试**]。
5. 在 [**您希望调试哪个用户？**] 字段中，键入 * 对所有用户进行调试，或键入要调试的数据库用户的名称。
6. 在右窗格中，单击要插入断点的那一行。

   在单击的行中会出现一个光标
7. 按 F9。

   一个红色圆圈即会出现在代码行的左侧。
8. 在 [**调试程序详细信息**] 窗格中，单击 [**局部**] 选项卡。
9. 在左窗格中，右击该过程并选择 [**从 Interactive SQL 执行**]。
10. 在 [**调试程序详细信息**] 窗格中，单击 [**调用堆栈**] 选项卡。

    过程的名称会显示在 [调用堆栈] 选项卡中。当前过程显示在该列表的最上面。调用它的过程紧接着显示在下面。

使用连接

[**连接**] 选项卡显示到数据库的连接。任何时候都可以运行多个连接。一些连接可能会在断点处停止，而其它连接可能不会。

若要切换连接，请双击 [**连接**] 选项卡上的连接。

设置断点是一项很有用的技术，这样就可以针对某一个用户 ID 中断执行。设置断点的方法是设置以下形式的断点条件：

```
CURRENT USER = 'user-name'
```

SQL 特殊值 CURRENT USER 保存连接的用户 ID。

有关详细信息，请参见“编辑断点条件”一节第 834 页和“CURRENT USER 特殊值”一节《SQL Anywhere 服务器 - SQL 参考》。

术语表

术语表

Adaptive Server Anywhere (ASA)

SQL Anywhere Studio 的关系数据库服务器组件，专供在移动和嵌入式环境中使用，或作为中小型企业的服务器使用。在版本 10.0.0 中，Adaptive Server Anywhere 更名为 SQL Anywhere 服务器，SQL Anywhere Studio 更名为 SQL Anywhere。

另请参见： “SQL Anywhere”一节第 858 页

包

Java 中相关类的集合。

被引用对象

一种对象（如表），该对象在另一个对象（如视图）的定义中被直接引用。

另请参见： “主键”一节第 868 页

编码

也称作字符编码，编码是一种方法，通过该方法可以将字符集中的每个字符映射到一个或多个字节的信息，这些信息通常以十六进制数字表示。编码的一个例子是 UTF-8。

另请参见：

- “字符集”一节第 868 页
- “代码页”一节第 843 页
- “归类”一节第 847 页

标识符

用于引用数据库对象（如表或列）的字符串。标识符可以包含 A 到 Z、a 到 z、0 到 9、下划线 (_)、at 符号 (@)、数字符号 (#) 或美元符号 ($) 中的任何字符。

并发

同时执行两个或更多个独立并且可能存在竞争关系的进程。SQL Anywhere 会自动使用锁定来隔离事务，并确保每个并发应用程序看到的数据集均一致。

另请参见：

- “事务”一节第 855 页
- “隔离级别”一节第 846 页

参考数据库

MobiLink 中一种用于 UltraLite 客户端开发的 SQL Anywhere 数据库。在开发过程中，可以将一个 SQL Anywhere 数据库同时作为参考数据库和统一数据库使用。通过其它产品建立的数据库无法用作参考数据库。

参照完整性

遵守数据一致性控制规则（具体而言，不同表中主键值与外键值之间的关系）。若要实现参照完整性，每个外键中的值必须与被引用表中行的主键值相符。

另请参见：

- “主键”一节第 868 页
- “外键”一节第 860 页

策略

QAnywhere 中指定应在何时进行消息传输的方式。

插件模块

Sybase Central 中一种用于访问和管理产品的方法。当您安装相应的产品时，插件通常会自动安装并注册 Sybase Central。通常，插件在 Sybase Central 主窗口中作为顶级容器出现，并且使用产品本身的名称，如 SQL Anywhere。

另请参见： “Sybase Central”一节第 859 页

查询

一条或一组 SQL 语句，用于访问和/或操作数据库中的数据。

另请参见： “SQL”一节第 858 页

冲突解决

在 MobiLink 中，冲突解决是指一种逻辑，它指定当两个用户修改不同远程数据库上同一行时的处理方法。

重定向器

一种 Web 服务器插件，用于为客户端与 MobiLink 服务器之间的请求和响应选择发送路径。此插件还实现了负荷平衡和故障转移机制。

抽取

SQL Remote 复制中从统一数据库卸载相应结构和数据的行为。此信息用于初始化远程数据库。

另请参见： “复制”一节第 845 页

触发器

一种特殊形式的存储过程，用户运行修改数据的查询时会自动执行该存储过程。

另请参见：

- “行级触发器”一节第 847 页
- “语句级触发器”一节第 865 页
- “完整性”一节第 861 页

传输规则

QAnywhere 中用于确定何时进行消息传输、传输哪些消息以及应在何时删除消息的逻辑。

窗口

作为分析功能执行对象的行组。一个窗口可以包含一行、多行或所有行的数据，这些数据已根据窗口定义中提供的分组规格进行了分区。窗口会进行移动，以包括为输入中的当前行执行计算所需的行数或行范围。窗口结构的主要优点是，不需要执行附加查询就可以有机会对结果进行分组和分析。

创建者 ID

UltraLite Palm OS 应用程序中一种在创建应用程序时指派的 ID。

存储过程

存储过程是数据库中存储的一组 SQL 指令，用于在数据库服务器上执行一组操作或查询。

代理表

一种本地表，它所包含的元数据可以像访问本地表一样访问远程数据库服务器上的表。

另请参见： “元数据”一节第 866 页

代理 ID

另请参见： “客户端消息存储库 ID”一节第 851 页

代码页

代码页是一种将字符集的字符映射到数字表示的编码，数字表示通常是 0 到 255 之间的一个整数。例如，Windows 代码页 1252 就是一个代码页。就本文档而言，代码页和编码这两个术语可以互换。

另请参见：

- “字符集”一节第 868 页
- “编码”一节第 841 页
- “归类”一节第 847 页

DBA 权限

使用户能够在数据库中执行管理活动的权限级别。DBA 用户在缺省情况下具有 DBA 权限。

另请参见： “数据库管理员 (DBA)”一节第 857 页

dbspace

用于创建更多数据存储空间的附加数据库文件。一个数据库可以包含在最多 13 个独立的文件（一个初始文件和 12 个 dbspace）中。每个表及其索引必须包含在单个数据库文件中。SQL 命令 CREATE DBSPACE 可将新文件添加到数据库中。

另请参见： “数据库文件”一节第 858 页

动态 SQL

执行前由程序以编程方式生成的 SQL。UltraLite 动态 SQL 是一种专用于小型设备的 SQL 变体。

对象树

Sybase Central 中数据库对象的层次。对象树的顶层显示您的 Sybase Central 版本所支持的全部产品。每种产品展开后会显示其自己的对象子树。

另请参见： “Sybase Central”一节第 859 页

EBF

快速错误修正软件。快速错误修正软件是含有一个或多个错误修正软件的软件子集。错误修正软件列在更新程序的发行说明中。错误修正软件更新可能只适用于具有相同版本号的已安装软件。已对该软件执行了一些测试，但该软件尚未进行完全测试。除非您自己已验证了软件的适用性，否则不要随应用程序分发这些文件。

发布

MobiLink 或 SQL Remote 中一种用于标识将要同步的数据的数据库对象。在 MobiLink 中，发布仅存在于客户端。一个发布包括多个项目。SQL Remote 用户可以通过预订发布来接收发布。MobiLink 用户可以通过创建发布的同步预订来同步发布。

另请参见：

- “复制”一节第 845 页
- “项目”一节第 863 页
- “发布更新”一节第 844 页

发布更新

SQL Remote 复制中对一个数据库中的一个或多个发布所做更改的列表。发布更新将作为复制消息的一部分定期发送到远程数据库。

另请参见：

- “复制”一节第 845 页
- “发布”一节第 844 页

发布者

SQL Remote 复制中数据库内可以与其它复制数据库交换复制消息的单个用户。

另请参见： “复制”一节第 845 页。

FILE

SQL Remote 复制中一种使用共享文件来交换复制消息的消息系统。它对测试以及在没有显式消息传送系统的情况下进行的安装很有用。

另请参见“复制”一节第 845 页。

分析树

查询的代数表示。

服务

在 Windows 操作系统上，服务是在运行应用程序的用户 ID 未登录时的应用程序运行方式。

服务器管理请求

一种 QAnywhere 消息，其格式设置为 XML 并发送到 QAnywhere 系统队列，作为一种管理服务器消息存储库或监控 QAnywhere 应用程序的方法。

服务器启动的同步

一种从 MobiLink 服务器启动 MobiLink 同步的方式。

服务器消息存储库

QAnywhere 中在消息传输到客户端消息存储库或 JMS 系统之前服务器上用于临时存储消息的关系数据库。消息通过服务器消息存储库在各客户端之间进行交换。

复制

在物理上不相同的数据库之间共享数据。Sybase 有三种复制技术：MobiLink、SQL Remote 和复制服务器。

复制代理

请参见： “LTM”一节第 852 页

复制服务器

Sybase 的一种基于连接的复制技术，用于与 SQL Anywhere 和 Adaptive Server Enterprise 一起使用。它专用于在一些数据库之间进行接近实时的复制。

另请参见： “LTM”一节第 852 页

复制频率

SQL Remote 复制中一项针对每个远程用户的设置，它决定发布者的消息代理向该远程用户发送复制消息的频率应为多少。

另请参见： “复制”一节第 845 页。

复制消息

SQL Remote 或复制服务器中一种在发布数据库与预订数据库之间发送的通信。消息包含复制系统所需的数据、直通语句及信息。

另请参见：

- “复制”一节第 845 页
- “发布更新”一节第 844 页

隔离级别

一个事务中的操作对其它并发事务中的操作的可见程度。隔离级别有四级，编号依次为 0 至 3。第 3 级提供最高级别的隔离。级别 0 为缺省设置。SQL Anywhere 还支持以下三个快照隔离级别：快照、语句快照和只读语句快照。

另请参见： “快照隔离”一节第 851 页

个人服务器

与客户端应用程序在同一台计算机上运行的数据库服务器。个人数据库服务器通常由单个用户在一台计算机上使用，但它可以支持来自该用户的几个并发连接。

工作表

一种内部存储区域，用于在查询优化过程中存储中间结果。

故障切换

在活动服务器、系统或网络出现故障或意外终止时切换到冗余或备用的服务器、系统或网络。故障转移会自动进行。

关系数据库管理系统 (RDBMS)

一种以相关表的形式存储数据的数据库管理系统。

另请参见： “数据库管理系统 (DBMS)”一节第 857 页

规范化

对数据库模式的改进，目的在于按照基于关系数据库理论的规则消除冗余并改善组织。

归类

定义数据库中文本属性的字符集与排序顺序的组合。对于 SQL Anywhere 数据库，缺省归类取决于运行服务器时所使用的操作系统和语言；例如，英语 Windows 系统上的缺省归类为 1252LATIN1。归类（也称作归类序列）用于对字符串进行比较和排序。

另请参见：

- “字符集”一节第 868 页
- “代码页”一节第 843 页
- “编码”一节第 841 页

行级触发器

每更改一行即执行一次的触发器。

另请参见：

- “触发器”一节第 843 页
- “语句级触发器”一节第 865 页

回退日志

对在每个未提交的事务执行过程中所做更改的记录。当收到 ROLLBACK 请求或者系统出现故障时，未提交的事务会从数据库中回退，将数据库返回其原先的状态。每个事务都有一个单独的回退日志，事务完成时日志会被删除。

另请参见： “事务”一节第 855 页

iAnywhere JDBC 驱动程序

iAnywhere JDBC 驱动程序提供了一个 JDBC 驱动程序，与纯 Java jConnect JDBC 驱动程序相比，该驱动程序拥有一些性能优势和功能优点，但它不是纯 Java 解决方案。建议在大多数情况下使用 iAnywhere JDBC 驱动程序。

另请参见：

- “JDBC”一节第 848 页
- “jConnect”一节第 848 页

InfoMaker

一种报告和数据维护工具，它用于创建复杂的表格、报告、图形、交叉表和表，并创建将这些报告用作构件块的应用程序。

Interactive SQL

一种 SQL Anywhere 应用程序，用于查询和更改数据库中的数据以及修改数据库的结构。Interactive SQL 不但提供了一个用于输入 SQL 语句的窗格，还提供了一些用于返回有关查询处理过程的信息和结果集的窗格。

JAR 文件

Java 档案文件。一种压缩的文件格式，由一个或多个用于 Java 应用程序的包的集合组成。它将安装和运行 Java 程序所需的全部资源都放在一个压缩文件中。

Java 类

Java 中的主要代码结构单元。它是组合在一起的过程和变量的集合，将过程和变量组合在一起的原因是它们都与某个特定的可识别类别有关。

jConnect

JavaSoft JDBC 标准的 Java 实现。它为 Java 开发人员提供多层和异类环境中的本地数据库访问。但在大多数情况下，iAnywhere JDBC 驱动程序是首选的 JDBC 驱动程序。

另请参见：

- “JDBC”一节第 848 页
- “iAnywhere JDBC 驱动程序”一节第 847 页

JDBC

Java 数据库连接。一种 SQL 语言编程接口，它允许 Java 应用程序访问关系数据。首选的 JDBC 驱动程序是 iAnywhere JDBC 驱动程序。

另请参见：

- “jConnect”一节第 848 页
- “iAnywhere JDBC 驱动程序”一节第 847 页

基表

永久性的数据表。有时为区别于临时表和视图，会将这种表称作**基表**。

另请参见：

- “临时表”一节第 851 页
- “视图”一节第 855 页

基于会话的同步

一种同步类型，这种同步会使数据表示在统一数据库和远程数据库都一致。MobiLink 基于会话。

基于脚本的上载

MobiLink 中一种将上载过程自定义为使用日志文件的替代方法的方式。

基于 SQL 的同步

MobiLink 中一种使用 MobiLink 事件将表数据与支持 MobiLink 的统一数据库进行同步的方式。对于基于 SQL 的同步，可以直接使用 SQL，也可以使用面向 Java 和 .NET 平台的 MobiLink 服务器 API 返回 SQL。

基于文件的下载

在 MobiLink 中同步数据的一种方式，其中下载以文件的方式进行分发，从而支持脱机分发同步更改。

集成登录

一种登录功能，它允许将同一个用户 ID 和口令用于操作系统登录、网络登录和数据库连接。

监听器

一个程序 (dblsn)，用于 MobiLink 服务器启动的同步。监听器安装在远程设备上，它们被配置为在接收到来自通告程序的信息时启动针对设备的操作。

另请参见： “服务器启动的同步”一节第 845 页

检查点

将对数据库的所有更改都保存到数据库文件中的时间点。在其它时间，所提交的更改仅保存到事务日志中。

检查约束

对列或列集强制实施指定条件的一种限制。

另请参见：

- “约束”一节第 867 页
- “外键约束”一节第 861 页
- “主键约束”一节第 868 页
- “唯一约束”一节第 862 页

脚本

MobiLink 中为处理 MobiLink 事件而编写的代码。脚本通过编程方式控制数据交换，以满足业务需要。

另请参见： “事件模型”一节第 855 页

脚本版本

MobiLink 中为创建同步而一起应用的一组同步脚本。

校验

测试数据库、表或索引是否受到特定类型的文件损坏。

校验和

随数据库页本身一起记录的计算出的数据库页位数。校验和能够确保数据库页写入磁盘时位数相符，因此数据库管理系统可以通过它来验证数据库页的完整性。如果计数相符，即认为数据库页已成功写入。

镜像日志

另请参见： “事务日志镜像”一节第 856 页

角色

概念性数据库建模中从一个角度描述某种关系的动词或短语。您可以用两个角色来描述每种关系。例如，"包含" 和 "隶属于" 便是角色。

角色名

外键的名称。由于它命名外表和主表之间的关系，因此称作角色名。缺省情况下，角色名就是表名，除非其它外键已经使用该名称（在这种情况下，缺省的角色名是表名后接一个三位的唯一数字）。也可以自己创建角色名。

另请参见： “外键”一节第 860 页

局部临时表

一种临时表，仅在复合语句执行期间或连接结束之前存在。当您只需要将数据集装载一次时，局部临时表非常有用。缺省情况下，行会在提交时被删除。

另请参见：

- “临时表”一节第 851 页
- “全局临时表”一节第 854 页

客户端/服务器

一种软件体系结构，在这种体系结构中，一个应用程序（客户端）从另一个应用程序（服务器）获取信息并向该应用程序发送信息。这两个应用程序常位于通过网络连接的不同计算机上。

客户端消息存储库

QAnywhere 中一种用于在远程设备上存储消息的 SQL Anywhere 数据库。

客户端消息存储库 ID

QAnywhere 中一种对客户端消息存储库进行唯一标识的 MobiLink 远程 ID。

快照隔离

一种为发出读请求的事务返回数据的已提交版本的隔离级别。SQL Anywhere 提供了以下三种快照隔离级别：快照、语句快照和只读语句快照。使用快照隔离时，读操作不会阻塞写操作。

另请参见： “隔离级别”一节第 846 页

连接

关系系统中的一种基本操作，它通过比较指定列中的值将两个或更多个表中的行链接在一起。

连接 ID

用于标识客户端应用程序与数据库之间给定连接的唯一编号。可以使用以下 SQL 语句来确定当前连接 ID：

```
SELECT CONNECTION_PROPERTY( 'Number' );
```

连接类型

SQL Anywhere 提供了四种类型的连接：交叉连接、键连接、自然连接和使用 ON 子句的连接。

另请参见： “连接”一节第 851 页

连接配置

连接到数据库所需的一组参数，如用户名、口令和服务器名称，它们在存储后即可方便地使用。

连接启动的同步

一种 MobiLink 服务器启动的同步，在这种同步下，连接发生变化时会启动同步。

另请参见： “服务器启动的同步”一节第 845 页

连接条件

一种影响连接结果的限制。您可以通过紧跟在连接语句的后面插入 ON 子句或 WHERE 子句来指定连接条件。对于自然连接和关键连接，SQL Anywhere 会生成连接条件。

另请参见：

- “连接”一节第 851 页
- “生成的连接条件”一节第 856 页

临时表

为临时存储数据而创建的表。有两种类型：全局临时表和局部临时表。

另请参见：

- “局部临时表”一节第 850 页
- “全局临时表”一节第 854 页

LTM

日志传送管理器（Log Transfer Manager，简称 LTM）也称作复制代理。LTM 是一个与 Replication Server 一起使用的程序，它读取数据库事务日志并将提交的更改发送到 Sybase 复制服务器。

请参见： “复制服务器”一节第 846 页

轮询

在 MobiLink 服务器启动的同步中，轻量级轮询器（例如 MobiLink 监听器）从通告程序请求推式通知的方式。

另请参见： “服务器启动的同步”一节第 845 页

逻辑索引

指向物理索引的引用（指针）。磁盘上不存储逻辑索引的索引结构。

命令文件

包含 SQL 语句的文本文件。命令文件可以手工建立，也可以通过数据库实用程序自动建立。例如，dbunload 实用程序会创建一个命令文件，其中包含重新创建给定数据库所需的 SQL 语句。

MobiLink

一种基于会话的同步技术，其设计用途是将 UltraLite 和 SQL Anywhere 远程数据库与统一数据库同步。

另请参见：

- “统一数据库”一节第 859 页
- “同步”一节第 859 页
- “UltraLite”一节第 860 页

MobiLink 服务器

运行 MobiLink 同步的计算机程序，即 mlsrv11。

MobiLink 监控器

一种用于监控 MobiLink 同步的图形化工具。

MobiLink 客户端

有两种 MobiLink 客户端。对于 SQL Anywhere 远程数据库，MobiLink 客户端是 dbmlsync 命令行实用程序。对于 UltraLite 远程数据库，MobiLink 客户端内置于 UltraLite 运行时库中。

MobiLink 系统表

MobiLink 同步所需的系统表。它们由 MobiLink 安装程序脚本安装到 MobiLink 统一数据库中。

MobiLink 用户

MobiLink 用户用于与 MobiLink 服务器进行连接。在远程数据库上创建 MobiLink 用户，然后在统一数据库中注册该用户。MobiLink 用户名完全独立于数据库用户名。

模式

数据库的结构，其中包括表、列和索引以及它们之间的关系。

内连接

一种连接，在这种连接中，仅当两个表都满足连接条件时行才会出现在结果集中。内连接是缺省设置。

另请参见：

- “连接”一节第 851 页
- “外连接”一节第 861 页

ODBC

开放式数据库连接。一种用于与数据库管理系统连接的标准 Windows 接口。ODBC 是 SQL Anywhere 所支持的几种接口之一。

ODBC 管理器

一种随 Windows 操作系统提供的 Microsoft 程序，用于设置 ODBC 数据源。

ODBC 数据源

用户要通过 ODBC 访问的数据的规范以及获取该数据时所需的信息。

PDB

Palm 数据库文件。

PowerDesigner

一种数据库建模应用程序。PowerDesigner 为设计数据库或数据仓库提供了结构化的方法。SQL Anywhere 包括 PowerDesigner 的 Physical Data Model 组件。

PowerJ

一种 Sybase 产品，用于开发 Java 应用程序。

QAnywhere

应用程序到应用程序的消息传递（包括移动设备到移动设备和移动设备与企业之间的消息传递），它使在移动或无线设备上运行的自定义程序能够与处在中央位置的服务器应用程序进行通信。

QAnywhere 代理

QAnywhere 中一种运行在客户端设备上的进程，用于监控客户端消息存储库和确定应在何时传输消息。

嵌入式 SQL

一种 C 语言程序编程接口。SQL Anywhere 嵌入式 SQL 是 ANSI 和 IBM 标准的实现。

轻量级轮询器

在 MobiLink 服务器启动的同步中，轮询来自 MobiLink 服务器的推式通知的设备应用程序。

另请参见： “服务器启动的同步”一节第 845 页

全局临时表

一种临时表，在被显式地删除之前，其数据定义对所有用户都可见。全局临时表允许用户各自打开一个表的相同实例。缺省情况下，行在提交时被删除，并且始终是在连接结束时被删除。

另请参见：

- “临时表”一节第 851 页
- “局部临时表”一节第 850 页

日志文件

SQL Anywhere 所维护的事务日志。该日志文件用于确保在出现系统或介质故障时可以恢复数据库、提高数据库性能以及使用 SQL Remote 实现数据复制。

另请参见：

- “事务日志”一节第 855 页
- “事务日志镜像”一节第 856 页
- “完全备份”一节第 861 页

散列

散列是一种将索引条目转化为键的索引优化。索引散列旨在通过将足够的行实际数据与其行 ID 包括在一起，以避免进行先查找行、后装载行然后再将行解出才能得出索引值的高开销操作。

上载

同步过程的一个阶段，在此阶段数据从远程数据库传送到统一数据库。

设备跟踪

在 MobiLink 服务器启动的同步中，允许使用标识设备的 MobiLink 用户名来对消息进行寻址的功能。

另请参见： “服务器启动的同步”一节第 845 页

实例化视图

实例化视图是指已计算并已存储在磁盘上的视图。实例化视图同时具有视图的特征（使用查询说明进行定义）和表的特征（可以对其执行大多数表操作）。

另请参见：

- “基表”一节第 848 页
- “视图”一节第 855 页

世代号

MobiLink 中的一种机制，用于强制远程数据库先上载数据，然后再应用任何其它下载文件。

另请参见： “基于文件的下载”一节第 849 页

事件模型

MobiLink 中组成同步的事件（如 begin_synchronization 和 download_cursor）序列。如果为事件创建了脚本，则会调用事件。

视图

一种作为对象存储在数据库中的 SELECT 语句。它使用户能够看到一个或多个表中的行子集或列子集。每当用户使用特定表或表组合的视图时，都将利用存储在这些表中的信息重新计算视图。视图对确保安全以及定制数据库信息的外观来使数据访问简单明了有帮助。

事务

组成一个逻辑工作单元的 SQL 语句序列。事务要么全部得到处理，要么根本不做处理。SQL Anywhere 支持事务处理，并内置了锁定功能，使并发事务能够访问数据库而又不损坏数据。事务要么以 COMMIT 语句结束，该语句使对数据的更改成为永久性更改；要么以 ROLLBACK 语句结束，该语句撤消在事务执行过程中所做的全部更改。

事务日志

一种按进行更改的顺序存储对数据库所做全部更改的文件。它会提高性能并支持在数据库文件损坏时恢复数据。

事务日志镜像

同时维护的事务日志文件的完全相同副本（可选）。每当数据库更改写入事务日志文件时，也会同时写入事务日志镜像文件。

镜像文件应该与事务日志保留在不同的设备上，这样在任意设备出现故障时，日志的其它副本会确保数据可以安全地恢复。

另请参见： “事务日志”一节第 855 页

事务完整性

MobiLink 中对整个同步系统事务的有保证维护。要么同步整个事务，要么不对事务的任何部分进行同步。

生成的连接条件

一种自动生成的对连接结果的限制。有两种类型：关键和自然。指定 KEY JOIN 或指定关键字 JOIN 但不使用关键字 CROSS、NATURAL 或 ON 时，会生成关键连接。对于关键连接，所生成的连接条件取决于表之间的外键关系。指定 NATURAL JOIN 时会生成自然连接；所生成的连接条件基于两个表中的公用列名。

另请参见：

- “连接”一节第 851 页
- “连接条件”一节第 851 页

受保护的功能

数据库服务器启动时由 -sf 选项指定的功能，该数据库服务器上运行的任何数据库都无法使用该功能。

授权选项

一种权限级别，它允许用户向其他用户授予权限。

数据操作语言 (DML)

用于操作数据库中数据的 SQL 语句子集。DML 语句可以检索、插入、更新和删除数据库中的数据。

数据定义语言 (DDL)

用于定义数据库中数据结构的 SQL 语句子集。DDL 语句可以创建、修改和删除数据库对象（如表和用户）。

数据类型

数据的格式，如 CHAR 或 NUMERIC。在 ANSI SQL 标准中，数据类型也可以包括对大小、字符集和归类的限制。

另请参见： “域”一节第 865 页

数据立方体

一种多维结果集，每一维都以不同的方式对相同的结果进行分组和排序。数据立方体提供了有关数据的综合性信息，如果不使用数据立方体，要获得同样的信息就必须进行自连接查询和相关子查询。数据立方体是 OLAP 功能的一部分。

数据库

通过主键和外键关联的表的集合。表包含数据库中的信息。表和键一起定义数据库的结构。数据库管理系统会访问此信息。

另请参见：

- “外键”一节第 860 页
- “主键”一节第 868 页
- “数据库管理系统 (DBMS)”一节第 857 页
- “关系数据库管理系统 (RDBMS)”一节第 846 页

数据库对象

包含或接收信息的数据库组件。表、索引、视图、过程和触发器便是数据库对象。

数据库服务器

对所有针对数据库信息的访问进行管理的计算机程序。SQL Anywhere 提供了两种类型的服务器：网络服务器和个人服务器。

数据库管理系统 (DBMS)

用于创建和使用数据库的程序的集合。

另请参见： “关系数据库管理系统 (RDBMS)”一节第 846 页

数据库管理员 (DBA)

具有维护数据库所需权限的用户。DBA 通常负责对数据库模式的所有更改以及管理用户和组。数据库管理员角色自动内置于数据库中，其用户 ID 为 DBA，口令是 sql。

数据库连接

客户端应用程序与数据库之间的通信渠道。必须具有有效的用户 ID 和口令才能建立连接。为用户 ID 授予的特权决定了在连接过程中可以执行的操作。

数据库名称

服务器装载数据库时为数据库指定的名称。缺省数据库名是初始数据库文件的文件名（不含扩展名）。

另请参见： “数据库文件”一节第 858 页

数据库所有者 (dbo)

一种特殊的用户，他拥有不归 SYS 所有的系统对象。

另请参见：

- “数据库管理员 (DBA)”一节第 857 页
- “SYS”一节第 859 页

数据库文件

数据库保存在一个或多个数据库文件中。其中一个是初始文件，后面的文件称作 dbspace。每个表（包括其索引）都必须包含在单个数据库文件中。

另请参见： “dbspace”一节第 844 页

死锁

一组事务会进入的一种特殊状态，在该状态下这些事务都不能继续执行。

SQL

用于与关系数据库进行通信的语言。ANSI 定义了 SQL 的标准，其最新标准是 SQL-2003。SQL 的非官方全称是结构化查询语言。

SQL Anywhere

SQL Anywhere 的关系数据库服务器组件，专供在移动和嵌入式环境中使用，或作为中小型企业的服务器使用。SQL Anywhere 也是包含 SQL Anywhere RDBMS、UltraLite RDBMS、MobiLink 同步软件和其它组件的软件包的名称。

SQL Remote

一种基于消息的数据复制技术，用于在统一数据库与远程数据库之间进行双向复制。统一数据库和远程数据库必须是 SQL Anywhere。

SQL 语句

包含用于将指令传递给 DBMS 的 SQL 关键字的字符串。

另请参见：

- “模式”一节第 853 页
- “SQL”一节第 858 页
- “数据库管理系统 (DBMS)”一节第 857 页

锁定

一种在同时执行多个事务的过程中保护数据完整性的并发控制机制。SQL Anywhere 会自动应用锁以防止两个连接同时更改同一数据，并防止其它连接读取正接受更改的数据。

您可以通过设置隔离级别来控制锁定。

另请参见：

- “隔离级别”一节第 846 页
- “并发”一节第 841 页
- “完整性”一节第 861 页

索引

一组已排序的、与基表中的一个或多个列关联的键和指针。在表中一个或多个列上设置索引可以提高性能。

Sybase Central

一种数据库管理工具，通过图形用户界面提供 SQL Anywhere 数据库设置、属性和实用程序。Sybase Central 也可用于管理其它 Sybase 产品，其中包括 MobiLink。

SYS

一种拥有大多数系统对象的特殊用户。无法以 SYS 身份登录。

统一数据库

在分布式数据库环境中，是指用于存储数据主副本的数据库。出现冲突或差异时，将把统一数据库视为具有数据的主副本。

另请参见：

- “同步”一节第 859 页
- “复制”一节第 845 页

通信流

MobiLink 中 MobiLink 客户端与 MobiLink 服务器之间进行通信时所使用的网络协议。

通告程序

一种由 MobiLink 服务器启动的同步使用的程序。通告程序集成在 MobiLink 服务器中。它们会检查统一数据库是否有推式请求，并发送推式通知。

另请参见：

- “服务器启动的同步”一节第 845 页
- “监听器”一节第 849 页

同步

利用 MobiLink 技术在数据库之间复制数据的过程。

在 SQL Remote 中，同步专指以初始数据集初始化远程数据库的过程。

另请参见：

- “MobiLink”一节第 852 页
- “SQL Remote”一节第 858 页

推式请求

在 MobiLink 服务器启动的同步中，通告程序通过检查它来确定推式通知是否需要发送到设备的结果集中的一行值。

另请参见： “服务器启动的同步”一节第 845 页

推式通知

QAnywhere 中一种从服务器传送到 QAnywhere 客户端的特殊消息，用于提示客户端启动消息传输。在 MobiLink 服务器启动的同步中，从通告程序传送到包含推式请求数据和内部信息的设备的特殊消息。

另请参见：

- “QAnywhere”一节第 854 页
- “服务器启动的同步”一节第 845 页

UltraLite

一种针对小型设备、移动设备和嵌入式设备进行了优化的数据库。所面向的平台包括手机、传呼机和个人记事本。

UltraLite 运行时

一种过程中关系数据库管理系统，其中包括一个内置 MobiLink 同步客户端。每个 UltraLite 编程接口使用的库以及 UltraLite 引擎中都包括 UltraLite 运行时。

外表

包含外键的表。

另请参见： “外键”一节第 860 页

外部登录

与远程服务器通信时使用的替代登录名和口令。缺省情况下，SQL Anywhere 每次代表其客户端连接到远程服务器时都会使用这些客户端的名称和口令。但是，您可以通过创建外部登录来替换这一缺省设置。外部登录是指与远程服务器通信时使用的替代登录名和口令。

外键

一个表中复制另一个表中主键值的一个或多个列。外键建立表间的关系。

另请参见：

- “主键”一节第 868 页
- “外表”一节第 860 页

外键约束

对单个列或一组列的限制，指定表中的数据与某个其它表中数据的关系。对列集施加外键约束可使这些列成为外键。

另请参见：

- “约束”一节第 867 页
- “检查约束”一节第 849 页
- “主键约束”一节第 868 页
- “唯一约束”一节第 862 页

外连接

一种保留表中所有行的连接。SQL Anywhere 支持左、右和完全外连接。左外连接保留表中位于连接运算符左侧的行，当右表中的行不满足连接条件时，它将返回空值。完全外连接保留两个表中的所有行。

另请参见：

- “连接”一节第 851 页
- “内连接”一节第 853 页

完全备份

对整个数据库和事务日志（可选）的备份。完全备份包含数据库中的所有信息，因此可以在系统或介质出现故障时提供保护。

另请参见： “增量备份”一节第 867 页

完整性

遵守完整性规则的情况，完整性规则确保数据正确并准确，而且数据库的关系结构保持不变。

另请参见： “参照完整性”一节第 842 页

网关

一种 MobiLink 对象，存储在 MobiLink 系统表或通告程序属性文件中，包含有关如何发送用于服务器启动同步的消息的信息。

另请参见： “服务器启动的同步”一节第 845 页

网络服务器

从共享公共网络的计算机接受连接的数据库服务器。

另请参见： “个人服务器”一节第 846 页

网络协议

通信类型，如 TCP/IP 或 HTTP。

维护版本

维护版本是一套完整的软件，它升级已安装的具有相同主版本号的较早版本的软件（版本号格式是 *major.minor.patch.build*）。升级程序的发行说明中列出了错误修正软件和其它更改。

唯一约束

对某个列或一组列的限制，它要求所有非空值都各不相同。一个表可以有多个唯一约束。

另请参见：

- “外键约束”一节第 861 页
- “主键约束”一节第 868 页
- “约束”一节第 867 页

谓语

一种条件表达式，可以选择性地将其与逻辑运算符 AND 和 OR 组合在一起，以组成 WHERE 或 HAVING 子句中的条件集。在 SQL 中，求值结果为 UNKNOWN 的谓语将解释为 FALSE。

位数组

位数组是一种用于有效率地存储位序列的数组数据结构。位数组与字符串类似，不同的是其各个部分由 0（零）和 1（一）而不是字符组成。位数组通常用于保存一串布尔值。

Windows

Microsoft Windows 操作系统系列，如 Windows Vista、Windows XP 和 Windows 200x。

Windows CE

请参见“Windows Mobile”一节第 862 页。

Windows Mobile

Microsoft 为移动设备制造的操作系统系列。

文件定义数据库

MobiLink 中一种用于创建下载文件的 SQL Anywhere 数据库。

另请参见： “基于文件的下载”一节第 849 页

物理索引

索引存储在磁盘上的实际索引结构。

系统表

一种表，由 SYS 或 dbo 拥有，用于保存元数据。系统表也称作数据字典表，由数据库服务器创建并维护。

系统对象

由 SYS 或 dbo 拥有的数据库对象。

系统视图

存在于每一个数据库中的一种视图，它以易于理解的格式表示系统表中包含的信息。

下载

同步过程的一个阶段，在此阶段数据从统一数据库传送到远程数据库。

相关名

查询的 FROM 子句中使用的表或视图的名称—要么是表或视图的原始名称，要么是在 FROM 子句中定义的替代名称。

项目

在 MobiLink 或 SQL Remote 中，项目是表示整个表或表中行和列子集的数据库对象。项目在发布中组合在一起。

另请参见：

- “复制”一节第 845 页
- “发布”一节第 844 页

消息存储库

QAnywhere 中客户端和服务器设备上存储消息的数据库。

另请参见：

- “客户端消息存储库”一节第 850 页
- “服务器消息存储库”一节第 845 页

消息类型

SQL Remote 复制中指定远程用户与统一数据库发布者通信方式的数据库对象。一个统一数据库可能定义了几种消息类型，这样一来，不同的远程用户就可以使用不同的消息系统与统一数据库进行通信。

另请参见：

- “复制”一节第 845 页
- “统一数据库”一节第 859 页

消息日志

可存储来自数据库服务器或 MobiLink 服务器等应用程序的消息的日志。此类信息还可以出现在消息窗口中或记录到文件中。消息日志包括信息性消息、错误、警告以及来自 MESSAGE 语句的消息。

消息系统

SQL Remote 复制中用于在统一数据库与远程数据库之间交换消息的协议。SQL Anywhere 包括对以下消息系统的支持：FILE、FTP 和 SMTP。

另请参见：

- “复制”一节第 845 页
- “FILE”一节第 845 页

卸载

卸载数据库时会将数据库的结构和/或数据导出到文本文件（如果是结构，则导出到 SQL 命令文件中；如果是数据，则导出到 ASCII 逗号分隔文件中）。使用卸载实用程序来卸载数据库。

此外，您也可以使用 UNLOAD 语句卸载数据的选定部分。

性能统计

反映数据库系统性能的值。例如，CURRREAD 统计表示数据库服务器已发出但尚未完成的文件读取次数。

业务规则

基于实际要求的准则。通常，业务规则通过检查约束、用户定义数据类型以及事务的正确使用来实现。

另请参见：

- “约束”一节第 867 页
- “用户定义数据类型”一节第 865 页

引用对象

一种对象（如视图），其定义直接引用数据库中的另一个对象（如表）。

另请参见： “外键”一节第 860 页

用户定义数据类型

请参见“域”一节第 865 页。

游标

指向结果集的已命名链接，用于通过编程接口访问和更新行。在 SQL Anywhere 中，游标支持在查询结果中进行向前和向后移动。游标由两部分组成：游标结果集（通常由 SELECT 语句定义）和游标位置。

另请参见：

- “游标结果集”一节第 865 页
- “游标位置”一节第 865 页

游标结果集

与游标关联的查询所得到的行集。

另请参见：

- “游标”一节第 865 页
- “游标位置”一节第 865 页

游标位置

指向游标结果集中一个行的指针。

另请参见：

- “游标”一节第 865 页
- “游标结果集”一节第 865 页

语句级触发器

在整个触发语句完成后执行的触发器。

另请参见：

- “触发器”一节第 843 页
- “行级触发器”一节第 847 页

域

内置数据类型的别名，其中包括适用的精度值和小数位值，还可以选择是否包括 DEFAULT 值和 CHECK 条件。SQL Anywhere 中预定义了一些域，如货币数据类型。也称作用户定义数据类型。

另请参见： “数据类型”一节第 856 页

预订

MobiLink 同步中发布与 MobiLink 用户之间的客户端数据库中的一个链接，它使发布所描述的数据能够得到同步。

SQL Remote 复制中发布与远程用户之间的一种链接，它使用户能够与统一数据库交换该发布上的更新。

另请参见：

- “发布”一节第 844 页
- “MobiLink 用户”一节第 853 页

元数据

数据的数据。元数据描述其它数据的性质和内容。

另请参见： “模式”一节第 853 页

原子事务

保证成功完成或保证根本不予完成的事务。如果错误使原子事务的一部分无法完成，则将回退事务以防止数据库处于不一致的状态。

REMOTE DBA 特权

在 SQL Remote 中，消息代理 (dbremote) 所需的权限级别。MobiLink 中 SQL Anywhere 同步客户端 (dbmlsync) 所需的权限级别。当消息代理或同步客户端作为具有该权限的用户建立连接时，它将具有完全的 DBA 访问权。如果不是通过消息代理或同步客户端进行连接，则该用户 ID 将不具有附加权限。

另请参见： “DBA 权限”一节第 844 页

远程 ID

SQL Anywhere 和 UltraLite 数据库中一种由 MobiLink 使用的唯一标识符。远程 ID 初始情况下设置为 NULL，在数据库第一次同步期间将设置为 GUID。

远程数据库

MobiLink 或 SQL Remote 中一种与统一数据库交换数据的数据库。远程数据库可以共享统一数据库中的全部或部分数据。

另请参见：

- “同步”一节第 859 页
- “统一数据库”一节第 859 页

约束

对特定数据库对象（如表或列）中所包含值的限制。例如，列可以具有唯一性约束，该约束要求该列中的所有值互不相同。表可以具有外键约束，该约束指定该表中的信息与某个其它表中数据的关系。

另请参见：

- “检查约束”一节第 849 页
- “外键约束”一节第 861 页
- “主键约束”一节第 868 页
- “唯一约束”一节第 862 页

运营公司

一种 MobiLink 对象，存储在 MobiLink 系统表或通告程序属性文件中，包含有关供服务器启动的同步使用的公共运营公司的信息。

另请参见： “服务器启动的同步”一节第 845 页

增量备份

仅包含事务日志的备份，通常在两次完全备份之间使用。

另请参见： “事务日志”一节第 855 页

争用

为获取资源而竞争的行为。例如，就数据库而言，如果有两个或更多个用户试图编辑数据库的同一行，就会为获得编辑该行的权利而发生争用。

正则表达式

正则表达式是字符、通配符和运算符的序列，用于定义某种模式以在字符串内进行搜索。

直方图

直方图是列统计信息最重要的组成部分，是一种表示数据分布的方式。SQL Anywhere 维护直方图以为优化程序提供有关列值分布情况的统计信息。

直接行处理

MobiLink 中一种用于将表数据同步到 MobiLink 支持的统一数据库以外的数据源的方法。使用直接行处理时，上载和下载都可以实现。

另请参见：

- “统一数据库”一节第 859 页
- “基于 SQL 的同步”一节第 849 页

主表

包含外键关系中的主键的表。

主键

其值唯一标识表中各行的一个列或多个列。

另请参见： “外键”一节第 860 页

主键约束

一种对主键列的唯一性约束。一个表只能有一个主键约束。

另请参见：

- “约束”一节第 867 页
- “检查约束”一节第 849 页
- “外键约束”一节第 861 页
- “唯一约束”一节第 862 页
- “完整性”一节第 861 页

子查询

嵌套在 SELECT、INSERT、UPDATE 或 DELETE 语句或者其它子查询中的 SELECT 语句。

有两种类型的子查询：相关子查询和嵌套子查询。

字符串

字符串是以单引号围起的字符序列。

字符集

字符集是一组符号，包括字母、数字、空格和其它符号。字符集的一个例子是 ISO-8859-1，又称作 Latin1。

另请参见：

- “代码页”一节第 843 页
- “编码”一节第 841 页
- “归类”一节第 847 页

索引

其它

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

