SYBASE®

用户定义的函数指南

Sybase IQ 15.2

文档 ID： DC01138-01-1520-01
最后修订日期： 2010 年 4 月

Sybase, Inc., One Sybase Drive, Dublin, CA 94568

目录

读者

《用户定义的函数指南》这本书是为希望扩展 Sybase® IQ 功能的用户编写的。 通过这本书，可了解如何构建非常复杂的逻辑并将其合并到 SQL 查询或语句中，并了解有关使用 Sybase IQ 对用户定义的标量和集合函数进行编程的过程的概念。

读者

相关文档

Sybase IQ 和 SQL Anywhere 文档中还囊括其它信息。

*相关 **Sybase IQ** 文档*
Sybase IQ 文档集包括：

- 针对所用平台的《发行公告》 – 包含未能及时写入书中的最新信息。
  最新版本的发行公告可能已提供。 若要了解本产品 CD 发行以后增加的重要产品或文档信息，请访问 Sybase Product Manuals Web 站点。
- 针对所用平台的《安装和配置指南》 – 介绍 Sybase IQ 的安装、升级和某些配置过程。
- New Features Summary Sybase IQ（《Sybase IQ 新增功能摘要》） – 汇总了当前版本的新增功能和行为更改。
- 《Sybase IQ 的高级安全性》 – 涵盖了 Sybase IQ 数据存储库中用户加密列的用法。需要有单独的许可证才能安装此产品选件。
- 《错误消息》 – 列出了由 Sybase 错误代码、SQLCode 和 SQLState 引用的 Sybase IQ 错误消息以及 SQL 预处理器错误和警告。
- 《IMSL 数字库用户指南第二卷（共两卷）：C Stat 库》 – 包含 IMSL C Stat 库时序 C 函数的简要说明。 本书仅适用于 RAP – Trading Edition™ Enterprise 用户。
- 《Sybase IQ 简介》 – 包括针对不熟悉 Sybase IQ 或 Sybase Central™ 数据库管理工具的用户的练习。
- 《性能和调优指南》 – 介绍了针对较大数据库的查询优化、设计和调优问题。
- 《快速入门》 – 讨论了生成并查询 Sybase IQ 提供的演示数据库以验证 Sybase IQ 软件安装的步骤。 其中包括有关将演示数据库转换为 Multiplex 数据库的信息。
- 《参考手册》 – Sybase IQ 的参考指南：
  - 《参考：构件块、表和过程》 – 介绍了 Sybase IQ 支持的 SQL、存储过程、数据类型和系统表。
  - 《参考：语句和选项》 – 介绍了 Sybase IQ 支持的 SQL 语句和选项。
- 《系统管理指南》 – 包括：
  - 《系统管理指南：第一卷》 – 介绍了启动、连接、数据库创建、填充和编制索引、版本控制、归类、系统备份和恢复、故障排除和数据库修复。
  - 《系统管理指南：第二卷》 – 介绍如何编写和运行过程和批处理、使用 OLAP 编程、访问远程数据以及将 IQ 设置为 Open Server。 本书还讨论调度和事件处理、XML 编程和调试。
- 《时序指南》介绍用于时序预测和分析的 SQL 函数。 需要 RAP – Trading Edition™ Enterprise 才能使用此产品选件。
- Unstructured Data Analytics in Sybase IQ（《Sybase IQ 中的非结构化数据分析》）说明如何在 Sybase IQ 数据存储库中存储和检索非结构化数据。 需要有单独的许可证才能安装此产品选件。

- 《用户定义的函数指南》提供有关用户定义的函数、这些函数的参数以及可能的使用情形的信息。
- Using Sybase IQ Multiplex（《使用 Sybase IQ Multiplex》）说明如何使用可管理跨多个节点的大型查询负载的 Multiplex 功能。
- 《实用程序指南》提供了 Sybase IQ 实用程序参考资料，如可用的语法、参数和选项。

相关 SQL Anywhere 文档

注意： 因为 Sybase IQ 与 SQL Anywhere Studio® 的组件 SQL Anywhere® 共享许多组件，所以 Sybase IQ 支持许多与 SQL Anywhere 相同的功能。 IQ 文档集会在适当的时候提示您参考 SQL Anywhere Studio 文档。

SQL Anywhere 的文档包括：

- 《SQL Anywhere Server — 数据库管理》介绍如何运行、管理和配置 SQL Anywhere 数据库。它介绍数据库连接、数据库服务器、数据库文件、备份过程、安全性、高可用性和使用 Replication Server 复制以及管理实用程序和选项。
- 《SQL Anywhere Server — 编程》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#）生成和配置数据库应用程序。本书还介绍诸如 ADO.NET 和 ODBC 之类的各种编程接口。
- 《SQL Anywhere Server — SQL 参考》提供系统过程的参考信息和目录（系统表和视图）。它还提供 SQL 语言的 SQL Anywhere 实现的说明（搜索条件、语法、数据类型和函数）。
- SQL Anywhere Server – SQL Usage（《SQL Anywhere Server – SQL 用法》）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生成存储过程和触发器。

此外，还可以参见“Product Manuals”（产品手册）的“SQL Anywhere Studio 11.0 集合”中以及“DocCommentXchange”中的 SQL Anywhere 文档。

了解用户定义的函数

Sybase IQ 概述

了解如何在 Sybase IQ 中使用用户定义的函数。

Sybase IQ 允许用户定义的函数 (UDF)，它们在数据库容器中执行。 UDF 执行功能作为可选组件提供，可在 Sybase IQ 中或 RAP - Trading Edition™ R3 的 RAPStore 组件中使用。

若要使用这些外部 C/C++ UDF 接口，需要具备 IQ_UDF 许可证。

这些外部 C/C++ UDF 与早期版本的 Sybase IQ 中提供的 Interactive SQL UDF 不同。 Interactive SQL UDF 保持不变且不需要特殊许可证。

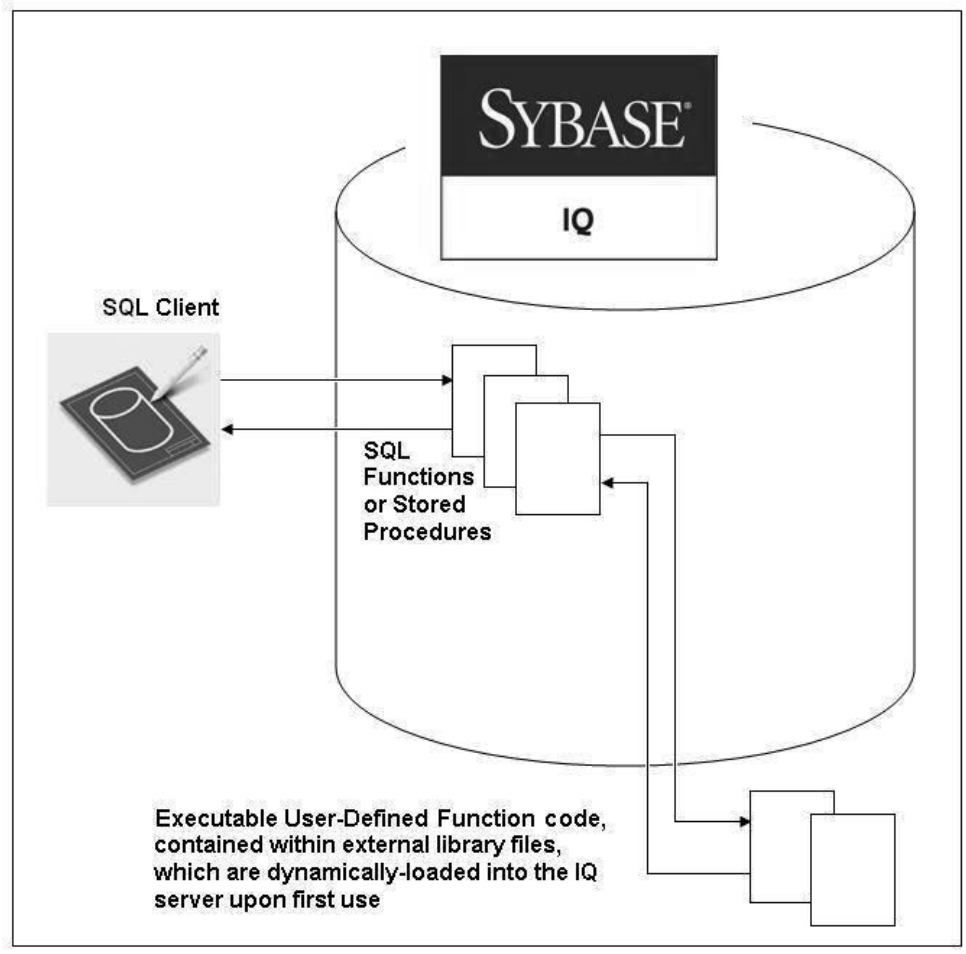
在 Sybase IQ 中执行的 UDF 利用了 IQ 服务器的卓越性能，同时还使用户可灵活分析数据并为其提供灵活的编程解决方案。 用户定义的函数包含两个组成部分：

- UDF 声明和
- UDF 可执行代码

UDF 是在 SQL 环境中通过可说明参数并提供对外部库的引用的 SQL 函数或存储过程进行声明的。

UDF 的实际可执行部分包含在外部（共享对象或动态装载）库文件中，在首次调用与该库相关的 UDF 声明函数或存储过程时，IQ 服务器会自动装载该文件。 装载后，库仍驻留在 IQ 服务器中，以便通过后续调用引用该库的 SQL 函数或存储过程来进行快速访问。

下图描绘了 Sybase IQ 用户定义的函数体系结构。



Sybase IQ 支持高性能进程中外部 C/C++ 用户定义的函数。 这种样式的 UDF 支持以 C 或 C++ 代码编写的函数，这些函数遵循本指南中介绍的接口。

UDF 的 C/C++ 源代码编译到一个或多个外部库中，在以后需要时，这些库将装载到 IQ 服务器的进程空间中。 通过 SQL 函数将 UDF 调用机制定义到 Sybase IQ 服务器。 在从 SQL 查询调用 SQL 函数时，IQ 服务器会装载相应的库（如果尚未装载）。

为简化 UDF 安装的管理，Sybase 建议 UDF 开发人员将多个 UDF 函数打包在单个库中。

为简化 UDF 的构造，Sybase IQ 包括基于 C 的 API。 API 包括一组预定义的 UDF 入口点、明确定义的上下文数据结构以及一系列提供从 UDF 返回到 Sybase IQ 服务器的通信机制的 SQL 回调函数。 Sybase IQ UDF API 允许软件供应商和专业的最终用户开发、打包和销售自己的 UDF。

用户定义的函数符合 Sybase IQ 数据库

开发用户定义的函数以与 Sybase IQ 数据库一起使用。

无缝执行

UDF 必须无缝运行在数据库容器中。 虽然 Sybase IQ 是一种包含许多文件的复杂产品，但主用户交互是通过使用行业标准结构化查询语言 (SQL) 的 IQ 服务器进程 (iqsrv15) 进行的。 UDF 的执行应完全通过 SQL 命令来完成；用户无需了解基础实现方法便可使用 UDF。

UDF 在 Sybase IQ 的覆盖下运行，因此不要编写主控台消息。 通过预定义的例外消息，将任何反馈提供给用户。

UDF 应管理 Sybase IQ UDF API 定义的内存和临时结果。

Sybase IQ 以可靠的方式管理磁盘 I/O 以确保数据可用性和完整性。 通常，不应将 UDF 写入文件系统中或从文件系统中读取 UDF。

Sybase IQ 是多用户应用程序。 多个用户可以同时执行同一 UDF。 某些 OLAP 查询导致在同一查询中多次执行 UDF，有时并行执行。 有关将 UDF 设置为并行运行的其它详细信息，请参见集合 UDF 调用模式 （第 74 页）。

国际化

Sybase IQ 已国际化，因此可以在世界上不同国家/地区销售给讲许多不同语言的用户。 错误消息已从代码中提取出并放入了外部文件中。 这允许您将错误消息本地化为新语言，而无需进行大量代码更改。

为支持多种语言，UDF 也应国际化。 一般来说，大多数 UDF 将对数值数据进行操作。 在某些情况下，UDF 可能接受字符串关键字作为一个或多个参数。 除了 UDF 使用的任何例外文本和日志消息之外，将这些关键字也放入外部文件中。

Sybase IQ 也已本地化为几种非英语的外国语言。 为支持本地化为 Sybase IQ 支持的语言，Sybase 建议您国际化 UDF，从而允许以后由独立组织本地化它们。

有关 Sybase IQ 中的国际语言支持的详细信息，请参见 Sybase IQ System Administration Guide（《Sybase IQ 系统管理指南》）>“International Languages and Character Sets”(国际语言和字符集）。

另请参见 www.Sybase.com 上的“使用交叉字符集映射进行调试”(Debugging Using Cross-Character-Set Maps)。 本白皮书讨论如何使用多字节数据，而不是输入关键字、例外消息和日志条目。

平台差异

开发 UDF，使其在 Sybase IQ 支持的各种平台上运行。 Sybase IQ 15.x 服务器运行在 64 位体系结构中，并受 MS Windows（64 位）系列的操作系统的几种平台支持。 各种类型和版本的 UNIX（64 位）也支持它，包括 Solaris、HP-UX、AIX 和 Linux。

应避免的做法

了解用于创建用户定义的函数的好的做法。

- 不要在 SQL 注册脚本中对库路径进行硬编码。 这种做法难以为用户提供将 UDF 与 Sybase IQ 安装到同一目录的灵活性。
- 不要编写输出文件。 Sybase IQ 版本 15.1 包括对 Sybase IQ 内的 UDF 结果的体系结构限制。 由于此限制，已经开发了一些 UDF，以便写入 Sybase IQ 容器之外的临时结果文件。 版本 15.1 的第一个 ESD 版本会将此体系结构限制扩展为可用大小。 计划从以后的 Sybase IQ 版本中完全删除此限制。
- 不要编写不明确的代码，或可能意外无限循环且没有为用户提供取消 UDF 调用的机制的构造（请参见 UDF 和 UDAF 回调函数 （第 73 页）中的函数 **'get_is_cancelled()'**）。
- 不要执行每次调用时都重复的复杂或使用大量内存的操作。 在对包含成千上万行的表调用 UDF 时，高效执行变得至关重要。 Sybase 建议您每次为一千至数千行分配内存块，而非逐行进行。
- 不要从 UDF 内打开数据库连接或执行数据库操作。 UDF 执行所需的所有参数和数据必须作为参数传递给 UDF。
- 在命名 UDF 时，不要使用保留字。

用户定义的函数的类型

用户定义的函数有多种类型。

- 标量或集合 – UDF 对单个值（标量）或多个值（集合）进行操作。 集合 UDF 有时也称为 UDA 或 UDAF。 用于对集合 UDF 进行编码的上下文结构与用于对标量 UDF 进行编码的上下文结构略有不同。
- 确定型或非确定型 – 函数结果可由输入参数和数据单独确定（确定型），也可由某个随机行为确定（非确定型）。 非确定型 UDF 的参数通常需要随机种子作为输入参数之一。
- （仅限集合 UDF）单个输出或多个输出 – 集合函数可生成单个结果或一组结果。 输出结果集中的数据点数可能未必与输入集中的数据点数相匹配。 在 Sybase IQ 15.1 中，多个输出的集合 UDF 必须使用临时输出文件来保存结果。 在以后的版本中，您可以实现返回打包在大二进制对象 (BLOB) 中的多个结果的集合函数。

用户定义的函数的命名约定

UDF 名称必须与 Sybase IQ 中的其它标识符遵循相同的限制。

Sybase IQ 标识符的最大长度为 128 字节。 为简化使用，UDF 名称应以字母字符开头。 Sybase IQ 定义的字母字符包括字母表中的字母以及下划线 (_)、at 符号 (@)、数字或

英镑符号 (#) 和美元符号 ($)。 UDF 名称应完全由这些字母字符和数字（数字 0 至 9）组成。 UDF 名称不应与 SQL 保留字冲突。 Sybase IQ 15.1《参考：构件块、表和过程》>“SQL 语言元素”>“保留字”一节中提供了 SQL 保留字的列表。

虽然 UDF 名称（与其它标识符一样）也可以包含保留字、空格、上面列出的那些以外的字符，并可以以非字母字符开头，但不建议这样做。 如果 UDF 名称具有其中任何特征，则您必须用引号或方括号括住它们，这会使它们的使用变得更加困难。

UDF 与其它 SQL 函数和存储过程驻留在相同的名称空间中。 为避免与现有存储过程和函数相冲突，应在 UDF 前面加上唯一的简短（2 至 5 个字母）首字母缩略词和下划线。 选择与本地环境中已定义的其它 SQL 函数或存储过程不冲突的 UDF 名称。

下面是一些已经使用的前缀：

- **debugger_tutorial** – 本机 Sybase IQ 安装提供的存储过程。
- **ManageContacts** – Sybase IQ 演示数据库提供的存储过程。
- **Show** – 用于显示 Sybase IQ 演示数据库中的数据的存储过程。
- **sp_Detect_MPX_DDL_conflicts** – 本机 Sybase IQ 安装提供的存储过程。
- **sp_iqevbegintxn** – 本机 Sybase IQ 安装提供的存储过程。
- **sp_iqmpx** – Sybase IQ 提供的用于帮助进行 Multiplex 管理的函数和存储过程。
- **ts_** – 可选的财务时序和预测函数。

用户定义的函数的设计基础

在开发 UDF 时，需要记住一些基本注意事项。

本文档假定 UDF 开发人员已熟悉开发软件的基础知识，包括好的程序设计和开发、独立测试等等。

除了标准软件开发实践外，UDF 开发人员还应该记住他们开发的代码将在 Sybase IQ 数据库容器内执行，并了解数据库容器施加的限制。

集合 UDF 的开发人员还应该熟悉 OLAP 查询以及如何将其转换为 UDF 调用模式。

因为 UDF 可能同时被多个线程调用，所以必须将 UDF 构造为线程安全的。

示例代码

从 Sybase IQ 15.1 开始，示例 UDF 源代码随产品一起提供。 最新版本的示例代码总是随最新版本的 Sybase IQ 一起提供。

若要了解是否有对示例 UDF 源代码的最新更改，请参见适用于相关版本和操作系统平台的 Sybase IQ 发行公告。

《用户定义的函数指南》中记录的示例 UDF 代码可能不是随 Sybase IQ 产品一起提供的最新版本。 在 UNIX 平台上，示例 UDF 代码位于：

```
$SYBASE/IQ-15_1/samples/udf
(where $SYBASE is the installation root)
```

在 Windows 平台上，示例 UDF 代码位于：

```
C:\Documents and Settings\All Users\SybaseIQ\samples\udf
```

创建并执行用户定义的函数

用户定义的函数 (UDF) 将单个值返回到调用环境中。

注意： 用户定义的函数是可获许可选项，需要 IQ_UDF 许可证。 安装许可证会启用用户定义的函数。

可以在多种配置中安装 Sybase IQ。 UDF 必须轻松安装在此环境中，并且必须能够运行在所有受支持的配置中。 Sybase IQ 安装程序提供缺省的安装目录，但允许用户选择其它安装目录。 UDF 开发人员应考虑为 UDF 库及相关的 SQL 函数定义脚本的安装提供相同的灵活性。

创建用户定义的函数

了解如何创建和配置外部 C/C++ 用户定义的函数 (UDF)。

有关使用 Interactive SQL 创建 UDF 的说明，请参见《系统管理指南：第二卷》>“使用过程和批处理”。

1. 使用 **CREATE FUNCTION** 或 **CREATE AGGREGATE FUNCTION** 语句向服务器声明 UDF。 将这些语句作为命令写入并执行，或者通过 Sybase Central 新建函数向导使用合适的 CREATE 语句。

   CREATE FUNCTION 语句的外部 C/C++ 形式需要 DBA 或 RESOURCE 权限，因此标准用户无权声明任何此类型的 UDF。
2. 编写 UDF 库标识函数。 （第 15 页）
3. 将 UDF 定义为一组 C 或 C++ 函数。 请参见“定义标量 UDF” （第 32 页）或“定义集合 UDF” （第 45 页）。
4. 用 C/C++ 实现函数入口点。
5. 编译 UDF 函数和库标识函数。 （第 16 页）
6. 将编译的文件链接到动态可链接库。

对 SQL 语句中的 UDF 的任何引用首先将链接动态可链接库（如果需要）。 然后将调用“调用模式” （第 73 页）。

由于这些高性能外部 C/C++ 用户定义的函数涉及将非服务器库代码装载到服务器的进程空间中，因此，编写错误或恶意编写的函数会导致数据完整性、数据安全性和服务器的稳定性存在潜在风险。 为了管理这些风险，每个 IQ 服务器都可以显式启用或禁用此功能 （第 25 页）。

使用 SQL Anywhere 方言创建用户定义的函数

Watcom-SQL 和 Transact-SQL 是 SQL Anywhere 支持的 SQL 方言，在创建用户定义的函数时可以使用。

1. 在 Sybase Central 中，以具有 DBA 或“资源”权限的用户的身份连接到数据库。
2. **选择“视图”>“文件夹”。**
3. 在左窗格中，右键单击“过程和函数”，然后选择“新建”>“函数”。
4. 为函数输入名称并选择将拥有该函数的用户。
5. 选择函数的 SQL 术语或语言。 单击“下一步”。
6. 选择要在函数中返回的值的类型，并指定值的大小、单位和范围。
7. 键入返回值的名称，然后单击“下一步”。
8. 添加说明新函数用途的注释。 单击**“完成”**。
9. 在右窗格中，单击“SQL”选项卡以填写过程代码。

在 Sybase Central 中声明用户定义的标量函数

Sybase IQ 支持可在任何可以使用 SQRT 函数的位置使用的简单标量 UDF。 这些标量 UDF 可以是确定型函数，这表示对于一组给定的参数值，函数始终返回相同的结果值。 Sybase IQ 还支持非确定型标量函数，这表示相同的参数可以返回不同的结果。

1. 在 Sybase Central 中，以具有 DBA 或“资源”权限的用户的身份连接到数据库。
2. 在左窗格中，右键单击“过程和函数”，然后选择“新建”>“函数”。
3. 在“欢迎”(Welcome) 对话框中，键入函数名称并选择将成为该函数所有者的用户。
4. 若要创建用户定义的函数，请选择“外部 C/C++”。 单击“下一步”。
5. 在“外部函数属性”对话框中，选择“标量”。
6. 键入动态可链接库文件的名称（省略 .so 或 .dll 扩展名）。
7. 键入描述符函数的名称。 单击“下一步”。
8. 选择要在函数中返回的值的类型，并指定值的大小、单位和范围。 单击“下一步”。
9. 选择函数是否为确定型函数。
10. 指定函数是使用空值还是忽略空值。
11. 选择用于运行函数的特权是来自定义用户（定义者）还是调用用户（调用者）。
12. 添加说明新函数用途的注释。 单击“完成”。
13. 在右窗格中，单击“SQL”选项卡以填写过程代码。

在 Sybase Central 中声明用户定义的集合函数

Sybase IQ 支持用户定义的集合函数 (UDAF)。SUM 函数就是一个内置集合函数。简单的集合函数采用一组参数值并根据该组输入生成一个结果值。可以编写用户定义的集合函数，并且可在任何可以使用 SUM 集合的位置使用所编写的函数。

1. 在 Sybase Central 中，以具有 DBA 或“资源”权限的用户的身份连接到数据库。
2. 在左窗格中，右键单击“过程和函数”，然后选择“新建”>“函数”。
3. 在“欢迎”(Welcome) 对话框中，键入函数名称并选择将成为该函数所有者的用户。
4. 若要创建用户定义的函数，请选择“外部 C/C++”。 单击“下一步”。
5. 选择“集合”。
6. 键入动态可链接库文件的名称（省略 .so 或 .dll 扩展名）。
7. 键入描述符函数的名称。 单击“下一步”。
8. 选择要在函数中返回的值的类型，并指定值的大小、单位和范围。 单击“下一步”。
9. 选择用于运行函数的特权是来自定义用户（定义者）还是调用用户（调用者）。
10. 指定是允许在 **OVER** 子句中使用函数，需要在该子句中使用函数，还是不允许在该子句中使用函数。 单击“下一步”。

    如果不允许在 **OVER** 子句中使用函数，请继续执行步骤 14。
11. 指定当函数用于定义窗口时，是否需要使用 **ORDER BY** 子句。 单击“下一步”。
12. 指定是允许在 **WINDOW FRAME** 子句中使用函数，需要在 **WINDOW FRAME** 子句中使用函数，还是不允许在 **WINDOW FRAME** 子句中使用函数。 单击“下一步”。

    如果不允许在 **WINDOW FRAME** 子句中使用函数，请跳到步骤 14。
13. 标识对 **WINDOW FRAME** 子句的约束。 单击“下一步”。
14. 指定是否需要在调用此函数前通过数据库服务器将重复输入值滤出。
15. 标识在不带任何数据调用函数时函数返回值是 **NULL**，还是固定值。 单击“下一步”。
16. 添加说明新函数用途的注释。 单击“完成”。
17. 在右窗格中，单击“SQL”选项卡以填写过程代码。

    新函数将显示在“过程和函数”中。

用户定义的函数限制

外部 C/C++ 用户定义的函数具有一些限制。

- 所有 UDF 的写入方式允许不同用户在接收不同上下文函数的同时调用这些 UDF。
- UDF 访问全局或共享数据结构时，UDF 定义必须实现其对该数据的访问的相应锁定，包括所有普通代码路径和所有错误处理情况下的该锁定的释放。
- C++ 中实现的 UDF 可为其类提供过载的“新”运算符，但绝不能过载全局的“新”运算符。 在某些平台上，这样做的影响不限于该特定库中定义的代码。
- 所有集合 UDF 以及所有确定型标量 UDF 的写入方式应当使相同输入值的接收始终能产生相同的输出值。 情形并非如此的任何标量函数必须声明为 NONDETERMINISTIC，以避免潜在的不正确回应。
- 用户无需 DBA 授权便可创建标准 SQL 函数，但他们无法在没有 DBA 权限的情况下创建将调用外部库的函数。 尝试执行此操作会产生一条错误消息“`You do not have permission to use the create function statement`”（您无权使用 create function 语句）。

调用用户定义的函数

在您使用内置的非集合函数的任何位置，都可以根据权限使用用户定义的函数。

以下 Interactive SQL 语句从包含名字和姓氏的两列返回全名：

```
SELECT fullname (GivenName, LastName)
FROM Employees;
```

fullname (Employees.GivenName,Employees.SurName)
Fran Whitney
Matthew Cobb
Philip Chin
...

以下语句从提供的名字和姓氏返回全名：

```
SELECT fullname ('Jane', 'Smith');
```

fullname ('Jane','Smith')
Jane Smith

已被授予函数的“执行”权限的任何用户都可以使用 *fullname* 函数。

设置动态库接口

指定要在动态可链接库中使用的接口样式。

每个动态装载的库只能包含此定义的一个副本：

```
extern "C" a_sql_uint32 extfn_use_new_api(void )
{
return EXTFN_V3_API;
}
```

此定义将通知服务器正在使用哪种接口样式，因此也通知服务器如何访问此动态可链接库中定义的 UDF。 对于高性能 IQ UDF，仅支持新的接口样式 (EXTFN_V3_API)。

删除用户定义的函数

用户定义的函数一经创建，便会保留在数据库中，直至它被显式删除。只有该函数的所有者或具有 DBA 权限的用户才可从数据库中删除函数。

例如，若要从数据库中删除 *fullname* 函数，请输入：

```
DROP FUNCTION fullname
```

授予和撤消权限

用户定义的函数由创建它的用户拥有，并且只有该用户无需权限便可执行它。 所有者可以使用 **GRANT EXECUTE** 命令向其他用户授予权限。

例如，*fullname* 函数的创建者可以通过发出以下命令允许 *another_user* 使用 *fullname*：

```
GRANT EXECUTE ON fullname TO another_user
```

或者可通过发出以下命令来撤消权限：

```
REVOKE EXECUTE ON fullname FROM another_user
```

请参见《系统管理指南：第一卷》>“管理用户 ID 和权限”>“授予针对过程的权限”。

用户定义的函数的维护

许多 Sybase IQ 安装都是在客户要求极高可用性的关键环境中。 系统管理员必须能够安装和升级 UDF，且对 Sybase IQ 服务器没有影响或影响很小。

在移动、覆盖或删除关联的库文件时，应用程序一定不要尝试访问外部库。 因为库会在调用相关的 SQL 函数时自动装载，所以在对现有 UDF 库执行任何类型的维护时，一定要完全按照下列步骤的顺序操作：

1. 确保调用 UDF 的所有用户没有任何正在进行的挂起查询
2. 撤消用户的执行权限，并删除引用外部 UDF 代码模块的 SQL 函数和存储过程
3. 使用 **call sa_external_library_unload** 命令从 IQ 服务器中卸载库（关闭 IQ 服务器也会自动卸载库）。
4. 对外部库文件执行所需的维护（复制、移动、更新、删除）。
5. 如果移动了库，请在注册脚本中编辑 SQL 函数和存储过程定义以反映外部库位置。
6. 授予用户执行权限，并运行注册脚本以重新创建引用外部 UDF 代码模块的 SQL 函数和存储过程。
7. 调用引用外部 UDF 代码的 SQL 函数或存储过程以确保 IQ 服务器可以动态装载外部库。

编译并链接源代码以构建动态可链接库

在为任何用户定义的函数构建动态可链接库时，使用编译和链接开关。

1. UDF 动态可链接库必须包含函数 **extfn_use_new_api()** 的实现。 此函数的源代码在“设置动态链接库接口”（第 15 页）中。此函数将库中所有函数需遵循的 API 样式通知服务器。 示例源文件 `my_main.cxx` 包含此函数，并且无需修改即可使用。
2. UDF 动态可链接库还必须包含至少一个 UDF 函数的对象代码。 UDF 动态可链接库可选择性地包含多个 UDF。
3. 将各个 UDF 的对象代码以及 **extfn_use_new_api()** 链接在一起，以形成单个库。

例如，若要构建库“libudfex”，请执行下列操作：

- 编译各个源文件以生成对象文件：

```
my_main.cxx
            my_bit_or.cxx
            my_bit_xor.cxx
            my_interpolate.cxx
            my_plus.cxx
            my_plus_counter.cxx
            my_sum.cxx
```

- 将各个对象链接在一起，生成单个库。

在编译并链接动态可链接库之后，请完成下列任务之一：

- （推荐）更新 CREATE FUNCTION ... EXTERNAL NAME，以包含 UDF 库的显式路径名称。
- 将 UDF 库文件放入存储所有 IQ 库的目录中。

- 用包含 UDF 库位置的库装载路径启动 IQ 服务器。
  在 UNIX 上，修改 `start_iq startup` 脚本中的 LD_LIBRARY_PATH。 虽然 LD_LIBRARY_PATH 对所有 UNIX 变体都通用，但在 HP 上首选使用 SHLIB_PATH，在 AIX 上首选使用 LIB_PATH。
  在 UNIX 平台上，外部名称规范可以包含完全限定名，这种情况下不使用 LD_LIBRARY_PATH。 在 Windows 平台上，无法使用完全限定名，且库搜索路径由 PATH 环境变量定义。

AIX 开关

在 AIX 上构建共享库时使用下列编译和链接开关。

PowerPC 上的 xlC 8.0

重要： 在每个 UDF 库中包括 extfn_use_new_api() 的代码。

注意： 若要在 AIX 6.1 系统上进行编译，xlC 编译器的最低级别为 8.0.0.24。

编译开关

```
-q64 -qarch=ppc64 -qtbtable=full -qsrcmsg -qalign=natural -
qnoansialias
-qmaxmem=-1 -qenum=int -qhalt=e -qflag=w -qthreaded -
qxflags=NLOOPING
-qtmplinst=none -qthreaded
```

链接开关

```
-brtl -G -lg  -lpthreads_compat -lpthreads -lm_r -ldl -bnolibpath -
v
```

HP-UX 开关

在 HP-UX 上构建共享库时使用下列编译和链接开关。

Itanium 上的 aCC 6.17

重要： 在每个 UDF 库中包括 extfn_use_new_api() 的代码。

编译开关

```
+noeh -ext +W740,749,829 +W1031 +DD64 +DSblended +FPD -Aa +ub
-U_HP_INSTANTIATE_T_IN_LIB -Wc,-ansi_for_scope,on -mt -z
```

链接开关

```
 -b -Wl,+s
```

Linux 开关

在 Linux 上构建共享库时使用下列编译和链接开关。

x86 上的 g++ 4.1.1

重要： 在每个 UDF 库中包括 extfn_use_new_api() 的代码。

编译开关

```
-fPIC -fsigned-char -fno-exceptions -pthread -fno-omit-frame-
pointer
-Wno-deprecated -Wno-ctor-dtor-privacy
```

链接开关

```
-ldl -lnsl -lm -lpthread -shared -Wl,-Bsymbolic -Wl,-shared
```

注意： 也可在 Linux 上使用 gcc。 用 **gcc** 链接时，可通过将 `-lstdc++` 添加到链接开关，在 C++ 运行时库中进行链接。

示例

- *示例 1*

  ```
  g++ -c my_interpolate.cxx -fPIC -fsigned-char -fno-exceptions -
  pthread
        -fno-omit-frame-pointer -Wno-deprecated -Wno-ctor-dtor-
  privacy
        -I${IQDIR15}/sdk/include/
  ```

- *示例 2*

  ```
  g++ -c my_main.cxx -fPIC -fsigned-char -fno-exceptions -pthread
        -fno-omit-frame-pointer -Wno-deprecated -Wno-ctor-dtor-
  privacy
        -I${IQDIR15}/sdk/include/
  ```

- *示例 3*

  ```
  ld -G my_main.o my_interpolate.o -ldl -lnsl -lm -lpthread -shared
        -o my_udf_library.so
  ```

PowerPC 上的 xlC 8.0

编译开关

```
 -q64 -qarch=ppc64 -qcheck=nullptr -qinfo=gen -qtbtable=full -
qsrcmsg
-qnoansialias -qminimaltoc -qmaxmem=-1 -qenum=int -qhalt=e -qflag=w
-qthreaded
-qxflags=NLOOPING -qtmplinst=none
```

链接开关

```
-qmkshrobj -ldl -lg  -qthreaded -lnsl -lm
```

Solaris 开关

在 Solaris 上构建共享库时使用下列编译和链接开关。

SPARC 上的 Sun Studio 12

重要： 在每个 UDF 库中包括 extfn_use_new_api() 的代码。

编译开关

```
-mt -noex +w -KPIC -i -instances=explicit -V -xtarget=ultra3cu -m64
-xlibmopt
-xlibmil -features=no%conststrings
-erroff=truncwarn,nokeyworddefine,diffenumtype
```

链接开关

```
-z defs -G -ldl -lnsl -lsocket -ladm -lposix4 -lCrun -lCstd -lc -lm
-lefi
-liostream -lkstat
```

x86 上的 Sun Studio 12

编译开关

```
+w2 -m64 -features=no%conststrings
-erroff=truncwarn,nokeyworddefine,diffenumtype,doubunder -errtags -
mt  -noex
-KPIC -instances=explicit -xlibmopt -xlibmil
```

链接开关

```
-z defs -G -ldl -lnsl -lsocket -ladm -lposix4 -lCrun -lCstd -lc -lm
-lefi
-liostream -lkstat -m64
```

Windows 开关

在 Windows 上构建共享库时使用下列编译和链接开关。

x86 上的 Visual Studio 2008

重要： 在每个 UDF 库中包括 extfn_use_new_api() 的代码。

编译和链接开关

本示例用于包含 my_plus 函数的 DLL。必须为 DLL 中所含的每个 UDF 的描述符函数包含一个 EXPORT 开关。

```
cl /Zi /LD   /I includefilepath     my_main.cxx my_plus.cxx  /link /
map
/INCREMENTAL:NO -EXPORT:extfn_use_new_api -EXPORT:my_plus  /
out:libiqudfex.dll
```

示例

环境设置

```
set VCBASE=c:\dev\vc9
      set MSSDK=C:\dev\mssdk6.0a
      set IQINSTALLDIR=C:\Sybase\IQ
      set OBJ_DIR=%IQINSTALLDIR%\IQ-15_1\samples\udf\objs
      set SRC_DIR=%IQINSTALLDIR%\IQ-15_1\samples\udf\src
      call %VCBASE%\VC\bin\vcvars32.bat
```

- *示例 1*

```
%VCBASE%\VC\bin\amd64\cl -c -nologo -DNDEBUG -DWINNT -D_USRDLL
      -D_WINDLL -D_WIN64 -DWIN64 -
D_WIN32_WINNT=_WIN32_WINNT_WINXP
     -DWINVER=_WIN32_WINNT_WINXP -D_MBCS -GS -W3 -Zi -favor:AMD64
      -DSYB_LP64 -D_LARGEFILE_SOURCE -D_FILE_OFFSET_BITS=64 -
DHMSWNT
      -D_CRT_SECURE_NO_DEPRECATE -D_CRT_NONSTDC_NO_DEPRECATE
      -DPOINTERS_ARE_64BITS -DLONG_IS_64BITS -
D_RWSTD_NO_EXCEPTIONS
      -I"%VCBASE%\VC\include" -I"%MSSDK%\include "-I"%MSSDK%\Lib
\AMD64"
      -I"%VCBASE%\VC\lib\amd64" -DMSDCXX -DINT64_WORKAROUND
      -DSUPPORTS_UDAF -Od -Zi -MD -I"%IQINSTALLDIR%\IQ-15_1\sdk
\include"
     -Fo"%OBJ_DIR%\my_interpolate.o" %SRC_DIR%\my_interpolate.cxx
```

- *示例 2*

```
%VCBASE%\VC\bin\amd64\cl -c -nologo -DNDEBUG -DWINNT -D_USRDLL
      -D_WINDLL -D_WIN64 -DWIN64 -
D_WIN32_WINNT=_WIN32_WINNT_WINXP
     -DWINVER=_WIN32_WINNT_WINXP -D_MBCS -GS -W3 -Zi -favor:AMD64
      -DSYB_LP64 -D_LARGEFILE_SOURCE -D_FILE_OFFSET_BITS=64 -
DHMSWNT
      -D_CRT_SECURE_NO_DEPRECATE -D_CRT_NONSTDC_NO_DEPRECATE
      -DPOINTERS_ARE_64BITS -DLONG_IS_64BITS -
D_RWSTD_NO_EXCEPTIONS
      -I"%VCBASE%\VC\include" -I"%MSSDK%\include "-I"%MSSDK%\Lib
\AMD64"
      -I"%VCBASE%\VC\lib\amd64" -DMSDCXX -DINT64_WORKAROUND
      -DSUPPORTS_UDAF -Od -Zi -MD -I"%IQINSTALLDIR%\IQ-15_1\sdk
\include"
      -Fo"%OBJ_DIR%\my_main.o" %SRC_DIR%\my_main.cxx
```

- *示例 3*

```
%VCBASE%\VC\bin\amd64\link /LIBPATH:%VCBASE%\VC\lib\amd64
      /LIBPATH:%MSSDK%\lib\bin64 kernel32.lib -manifest -DLL -
nologo
      -MAP:"%OBJ_DIR%\libudfex.map_deco" /OUT:"%OBJ_DIR%
\libudfex.dll"
      "%OBJ_DIR%\my_interpolate.o" "%OBJ_DIR%\my_main.o" /DLL
      -EXPORT:extfn_use_new_api -EXPORT:my_interpolate
```

- *示例 4*

```
%MSSDK%\bin\mt -nologo -manifest "%OBJ_DIR%
\libudfex.dll.manifest"
      -outputresource:"%OBJ_DIR%\libudfex.dll;2"
```

对用户定义的函数使用 Microsoft Visual Studio 调试程序

这些步骤将使 Microsoft Visual Studio 2008 开发人员能够分步完成用户定义的函数代码。

1. 将调试程序附加到运行的服务器：

   ```
   devenv /debugexe "%IQDIR15%\bin32\iqsrv15.exe"
   ```

2. 转到“调试”(Debug) |“附加到进程”(Attach to Process)
3. 若要同时启动服务器和调试程序，请执行下列操作：

   ```
   devenv /debugexe "%IQDIR15%\bin32\iqsrv15.exe" [commandline
   options for your server]
   ```

   每个平台都将有调试程序且每个都将有自己的命令行语法。无需 Sybase IQ 源代码。msvs 调试程序将识别用户定义的函数源的执行时间和在设置的断点中断的时间。当控制从用户定义的函数返回到服务器时，您将只看到计算机代码。

SQL 数据类型

UDF 声明仅支持某些 SQL 数据类型。

可以在 UDF 声明中作为 UDF 参数的数据类型或返回值数据类型使用以下 SQL 数据类型：

- UNSIGNED BIGINT – 不带符号的 64 位整数，需要 8 个字节的存储空间。要在 UDF 代码中使用的数据类型标识符为 DT_UNSBIGINT，要对 UDF 中的这种值使用的 C/C++ 数据类型 typedef 为“a_sql_uint64”。Sybase IQ 中附带了多个 C/C++ typedef，以便应用程序开发人员可以更方便地编写可移植 UDF 实现。
- BIGINT – 带符号的 64 位整数，需要 8 个字节的存储空间。数据类型标识符为 DT_BIGINT，要对这种值使用的 C/C++ 数据类型 typedef 为“a_sql_int64”。
- UNSIGNED INT – 不带符号的 32 位整数，需要 4 个字节的存储空间。数据类型标识符为 DT_UNSINT，要对这种值使用的 C/C++ 数据类型 typedef 为“a_sql_uint32”。
- INT – 带符号的 32 位整数，需要 4 个字节的存储空间。数据类型标识符为 DT_INT，要对这种值使用的 C/C++ 数据类型 typedef 为“a_sql_int32”。
- SMALLINT – 带符号的 16 位整数，需要 2 个字节的存储空间。数据类型标识符为 DT_SMALLINT，要对这种值使用的 C/C++ 数据类型为“short”。
- TINYINT – 不带符号的 8 位整数，需要 1 个字节的存储空间。数据类型标识符为 DT_TINYINT，要对这种值使用的 C/C++ 数据类型为“unsigned char”。
- DOUBLE – 带符号的 64 位双精度浮点数，需要 8 个字节的存储空间。数据类型标识符为 DT_DOUBLE，要对这种值使用的 C/C++ 数据类型为“double”。

- REAL – 带符号的 32 位浮点数，需要 4 个字节的存储空间。 数据类型标识符为 DT_FLOAT，要对这种值使用的 C/C++ 数据类型为“float”。
- FLOAT – 在 SQL 中，根据关联的精度的不同，FLOAT 可以为需要 4 个字节的存储空间的带符号 32 位浮点数，也可以为需要 8 个字节的存储空间的带符号 64 位双精度浮点数。 仅当未提供 FLOAT 数据类型的可选精度时，才能在 UDF 声明中使用 SQL 数据类型 FLOAT。 如果没有提供精度，FLOAT 则是 REAL 的同义词，其数据类型标识符为 DT_FLOAT，要对这种值使用的 C/C++ 数据类型为“float”。
- CHAR(<n>) – 数据库缺省字符集中用空白填充的固定长度字符串。 最大长度“<n>”为 32767。数据不会在空字节处终止。 数据类型标识符为 DT_FIXCHAR，要对这种值使用的 C/C++ 数据类型为“char *”。
- VARCHAR(<n>) – 数据库缺省字符集中的可变长度字符串。 最大长度“<n>”为 32767。数据不会在空字节处终止。 对于 UDF 输入参数，当值不为空值时，必须从 an_extfn_value 结构中的 total_length 字段检索实际长度。 同样，对于此类型的 UDF 结果，必须在 total_length 字段中设置实际长度。 数据类型标识符为 DT_VARCHAR，要对这种值使用的 C/C++ 数据类型为“char *”。
- BINARY(<n>) – 用空字节填充的固定长度二进制值，其最大二进制长度“<n>”为 32767。数据不会在空字节处终止。 数据类型标识符为 DT_BINARY，对这种值通常使用的 C/C++ 数据类型为“unsigned char *”。
- VARBINARY(<n>) – 可变长度二进制值，其最大长度“<n>”为 32767。数据不会在空字节处终止。 对于 UDF 输入参数，当值不为空值时，必须从 an_extfn_value 结构中的 total_length 字段检索实际长度。 同样，对于此类型的 UDF 结果，必须在 total_length 字段中设置实际长度。 数据不会在空字节处终止。 数据类型标识符为 DT_BINARY，对这种值通常使用的 C/C++ 数据类型为“unsigned char *”。
- DATE – 日历日期值，该值作为不带符号的整数传递至 UDF 或从 UDF 中传出。 确保 UDF 的给定值在比较和排序运算中可用。 值越大表示日期越晚。 如果需要实际日期组件，则 UDF 必须调用 convert_value api 才能转换为 DT_TIMESTAMP_STRUCT 类型。 此数据类型代表具有以下结构的日期和时间：

```
typedef struct sqldatetime {
    unsigned short    year;         /* e.g. 1992            */
    unsigned char     month;        /* 0-11                 */
   unsigned char    day_of_week;    /* 0-6  0=Sunday, 1=Monday, ...
*/
    unsigned short    day_of_year;     /* 0-365              */
    unsigned char     day;         /* 1-31                  */
    unsigned char     hour;         /* 0-23                 */
    unsigned char     minute;     /* 0-59                   */
    unsigned char     second;     /* 0-59                   */
    a_sql_uint32      microsecond;    /* 0-999999           */
} SQLDATETIME;
```

- TIME – 精确说明某一给定的日子的某一时刻的值。此值作为 UNSIGNED BIGINT 传递给 UDF。 确保 UDF 的给定值在比较和排序运算中可用。 值越大表示时间越晚。 如果需要实际时间组件，则 UDF 必须调用 convert_value 才能转换为 DT_TIMESTAMP_STRUCT 类型。
- DATETIME、SMALLDATETIME 或 TIMESTAMP – 日历日期和时间值，该值作为 UNSIGNED BIGINT 传递至 UDF 或从 UDF 中传出。 确保 UDF 的给定值在比较

和排序运算中可用。 值越大表示日期时间越晚。 如果需要实际时间组件，则 UDF 必须调用 convert_value 才能转换为 DT_TIMESTAMP_STRUCT 类型。

不支持的数据类型

无法在 UDF 声明中作为 UDF 参数的数据类型或返回值数据类型使用以下 SQL 数据类型：

- BIT – 通常应作为 TINYINT 数据类型在 UDF 声明中进行处理，随后，从 BIT 隐式转换的数据类型将自动处理值转换。
- DECIMAL(<precision>, <scale>) 或 NUMERIC(<precision>, <scale>) – 根据使用情况的不同，DECIMAL 通常作为 DOUBLE 数据类型进行处理，但可能需要强制进行各种转换才能使用 INT 或 BIGINT 数据类型。
- LONG VARCHAR – 当前不受支持。
- LONG BINARY – 当前不受支持。
- TEXT – 当前不受支持。

测试用户定义的函数

在对 UDF 外部代码进行了编码、编译和链接且定义了相应的 SQL 函数和存储过程后，便可对 UDF 进行测试。

数据库要求的可靠性极高。 数据库环境中运行的 UDF 必须保持此高水平的可靠性。第一次实现 UDF API 后，UDF 在 Sybase IQ 服务器中运行。 如果 UDF 过早或意外中止，则 Sybase IQ 服务器可能中止。 通过在开发或测试环境中进行彻底的测试，确保 UDF 在任何情况下都不会过早终止或意外中止。

启用和禁用用户定义的函数

Sybase IQ 包括安全性功能 external_procedure_v3，该功能使服务器能够使用或禁止其使用高性能进程中 UDF。

数据库应保持数据完整性。在任何情况下，都不应丢失、修改、增加或损坏数据。 由于 UDF 在 Sybase IQ 服务器中执行，存在损坏数据的风险，因此，管理内存及使用任何其它指针时要小心。 Sybase 强烈建议您在只读 Multiplex 节点中安装和执行 UDF。为增加保护，请在每个 IQ 服务器中使用启动选项来启用或禁用 UDF 的执行。

注意： 缺省情况下，禁用 Multiplex 写入程序和事务协调器节点上的 UDF 执行。 缺省情况下，将启用所有其它节点。

管理员可以通过在服务器启动命令中或在配置文件中指定以下代码，为任何服务器启用第 3 版 UDF：

```
-sf -external_procedure_v3
```

管理员可以通过在服务器启动命令中或在配置文件中指定以下代码，为任何服务器禁用第 3 版 UDF：

```
-sf external_procedure_v3
```

《SQL Anywhere Server — 数据库管理》指南中提供了有关 -sf 标志的其它信息。 不要使用 SQL Anywhere 文档中列出的适用于 Sybase IQ 的值。

初次执行用户定义的函数

为确保可能的最安全的环境，Sybase 强烈建议您从 Multiplex 安装中的只读 IQ 服务器节点安装并调用 UDF。

Sybase IQ 服务器不会在首次调用 UDF 之前装载包含 UDF 代码的库。 首次执行驻留在尚未装载的库中的 UDF 时可能会异常缓慢。 在库装载后，以后对同一 UDF 或同一库中包含的其它 UDF 进行调用时将具有预期性能。

使用存储过程 SA_EXTERNAL_LIBRARY_UNLOAD 的库。 在 IQ 服务器停止并重新启动时，不会重新装载这些库。

在下班以后的维护操作需要关闭和重新启动 IQ 服务器的环境中，在 Sybase IQ 服务器重新启动后运行一些测试查询。 这样可以确保相应的库已装载到内存中，以在工作时间内实现最优查询性能。

管理外部库

每个外部库在首次调用需要它的 UDF 时装载。 已装载的库在服务器有效期内保持装载状态。 它不会在调用 CREATE FUNCTION 时进行装载，也不会在调用 DROP FUNCTION 时自动卸载。

如果库版本必须更新，**dbo.sa_external_library_unload** 过程会强制卸载该库，但不重新启动服务器。 执行外部库卸载的调用只有在该库当前未在使用时才会成功。 该过程会采用一个可选的 long varchar 参数，该参数指定要卸载的库的名称。 如果未指定任何参数，则未在使用的所有外部库都会被卸载。

注意： 在替换动态链接库之前，从运行中的 Sybase IQ 服务器卸载现有的库。 若未能将库卸载，可导致服务器崩溃。 在替换动态可链接库前，可关闭 Sybase IQ 服务器，或者使用 sa_external_library_unload 函数将库卸载。

对于 Windows，使用以下命令卸载外部函数库：

```
call sa_external_library_unload('library')
```

对于 UNIX，使用以下命令卸载外部函数库：

```
call sa_external_library_unload('library.so')
```

如果已注册的函数使用完整的路径（如 `/abc/def/library`），请先注销该函数。

在 Windows 中，请使用

```
call sa_external_library_unload('/abc/def/library')
```

在 UNIX 中，请使用

```
call sa_external_library_unload('\abc\def\library.so')
```

注意： 仅当库尚未位于库装载路径中的目录中时，SQL 函数声明中才需要库路径。

控制错误检查和调用跟踪

external_UDF_execution_mode 选项控制在涉及外部 V3 用户定义函数的语句求值时所执行的错误检查和调用跟踪的数量。

可以在 UDF 开发期间使用 **external_UDF_execution_mode** 来帮助开发 UDF 时进行的调试。

允许值
0, 1, 2

缺省值
0

范围
可设为公共、临时或用户。

说明
设为 0（缺省值）时，外部 UDF 求值的方式可以优化使用 UDF 的语句的性能。

设为 1 时，外部 UDF 求值的方式可以验证向每个 UDF 函数来回传递的信息。

设为 2 时，外部 UDF 的求值不仅可以验证向 UDF 来回传递的信息，还可在 `iqmsg` 文件中记录对由 UDF 提供的函数的每次调用以及从这些函数返回到服务器的每次回调。

在调试环境中启用完整跟踪

在调试环境中，考虑启用 Sybase IQ 内的完整跟踪。 若要启用 IQ 跟踪，请将以下标志添加到 Sybase IQ 服务器启动命令行或 Sybase IQ 配置文件中：

```
-zr all -zo filename
```

其中，*filename* 是跟踪输出文件的完整路径。

查看 Sybase IQ 日志文件

Sybase IQ 提供了大量日志记录和跟踪功能。 在 UDF 代码出现问题时，UDF 应提供相同或更好的详细日志记录。

数据库的日志文件通常与数据库文件和配置文件在相同位置。 在 Unix 平台上，有两个以数据库实例命名的文件，一个文件的扩展名为 .stderr，另一个文件的扩展名为 .stdout。 在 Windows 上，缺省情况下，不会生成 stderr 文件。 若要在 Windows 下捕获 stderr 消息以及 stdout 消息，请重定向 stdout 和 stderr：

```
iqsrv15.exe @iqdemo.cfg iqdemo.db 2>&1 > iqdemo.stdout
```

Windows 输出消息与 Unix 平台上生成的输出消息略有不同。

用户定义的标量函数

Sybase IQ 支持可在任何可以使用 SQRT 函数的位置使用的简单的用户定义的标量函数 (UDF)。

这些标量 UDF 可以是确定型函数（这意味着对于一组给定的参数值，函数始终返回相同的结果值），也可以是非确定型标量函数（这意味着相同的参数可以返回不同结果）。

注意： 本章中引用的标量 UDF 示例随 IQ 服务器一起安装，这些示例可作为 .cxx 文件存在于 `$IQDIR15/samples/udf` 下。 您还可以在 `$IQDIR15/lib64/libudfex` 动态可链接库中找到这些示例。

声明标量 UDF

只有 DBA 或具有 DBA 权限的用户才可以声明进程中外部 UDF。 还有一个服务器启动选项，允许管理员启用或禁用此样式的用户定义的函数。

编写并编译了 UDF 代码后，创建一个从相应库文件中调用 UDF 并将输入数据发送到 UDF 的 SQL 函数。

注意： 您还可以“在 Sybase Central 中创建用户定义的函数声明” （第 12 页）。

缺省情况下，所有用户定义的函数使用 UDF 的所有者的访问权限。

注意： 用户需要具有 DBA 授权才能声明 UDF 函数。

创建 IQ 标量 UDF 的语法为：

```
scalar-udf-declaration:
CREATE FUNCTION [ owner.]function-name
     ( [ parameter , ... ] )
RETURNS data-type
     [ routine-characteristics ... ]
EXTERNAL NAME library-and-entry-point-name-string

parameter:
    param-name data-type [ DEFAULT value ]

routine-characteristics:
     [NOT] DETERMINISTIC
  | { IGNORE | RESPECT } NULL VALUES
     | SQL SECURITY { INVOKER | DEFINER }
```

上述语法中的特性的缺省值为：

```
DETERMINISTIC
RESPECT NULL VALUES
SQL SECURITY DEFINER
```

为了最大限度地消除安全隐患，Sybase 建议您对 EXTERNAL NAME 子句的库名称部分的安全目录使用完全限定路径名。

SQL 安全性

定义是作为 INVOKER（调用函数的用户）还是作为 DEFINER（拥有函数的用户）执行函数。缺省为 DEFINER。

SQL SECURITY INVOKER 使用更多内存，这是因为调用过程的每个用户都需要注释。此外，将同时对用户名和 INVOKER 执行名称解析。 使用所有对象名（表、过程等）的相应所有者限定这些名称。

外部名称

使用 **EXTERNAL NAME** 子句的函数是包含对外部库函数调用的包装。使用 **EXTERNAL NAME** 的函数在 **RETURNS** 子句后可以不使用其它子句。库名可包含文件扩展名，在 Windows 中通常为 `.dll`，在 UNIX 中通常为 `.so`。在没有扩展名的情况下，该软件附加平台特定的缺省库文件扩展名。

临时函数不支持 **EXTERNAL NAME** 子句。请参见《SQL Anywhere Server — 编程》中的“从过程调用外部库”。

可以通过包括 UDF 库位置的库装载路径启动 IQ 服务器。 在 Unix 变体中，修改 `start_iq startup` 脚本中的 LD_LIBRARY_PATH。 虽然 LD_LIBRARY_PATH 对所有 UNIX 变体都通用，但在 HP 上首选使用 SHLIB_PATH，在 AIX 上首选使用 LIB_PATH。

在 Unix 平台上，外部名称规范可以包含完全限定名，这种情况下不使用 LD_LIBRARY_PATH。在 Windows 平台上，无法使用完全限定名，且库搜索路径由 PATH 环境变量定义。

注意： 可更新游标中不支持标量用户定义函数和用户定义集合函数。

UDF 示例：my_plus 声明

“my_plus”示例是返回将函数的两个整数参数值相加所得结果的简单标量函数。

my_plus 声明

如果 my_plus 驻留在动态可链接库 my_shared_lib 中，此示例的声明将类似于：

```
CREATE FUNCTION my_plus (IN arg1 INT, IN arg2 INT)
     RETURNS INT
     DETERMINISTIC
     IGNORE NULL VALUES
     EXTERNAL NAME 'my_plus@libudfex'
```

此声明指出 my_plus 是一个简单标量 UDF，该函数驻留在具有名为 describe_my_plus 的描述符例程的 my_shared_lib 中。由于 UDF 的行为可能需要多个实际 C/C++ 入口点进行实现，因此，此组入口点未直接包含在 CREATE FUNCTION 语法中。其实，CREATE FUNCTION 语句的 EXTERNAL NAME 子句标识了此 UDF 的描述符函数。描述符函数在调用时将返回描述符结构，下一节中对此结构进行了详细定义。该描述符结构包含了体现此 UDF 实现的必要和可选函数指针。

此声明指出 my_plus 接受两个 INT 参数并返回 INT 结果值。如果使用不是 INT 的参数调用此函数，且该参数可以隐式转换为 INT，则在调用此函数之前将进行转换。如果使用无法隐式转换为 INT 的参数调用此函数，将生成转换错误。

而且，该声明还指出此函数是确定型函数。确定型函数在提供了相同的输入值的情况下始终返回相同的结果值。这意味着结果不依赖于除提供的参数值之外的任何外部信息，也不受以前调用所产生的任何副作用的影响。缺省情况下，假定函数是确定型函数，因此，如果忽略 CREATE 语句中的此特性，结果将相同。

上述声明的最后一部分是 IGNORE NULL VALUES 特性。如果任何输入参数为空值，则几乎所有内置标量函数都将返回空结果值。IGNORE NULL VALUES 指出 my_plus 函数遵守该约定，因此，如果其任一输入值为空值，则实际不会调用此 UDF 例程。由于 RESPECT NULL VALUES 是函数的缺省值，因此必须在此 UDF 的声明中指定此特性才能获取性能优势。在给定空输入值时可能返回非空结果的所有函数，必须使用缺省的 RESPECT NULL VALUES 特性。

在以下查询示例中，my_plus 显示在 SELECT 列表以及等效算术表达式中：

```
SELECT my_plus(t.x, t.y) AS x_plus_y_one, (t.x + t.y)AS x_plus_y_two
FROM t
WHERE t.z = 2
```

在以下示例中，以不同方式在同一查询的多个不同位置使用 my_plus：

```
SELECT my_plus(t.x, t.y), count(*)
FROM t
WHERE t.z = 2
AND my_plus(t.x, 5) > 10
AND my_plus(t.y, 5) > 10
GROUP BY my_plus(t.x, t.y)
```

UDF 示例：my_plus_counter 声明

“my_plus_counter”示例是一个简单非确定型标量 UDF，该函数采用单个整数参数，并返回将该参数值与内部整数使用情况计数器中的值相加所得的结果。如果输入参数值为空值，则结果为使用情况计数器的当前值。

my_plus_counter 声明

假设 my_plus_counter 也位于动态链接库 my_shared_lib 内，此示例的声明如下：

```
CREATE FUNCTION my_plus_counter (IN arg1 INT DEFAULT 0)
     RETURNS INT
     NOT DETERMINISTIC
```

```
    RESPECT NULL VALUES
    EXTERNAL NAME 'describe_my_plus_counter@my_shared_lib'
```

RESPECT NULL VALUES 特性表示，即使输入参数的值为 NULL 也会调用该函数。这确实有必要，因为 my_plus_counter 的语义中包含如下含义：

- 内部保留一个使用计数，即使参数为 NULL 时该计数也会递增。
- 传递空值参数时也会生成非空值。

因为 RESPECT NULL VALUES 是缺省值，所以即使声明中省略该子句，结果也完全相同。

IQ 限制所有非确定性函数的使用。这些函数只能用于顶级查询块的 SELECT 列表中，或用于 UPDATE 语句的 SET 子句中。它们不能用于子查询中或 WHERE、ON、GROUP BY 或 HAVING 子句中。此限制适用于非确定性 UDF 以及非确定性内置函数，例如 GETUID 或 NUMBER。

如上声明中需要说明的最后一点是输入参数上的 DEFAULT 限定符。该限定符向服务器指明，调用该函数时可以不带参数，届时服务器会针对缺少的参数自动以零作为其参数值。如果指定缺省值，则它必须可以隐式转换为该参数的数据类型。

在下面的示例中，第一个 SELECT 列表项将运行的计数器添加到各行的 t.x 的值上。第二个和第三个 SELECT 列表项返回的各行的值均与 NUMBER 函数的返回值相同。

```
SELECT my_plus_counter(t.x),
     my_plus_counter(0),
     my_plus_counter(),
     NUMBER()
FROM t
```

定义标量 UDF

用于定义标量用户定义的函数的 C/C++ 代码包含 4 个必需项。

- **extfnapiv3.h** – 包含 UDF 接口定义头文件。
- **_evaluate_extfn** — 求值函数。所有求值函数都有两个参数：
  - 标量 UDF 上下文结构的一个实例，实例在每次使用 UDF 时都是唯一的，其中包含一组回调函数指针以及一个 UDF 可用于存储 UDF 专用数据的指针。
  - 指向数据结构的指针，允许访问参数值以及通过所提供回调生成的结果值。
- **a_v3_extfn_scalar** – 标量 UDF 描述符结构的一个实例，其中包含指向求值函数的一个指针。
- **Descriptor function** – 返回指向标量 UDF 描述符结构的一个指针。

这些部分是可选的：

- **_start_extfn** – 一个初始化函数，每次使用 SQL 时通常调用一次。 如果提供此项，您还必须在标量 UDF 描述符结构中包含一个指向该函数的指针。 所有初始化函数均使用一个参数，即一个指向标量 UDF 上下文结构的指针，每次使用 UDF

时该指针都是唯一的。 所传递的上下文结构与向求值例程中传递的上下文结构相同。

- **_finish_extfn** – 一个关闭函数，每次使用 SQL 时通常调用一次。 如果提供此项，您还必须在标量 UDF 描述符结构中包含一个指向该函数的指针。 所有关闭函数均使用一个参数，即一个指向标量 UDF 上下文结构的指针，每次使用 UDF 时该指针都是唯一的。 所传递的上下文结构与向求值例程中传递的上下文结构相同。

标量 UDF 描述符结构

标量 UDF 描述符结构 a_v3_extfn_scalar 定义如下：

```
typedef struct a_v3_extfn_scalar {    //
    // Metadata descriptor for a scalar UDF
    // supplied by the UDF library to the server
    // An optional pointer to an initialize function
    void (*_start_extfn)(a_v3_extfn_scalar_context * cntxt);
    //
    // An optional pointer to a shutdown function
    void (*_finish_extfn)(a_v3_extfn_scalar_context *    cntxt);
    //
    // A required pointer to a function that will be
    // called for each invocation of the UDF on a
    // new set of argument values
    void (*_evaluate_extfn)(a_v3_extfn_scalar_context * cntxt, void
*args_handle);
    // RESERVED FIELDS MUST BE INITIALIZED TO NULL
 void  *reserved1_must_be_null;
 void  *reserved2_must_be_null;
 void  *reserved3_must_be_null;
 void  *reserved4_must_be_null;
 void  *reserved5_must_be_null;
                  …

} a_v3_extfn_scalar;
```

对于每个定义的标量 UDF，始终应该有一个 **a_v3_extfn_scalar** 实例。 如果没有提供可选初始化函数，则描述符结构中的对应值应该为空指针。 同理，如果没有提供关闭函数，则描述符结构中的对应值应为空指针。

在调用任何求值例程之前至少要调用一次初始化函数，而在最后一次调用求值函数之后至少要调用一次关闭函数。初始化函数和关闭函数通常在每次使用时只调用一次。

标量 UDF 上下文结构

传递至标量 UDF 描述符结构内指定的每个函数的标量 UDF 上下文结构 **a_v3_extfn_scalar_context** 定义如下：

```
typedef struct a_v3_extfn_scalar_context {
//-------- Callbacks available via the context --------
//
  short (SQL_CALLBACK *get_value)(
    void           *arg_handle,
```

```
            a_sql_uint32    arg_num,
            an_extfn_value *value
            );
short (SQL_CALLBACK *get_piece)(
            void *          arg_handle,
            a_sql_uint32    arg_num,
            an_extfn_value *value,
            a_sql_uint32    offset
            );
short (SQL_CALLBACK *get_value_is_constant)(
            void *          arg_handle,
            a_sql_uint32    arg_num,
            a_sql_uint32 *  value_is_constant
            );
short (SQL_CALLBACK *set_value)(
            void *          arg_handle,
            an_extfn_value *value,
            short              append
            );
a_sql_uint32 (SQL_CALLBACK *get_is_cancelled)(
            a_v3_extfn_scalar_context * cntxt
            );
short (SQL_CALLBACK *set_error)(
            a_v3_extfn_scalar_context * cntxt,
            a_sql_uint32    error_number,
            const char *    error_desc_string
            );
void (SQL_CALLBACK *log_message)(
            const char *msg,
            short msg_length
               );
short (SQL_CALLBACK *convert_value)(
               an_extfn_value   *input,
               an_extfn_value   *output
//---------- Data available from the context ----------
void * _user_data;   // read-write field
//---------- For Server Internal Use Only -------------
void  * _for_server_internal_use;
} a_v3_extfn_scalar_context;
```

可将 UDF 需要的数据填充在标量 UDF 上下文结构内的 _user_data 字段中。通常，该字段中将由 _start_extfn 函数填入已分配堆的结构，由 _finish_extfn 函数释放。

标量 UDF 上下文结构的其余部分将被填入引擎提供一组回调函数，以便在每个用户的 UDF 函数内使用。这些回调函数多数会通过一个短结果值返回成功状态；返回 true 则表示成功。编写良好的 UDF 实现不会导致故障状态，但在开发期间（也可能在给定 UDF 库的所有内部调试中），Sybase 建议您检查回调返回的状态值。故障可能源自 UDF 实现内的编码错误，例如要求提供的参数数量超过 UDF 定义使用的参数数量。

多数回调使用的一般参数集包括：

- **arg_handle** — 可由所有形式的求值方法接收的一个指针，通过该指针提供传递给 UDF 的输入参数的值，还可以通过该指针设置 UDF 结果值。

- **arg_num** — 表示将被访问的输入参数的一个整数。输入参数将从 1 开始，按从左到右的升序顺序编号。
- **cntxt** — 指向服务器传递给所有 UDF 入口点的上下文结构的指针。
- **value** — 指向 an_extfn_value 结构的一个实例的指针，用于从服务器获得输入参数值，或用于设置函数的结果值。an_extfn_value 结构的格式如下：

```
typedef struct an_extfn_value {
void * data;
a_SQL_uint32 piece_len;
union {
        a_SQL_uint32 total_len;
        a_SQL_uint32 remain_len;
} len;
a_SQL_data_type type;
} an_extfn_value;
```

UDF 示例：my_plus 定义

my_plus 示例的定义。

my_plus 定义

由于此 UDF 不需要初始化函数或关闭函数，因此描述符结构内的对应值将被设置为 0。描述符函数名称与声明中使用的 EXTERNAL NAME 相匹配。求值方法不检查参数的数据类型，因为它们被声明为 INT 类型。

```
#include "extfnapiv3.h"
#include <stdlib.h>


//  A simple deterministic scalar UDF that just adds
//  two integer arguments and then returns the result.
//
//  Corresponding SQL declaration:
//
//    CREATE FUNCTION my_plus(IN arg1 INT, IN arg2 INT)
//                                            RETURNS INT
//                                            DETERMINISTIC
//                                    IGNORE NULL VALUES
//                                            EXTERNAL NAME
'my_plus@libudfex'
//
#if defined __cplusplus
extern "C" {
#endif

static void my_plus_evaluate(a_v3_extfn_scalar_context *cntxt,
                      void *arg_handle)
{
  an_extfn_value  arg;
  an_extfn_value  outval;
  a_sql_int32 arg1, arg2, result;

  //  Get first argument
```

```
  (void) cntxt->get_value( arg_handle, 1, &arg );
  if (arg.data == NULL)
  {
      return;
  }
  arg1 = *((a_sql_int32 *)arg.data);

  //  Get second argument
  (void) cntxt->get_value( arg_handle, 2, &arg );
  if (arg.data == NULL)
  {
      return;
  }
  arg2 = *((a_sql_int32 *)arg.data);

  //  Set the result value
  outval.type = DT_INT;
  outval.piece_len = sizeof(a_sql_int32);
  result = arg1 + arg2;
  outval.data = &result;
  cntxt->set_value( arg_handle, &outval, 0 );
}


static a_v3_extfn_scalar my_plus_descriptor = {
    0,
    0,
    &my_plus_evaluate,
    0,            // Reserved - initialize to NULL
    0,            // Reserved - initialize to NULL
    0,            // Reserved - initialize to NULL
    0,            // Reserved - initialize to NULL
    0,            // Reserved - initialize to NULL
        NULL                    // _for_server_internal_use
};


a_v3_extfn_scalar *my_plus()
{
  return &my_plus_descriptor;
}

#if defined __cplusplus
}
#endif
```

UDF 示例：my_plus _counter 定义

此示例检查参数值指针数据以查看输入参数值是否为 NULL。它还有一个初始化函数和一个关闭函数，两个函数都允许进行多次调用。

my_plus _counter 定义

```
#include "extfnapiv3.h"
#include <stdlib.h>
```

```
//  A simple non-deterministic scalar UDF that adds
//  an internal integer usage counter to its integer
//  argument and then returns the resulting integer.
//
//  Here, the start function creates a little structure for
//  the counter, and then the finish function deallocates it.
//
//  Corresponding SQL declaration:
//
//          CREATE FUNCTION plus_counter(IN arg1 INT)
//                      RETURNS INT
//                      NOT DETERMINISTIC
//                      RESPECT NULL VALUES
//                      EXTERNAL NAME 'my_plus_counter@libudfex'


typedef struct my_counter {
  a_sql_int32 _counter;
} my_counter;

#if defined __cplusplus
extern "C" {
#endif

static void my_plus_counter_start(a_v3_extfn_scalar_context *cntxt)
{
  my_counter *cptr = (my_counter *)cntxt->_user_data;
  //  If we have not already allocated the
  //  counter structure, then do so now
  if (!cptr) {
    cptr = (my_counter *)malloc(sizeof(my_counter));
    cntxt->_user_data = cptr;
  }
  cptr->_counter = 0;
}


static void my_plus_counter_finish(a_v3_extfn_scalar_context *cntxt)
{
  //  If we still have an allocated the
  //  counter structure, then free it now
  if (cntxt->_user_data) {
    free(cntxt->_user_data);
    cntxt->_user_data = 0;
  }
}


static void my_plus_counter_evaluate(a_v3_extfn_scalar_context
*cntxt,
                                     void *arg_handle)
{
  an_extfn_value  arg;
  an_extfn_value  outval;
  a_sql_int32 arg1, result;
```

```
  //  Increment the usage counter
  my_counter *cptr = (my_counter *)cntxt->_user_data;
  cptr->_counter += 1;

  //  Get the one argument
  (void) cntxt->get_value( arg_handle, 1, &arg );
  if (!arg.data) {
    // argument value was NULL;
    arg1 = 0;
  } else {
    arg1 = *((a_sql_int32 *)arg.data);
  }

  outval.type = DT_INT;
  outval.piece_len = sizeof(a_sql_int32);
  result = arg1 + cptr->_counter;
  outval.data = &result;
  cntxt->set_value( arg_handle, &outval, 0 );
}


static a_v3_extfn_scalar my_plus_counter_descriptor =
    {   &my_plus_counter_start,
        &my_plus_counter_finish,
        &my_plus_counter_evaluate,
        NULL,                   // Reserved - initialize to NULL
        NULL,                   // Reserved - initialize to NULL
        NULL,                   // Reserved - initialize to NULL
        NULL,                   // Reserved - initialize to NULL
        NULL,                   // Reserved - initialize to NULL
        NULL,                   // _for_server_internal_use
};


a_v3_extfn_scalar *my_plus_counter()
{
  return &my_plus_counter_descriptor;
}

#if defined __cplusplus
}
#endif
```

用户定义的集合函数

Sybase IQ 支持用户定义的集合函数 (UDAF)。SUM 函数就是一个内置集合函数。简单集合函数会根据一组参数值生成一个结果值。您可以编写可在任何可以使用 SUM 集合的位置使用的 UDAF。

注意： 此处引用的集合 UDF 示例随 IQ 服务器一起安装，这些示例可作为 .cxx 文件存在于 `$IQDIR15/samples/udf` 下。 您还可以在 `$IQDIR15/lib64/libudfex` 动态可链接库中找到这些示例。

声明 UDAF

集合 UDAF 比标量 UDF 功能更强大，其创建也更复杂。

编写并编译了 UDF 代码后，创建一个从相应库文件中调用 UDF 并将输入数据发送到 UDF 的 SQL 函数。

注意： 您还可以“在 Sybase Central 中创建用户定义的函数声明” （第 12 页）。

实现 UDAF 时，您必须确定：

- 它是否将仅在整个数据集或分区上用作联机分析处理 (OLAP) 样式集合，例如 RANK。
- 它是否将用作简单集合或 OLAP 样式集合，例如 SUM。
- 它是否仅用作整个组上的简单集合。

UDAF 的声明和定义将反映以上使用决定。

创建 IQ 用户定义集合函数的语法为：

```
aggregate-udf-declaration:
    CREATE AGGREGATE FUNCTION [ owner.]function-name
          ( [ parameter , ... ] )
    RETURNS data-type
        [ aggregate-routine-characteristics ... ]
    EXTERNAL NAME library-and-entry-point-name-string

parameter:
    param-name data-type [ DEFAULT value ]

aggregate-routine-characteristics:
      DUPLICATE { SENSITIVE | INSENSITIVE }
        -- is the server allowed to eliminate DISTINCT
    | SQL SECURITY {INVOKER | DEFINER}
    | OVER restrict
    | ORDER order-restrict
        -- Must the window-spec contain an ORDER BY?
    | WINDOW FRAME
```

```
        { { ALLOWED | REQUIRED }
            [ window-frame-constraints ... ]
          | NOT ALLOWED }
    | ON EMPTY INPUT RETURNS { NULL | VALUE }
  -- Call or skip function on NULL inputs

window-frame-constraints:
      VALUES { [ NOT ] ALLOWED }
    | CURRENT ROW { REQUIRED | ALLOWED }
    | [ UNBOUNDED ] { PRECEDING | FOLLOWING } restrict

restrict:   { [ NOT ] ALLOWED } | REQUIRED

order-restrict:
{ NOT ALLOWED | SENSITIVE | INSENSITIVE | REQUIRED
```

返回数据类型、参数、数据类型和缺省值的处理都与标量 UDF 定义中的处理相同。

如果 UDAF 可用作简单集合，则它可能可以使用 DISTINCT 限定符。UDAF 声明中的 DUPLICATE 子句确定：

- 是否因为结果对重复条目敏感而考虑在调用 UDAF 之前删除重复值（例如对内置的“COUNT(DISTINCT T.A)”）或，
- 结果是否对重复条目不敏感（例如对于“MAX(DISTINCT T.A)”）。

DUPLICATE INSENSITIVE 选项允许优化程序考虑删除重复条目而不影响结果，因此优化程序可以选择如何执行查询。 UDAF 的编写必须考虑到重复条目。 如果需要删除重复条目，则服务器会在启动 _next_value_extfn 调用集合之前执行删除操作。

其余大部分子句均不属于标量 UDF 语法，利用这些子句可指定该函数的用法。缺省情况下，假设 UDAF 既可用作简单集合，也可用作采用任意类型窗口构架的 OLAP 样式集合。

对于仅用作简单集合函数的 UDAF，声明时请使用：

```
OVER NOT ALLOWED
```

任何随后将该集合用作 OLAP 样式集合的尝试都将生成错误。

对于允许或要求使用 OVER 子句的 UDAF，UDF 定义者可在使用 ORDER BY 子句的情况下在 OVER 子句来指定限制条件，即指定“ORDER”并在其后指定限制类型。窗口排序限制类型：

- **REQUIRED** — 必须指定 ORDER BY，不可将其删除。
- **SENSITIVE** — 指定或不指定 ORDER BY 均可，但一旦指定则不可将其删除。
- **INSENSITIVE** — 指定或不指定 ORDER BY 均可，但服务器可删除排序以提高效率。
- **NOT ALLOWED** — 不可指定 ORDER BY。

声明 UDAF 仅可在已排序的整个集合或分区上用作 OLAP 样式集合，例如内置的 RANK，请使用：

```
OVER REQUIRED
ORDER REQUIRED
WINDOW FRAME NOT ALLOWED
```

声明 UDAF 仅可用作使用缺省窗口构架 UNBOUNDED PRECEDING 到 CURRENT ROW 的 OLAP 样式集合，请使用：

```
OVER REQUIRED
ORDER REQUIRED
WINDOW FRAME ALLOWED
    RANGE NOT ALLOWED
    UNBOUNDED PRECEDING REQUIRED
    CURRENT ROW REQUIRED
    FOLLOWING NOT ALLOWED
```

所有各种选项和限制设置的缺省值如下：

```
DUPLICATE SENSITIVE
SQL SECURITY DEFINER
OVER ALLOWED
ORDER SENSITIVE
WINDOW FRAME ALLOWED
CURRENT ROW ALLOWED
PRECEDING ALLOWED
UNBOUNDED PRECEDING ALLOWED
FOLLOWING ALLOWED
UNBOUNDED FOLLOWING ALLOWED
```

SQL 安全性

定义是作为 INVOKER（调用函数的用户）还是作为 DEFINER（拥有函数的用户）执行函数。缺省为 DEFINER。

如果指定了 **SQL SECURITY INVOKER**，将使用更多内存，这是因为调用过程的每个用户都需要注释。此外，如果指定了 **SQL SECURITY INVOKER**，将同时对用户名和 INVOKER 执行名称解析。使用所有对象名（表、过程等）的相应所有者限定这些名称。

外部名称

使用 **EXTERNAL NAME** 子句的函数是包含对外部库函数调用的包装。使用 **EXTERNAL NAME** 的函数在 **RETURNS** 子句后可以不使用其它子句。库名可包含文件扩展名，在 Windows 中通常为 `.dll`，在 UNIX 中通常为 `.so`。在没有扩展名的情况下，该软件附加平台特定的缺省库文件扩展名。

临时函数不支持 **EXTERNAL NAME** 子句。请参见《SQL Anywhere Server — 编程》中的“从过程调用外部库”。

可以通过包括 UDF 库位置的库装载路径启动 IQ 服务器。在 Unix 变体中，通过修改 start_iq 启动脚本中的 LD_LIBRARY_PATH 可以实现此操作。虽然 LD_LIBRARY_PATH 对所有 UNIX 变体都通用，但在 HP 上首选使用 SHLIB_PATH，在 AIX 上首选使用 LIB_PATH。

在 Unix 平台上，外部名称规范可以包含完全限定名，这种情况下不使用 LD_LIBRARY_PATH。在 Windows 平台上，无法使用完全限定名，且库搜索路径由 PATH 环境变量定义。

注意： 可更新游标中不支持标量用户定义函数和用户定义集合函数。

UDAF 示例：my_sum 声明

“my_sum”示例与内置 SUM 相似，只是它仅执行整数运算。

my_sum 声明

既然 my_sum 像 SUM 一样可以在任意上下文内使用，它的声明相对比较简单：

```
CREATE AGGREGATE FUNCTION my_sum(IN arg1 INT)
    RETURNS BIGINT
    ON EMPTY INPUT RETURNS NULL
    EXTERNAL NAME 'describe_my_integer_sum@my_shared_lib'
```

各种使用限制均缺省为 ALLOWED，指定该函数可在 SQL 语句内允许使用集合函数的任意位置上使用。

没有任何使用限制时，my_sum 可用作整个行集上的简单集合，如下所示：

```
SELECT MIN(t.x), COUNT (*), my_sum(t.y)
FROM t
```

没有任何使用限制时，my_sum 也可用作针对 GROUP BY 子句所指定的每个组进行计算的简单集合：

```
SELECT t.x, COUNT(*), my_sum(t.y)
FROM t
GROUP BY t.x
```

由于没有使用限制，my_sum 可用作带有 OVER 子句的 OLAP 样式集合，如下累计求和示例所示：

```
SELECT t.x,
    my_sum(t.x)
        OVER (ORDER BY t.x ROWS BETWEEN UNBOUNDED PRECEDING AND
CURRENT ROW)
        AS cumulative_x,
    COUNT(*)
FROM t
GROUP BY t.x
ORDER BY t.x
```

UDAF 示例：my_bit_xor 声明

“my_bit_xor”示例类似于 SQL Anywhere (SA) 的内置 BIT_XOR，只是它仅仅执行无符号整数运算。

my_bit_xor 声明

形成的声明为：

```
CREATE AGGREGATE FUNCTION my_bit_xor(IN arg1 UNSIGNED    INT)
 RETURNS UNSIGNED INT
    ON EMPTY INPUT RETURNS NULL
    EXTERNAL NAME 'describe_my_bit_xor@my_shared_lib'
```

像 my_sum 示例一样，my_bit_xor 没有关联任何使用限制，因此可用作简单集合或具有任意类型窗口的 OLAP 样式集合。

UDAF 示例：my_bit_or 声明

“my_bit_or”示例类似于 SA 内置的 BIT_OR，只是它仅仅执行无符号整数运算，只能用作简单集合。

my_bit_or 声明

形成的声明类似如下：

```
CREATE AGGREGATE FUNCTION my_bit_or(IN arg1 UNSIGNED INT)
     RETURNS UNSIGNED INT
     ON EMPTY INPUT RETURNS NULL
     OVER NOT ALLOWED
     EXTERNAL NAME 'describe_my_bit_or@ my_shared_lib'
```

与 my_bit_xor 示例不同，声明中的 OVER NOT ALLOWED 短语限制该函数用于简单集合。 由于该使用限制，my_bit_or 只能用作整个行集上的简单集合，或用作针对 GROUP BY 子句所指定的每个组进行计算的简单集合，如下示例所示：

```
SELECT t.x, COUNT(*), my_bit_or(t.y)
FROM t
GROUP BY t.x
```

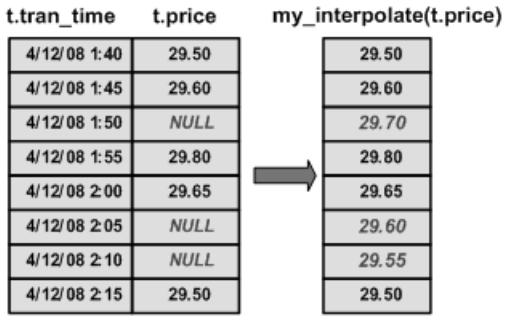
UDAF 示例：my_interpolate 声明

“my_interpolate”示例是 OLAP 样式的 UDAF，它跨任意一组与两个方向上最近非空值相邻的空值执行线性内插操作，尝试在序列中缺少值的位置中填入值（其中由空值表示缺少值）。

my_interpolate 声明

如果给定行上的输入不是 NULL，则该行的结果与输入值相同。

图 1： my_interpolate 结果



为合理控制运算成本，运行 my_interpolate 时必须使用固定宽度行式窗口，但用户可以根据他/她期望看到的相邻 NULL 值的最大数目设置窗口宽度。 该函数输入一组双精度浮点值，通过运算生成一组双精度浮点值。

形成的 UDAF 声明类似如下：

```
CREATE AGGREGATE FUNCTION my_interpolate (IN arg1 DOUBLE)
RETURNS DOUBLE
    OVER REQUIRED
    WINDOW FRAME REQUIRED
        RANGE NOT ALLOWED
        PRECEDING REQUIRED
        UNBOUNDED PRECEDING NOT ALLOWED
        FOLLOWING REQUIRED
        UNBOUNDED FOLLOWING NOT ALLOWED
    EXTERNAL NAME 'describe_my_interpolate@my_shared_lib'
```

OVER REQUIRED 表示该函数不能用作简单集合（即使使用 ON EMPTY INPUT 也没有意义）。

WINDOW FRAME 详细信息指定使用此函数时必须使用固定宽度行式窗口，该窗口可从当前行向前和向后两个方向延伸。 由于这些使用限制，my_interpolate 用作带有 OVER 子句的 OLAP 样式集合，类似于：

```
SELECT t.x,
    my_interpolate(t.x)
    OVER (ORDER BY t.x ROWS BETWEEN 5 PRECEDING AND 5 FOLLOWING)
        AS x_with_gaps_filled,
    COUNT(*)
FROM t
GROUP BY t.x
ORDER BY t.x
```

在 my_interpolate 的 OVER 子句中，前面的行和后面的行的精确数值可以变化，您可以选择使用 PARTITION BY 子句；否则，这些行一定会与在声明中给定使用限制的以上示例类似。

定义集合 UDF

用于定义用户定义集合函数的 C/C++ 代码包含 8 个必需项。

- **extfnapiv3.h** – UDF 接口定义头文件。
- **_start_extfn** – 一个初始化函数，每次使用 SQL 时调用一次。 所有初始化函数均使用一个参数：一个指向集合 UDF 上下文结构的指针，每次使用 UDAF 时该指针都是唯一的。 所传递的上下文结构与向针对该用法提供的所有函数传递的上下文结构相同。
- **_finish_extfn** – 一个关闭函数，每次使用 SQL 时调用一次。 所有关闭函数均使用一个参数：一个指向 UDAF 上下文结构的指针，每次使用 UDAF 时该指针都是唯一的。
- **_reset_extfn** – 一个重置函数，启动每个新组、新分区时调用一次，如有必要，启动每个窗口动作时也将调用该函数。 所有重置函数均使用一个参数：一个指向 UDAF 上下文结构的指针，每次使用 UDAF 时该指针都是唯一的。
- **_next_value_extfn** – 针对每组新输入参数调用的函数。 _next_value_extfn 使用两个参数：
  - 指向 UDAF 上下文的指针，以及
  - 一个 args_handle。

  与在标量 UDF 中一样，arg_handle 可与所提供的回调函数指针一起使用以访问实际参数值。
- **_evaluate_extfn** – 与标量 UDF 求值函数类似的一个求值函数。 所有求值函数都有两个参数：
  - 指向 UDAF 上下文结构的指针，以及
  - 一个 args_handle。
- **a_v3_extfn_aggregate** – 集合 UDF 描述符结构的一个实例，包含指向该 UDF 所有已提供函数的指针。
- **Descriptor function** – 一个描述符函数，返回指向该集合 UDF 描述符结构的一个指针。

除必需项外，还有若干可选项，支持针对特定使用情况实现更加优化的访问：

- **_drop_value_extfn** – 针对超出移动窗口构架的每个输入参数值集调用的可选函数指针。 该函数不应设置集合结果。 使用 get_value callback 函数访问输入参数值，如有必要，可重复调用 get_piece 回调函数。
  在下列情况下，可将函数指针设置为空指针：
  - 集合不能与窗口构架一起使用，
  - 集合在某种程度上不可逆，或
  - 用户对最佳性能没有兴趣。

  如果未提供 _drop_value_extfn，且用户已经指定移动窗口，则可在每次窗口构架移动时调用重置函数，并通过调用 next_value 函数包含该窗口内的各行，最后调用求值函数。

如果已提供 _drop_value_extfn，则在每次窗口构架移动时，针对超出窗口构架的各行调用该删除值函数，然后针对刚刚添加到窗口构架中的各行调用 next_value 函数，最后调用求值函数以生成集合结果。

- **_evaluate_cumulative_extfn** – 针对每组新输入参数值调用的可选函数指针。如果已提供该函数，且在跨越从 UNBOUNDED PRECEDING 到 CURRENT ROW 的基于行的窗口构架内使用，则可调用此函数，而无需调用“下一个值”函数以及紧邻其后调用的求值函数。

  _evaluate_cumulative_extfn 必须通过 set_value 回调设置集合结果。 可通过常用的 get_value 回调函数访问其输入参数值集合。 在下列情况下，该函数指针应设置为空指针：
  - 绝不会以此方式使用该集合，或
  - 用户不担心最佳性能。
- **_next_subaggregate_extfn** – 可与 _evaluate_superaggregate_extfn 一起使用的可选回调函数指针，支持通过并行运行来优化该集合的某些用法。

  某些集合用作简单集合（换言之即带有 OVER 子句的非 OLAP 样式集合）时，可以通过首先生成一组中间集合结果来分区，其中每个中间结果均为来自一个不连接的输入行子集的计算结果。

  此类可分区集合的示例包括：
  - SUM，其中，可通过针对每个不连接的输入行子集执行 SUM，然后在 sub-SUM 上执行 SUM，由此计算得出最终 SUM；以及
  - COUNT(*)，其中，可通过针对每个不连接的输入行子集执行 COUNT，然后在每个分区的 COUNT 上执行 SUM，由此计算得出最终 COUNT。

  当集合满足上述条件时，服务器可以选择并行执行该集合的计算。对于集合 UDF，仅当已提供 _next_subaggregate_extfn 函数指针和 _evaluate_superaggregate_extfn 指针时，才能应用该并行优化。

  _reset_extfn 函数不设置集合最终结果，而且按照定义，它将使用与集合 UDF 的定义返回值完全相同的数据类型的唯一输入参数值。

  通过正常的 get_value 回调函数访问子集合输入值。 子集合与超级集合之间不可进行直接通信；所有此类通信将由服务器处理。 子集合和超级集合不共用上下文结构。 而是将各个子集合完全视为非分区集合。 独立超级集合将看到类似如下的调用模式：

```
_start_extfn
_reset_extfn
_next_subaggregate_extfn (repeated 0 to N times)
_evaluate_superaggregate_extfn
_finish_extfn
```

  或类似如下：

```
_start_extfn
_reset_extfn
_next_subaggregate_extfn (repeated 0 to N times)
_evaluate_superaggregate_extfn
_reset_extfn
_next_subaggregate_extfn (repeated 0 to N times)
_evaluate_superaggregate_extfn
_reset_extfn
```

```
_next_subaggregate_extfn (repeated 0 to N times)
_evaluate_superaggregate_extfn
_finish_extfn
```

如果 _evaluate_superaggregate_extfn 或 _next_subaggregate_extfn 均未提供，则 UDAF 将受限制，不允许其在包含 GROUP BY CUBE 或 GROUP BY ROLLUP 的查询块内用作简单集合。

- **_evaluate_superaggregate_extfn** – 可与 _next_subaggregate_extfn 一起使用的可选回调函数指针，支持通过并行来优化用作简单集合时的某些用法。 调用 _evaluate_superaggregate_extfn 返回分区集合的结果。 通过使用 a_v3_extfn_aggregate_context 结构中的正常 set_value 回调函数将结果值发送至服务器。

集合 UDF 描述符结构

集合 UDF 描述符结构包含下列项：

- **typedef struct a_v3_extfn_aggregate** – 库提供的集合 UDF 函数的元数据描述符。
- **_start_extfn** – 初始化函数的必需指针，其唯一参数是指向 a_v3_extfn_aggregate_context 的指针。 通常用于分配某些结构并将其地址存储在 a_v3_extfn_aggregate_context 内的 _user_data 字段中。 每 a_v3_extfn_aggregate_context 只能调用一次 _start_extfn。

  ```
  void (*_start_extfn)(a_v3_extfn_aggregate_context *);
  ```
- **_finish_extfn** – 关闭函数的必需指针，其唯一参数是指向 a_v3_extfn_aggregate_context 的指针。 通常用于释放地址存储在 a_v3_extfn_aggregate_context 中的 _user_data 字段内的某些结构。 每 a_v3_extfn_aggregate_context 只能调用一次 _finish_extfn。

  ```
  void (*_finish_extfn)(a_v3_extfn_aggregate_context *);
  ```
- **_reset_extfn** – “启动新组”函数的必需指针，其唯一参数是指向 a_v3_extfn_aggregate_context 的指针。 通常用于重置其地址存放在 a_v3_extfn_aggregate_context 中的 _user_data 字段内的结构中的某些值。 _reset_extfn 可重复调用。

  ```
  void (*_reset_extfn)(a_v3_extfn_aggregate_context *);
  ```
- **_next_value_extfn** – 针对每组新输入参数值调用的必需函数指针。 该函数不设置集合结果。 可通过 get_value 回调函数访问输入参数值，如有必要，可重复调用 get_piece 回调函数，但仅当 piece_len 小于 total_len 时才需执行该操作。

  ```
  void (*_next_value_extfn)(a_v3_extfn_aggregate_context *cntxt,
  void *args_handle);
  ```
- **_evaluate_extfn** – 必需函数指针，调用它可返回生成的集合结果值。 _evaluate_extfn 将通过 set_value 回调函数发送至服务器。

  ```
  void (*_evaluate_extfn)(a_v3_extfn_aggregate_context *cntxt, void
  *args_handle);
  ```
- **_drop_value_extfn** – 针对超出移动窗口构架的每个输入参数值集调用的可选函数指针。 不要使用此函数设置集合结果。 可通过 get_value 回调函数访问输入参

数值，如有必要，可重复调用 get_piece 回调函数，但仅当 piece_len 小于 total_len 时才需访问输入参数值。 在下列情况下，请将 _drop_value_extfn 设置为空指针：

- 集合不能与窗口构架一起使用。
- 集合在某种程度上不可逆。
- 用户对最佳性能没有兴趣。

如果未提供该函数，且用户已经指定移动窗口，则可在每次窗口构架移动时调用重置函数，并通过调用 next_value 函数包含该窗口内的现有各行。最后调用求值函数。

但是，如果已提供该函数，则每次窗口构架移动时，均将针对超出窗口架构的各行调用 drop_value 函数，然后针对刚刚添加到窗口构架中的各行调用 next_value 函数。最后，调用求值函数以生成集合结果。

```
void (*_drop_value_extfn)(a_v3_extfn_aggregate_context    *cntxt,
void *args_handle);
```

- **_evaluate_cumulative_extfn** – 针对每组新输入参数值调用的可选函数指针。如果已提供该函数，且在跨越从 UNBOUNDED PRECEDING 到 CURRENT ROW 的行式窗口构架内使用，则可调用此函数，而无需调用 next_value 函数以及紧邻其后调用的求值函数。 _evaluate_cumulative_extfn 必须通过 set_value 回调设置集合结果。 可通过 get_value 回调函数访问输入参数值，如有必要，可重复调用 get_piece 回调函数，但仅当 piece_len 小于 total_len 时才需执行该操作。

```
void (*_evaluate_cumulative_extfn)(a_v3_extfn_aggregate_context
*cntxt, void *args_handle);
```

- **_next_subaggregate_extfn** – 可选回调函数指针，通过 _evaluate_superaggregate_extfn 函数（在某些用法中还使用 _drop_subaggregate_extfn 函数），支持通过并行和部分结果集合来优化集合的某些用法。

  某些集合用作简单集合（换言之即带有 OVER 子句的非 OLAP 样式集合）时，可以通过首先生成一组中间集合结果来分区，其中每个中间结果均为来自一个不连接的输入行子集的计算结果。此类可分区集合的示例包括：

  - SUM，其中，可通过针对每个不连接的输入行子集执行 SUM，然后在 sub-SUM 上执行 SUM，由此计算得出最终 SUM；以及
  - COUNT(*)，其中，可通过针对每个不连接的输入行子集执行 COUNT，然后在每个分区的 COUNT 上执行 SUM，由此计算得出最终 COUNT。

  当集合满足上述条件时，服务器可以选择并行执行该集合的计算。对于集合 UDF，仅当已提供 _next_subaggregate_extfn 回调和 _evaluate_superaggregate_extfn 回调时，才能应用该优化。此用法模式不需要使用 _drop_subaggregate_extfn。

  同理，如果集合可与基于 RANGE 的 OVER 子句一起使用，且如果 UDAF 实现已完全提供 _next_subaggregate_extfn、_drop_subaggregate_extfn 和 _evaluate_superaggregate_extfn 函数，则可应用优化。

  _next_subaggregate_extfn 不设置集合最终结果，而且按照定义，它将使用与集合 UDF 的返回值完全相同的数据类型的唯一输入参数值。可通过 get_value 回调函数访问子集输入值，如有必要，可重复调用 get_piece 回调函数，但仅当 piece_len 小于 total_len 时才需执行该操作。

子集合与超级集合之间不可进行直接通信；所有此类通信将由服务器处理。 子集合和超级集合不共用上下文结构。 而将各个子集合完全视为非分区集合。 独立超级集合将看到类似如下的调用模式：

```
       _start_extfn
 _reset_extfn
       _next_subaggregate_extfn (repeated 0 to N times)
       _evaluate_superaggregate_extfn
       _finish_extfn
```

```
void (*_next_subaggregate_extfn)(a_v3_extfn_aggregate_context
*cntxt, void *args_handle);
```

- **_drop_subaggregate_extfn** – 可选回调函数指针，与 _next_subaggregate_extfn 和 _evaluate_superaggregate_extfn 一起使用，支持通过部分集合来优化使用基于 RANGE 的 OVER 子句时的某些用法。 每当共享通用排序键值的一组行全部超出移动窗口时即调用 _drop_subaggregate_extfn。 仅当所有三个函数均由 UDF 提供时才能应用此优化。

  ```
  void (*_drop_subaggregate_extfn)(a_v3_extfn_aggregate_context
  *cntxt,    void *args_handle);
  ```

- **_evaluate_superaggregate_extfn** – 可选回调函数指针，与 _next_subaggregate_extfn（有时也可与 _drop_subaggregate_extfn）一起使用时，支持通过并行运行来优化某些用法。

  将要返回分区集合结果时，可按如上所述调用 _evaluate_superaggregate_extfn。通过使用 a_v3_extfn_aggregate_context 结构中的 set_value 回调函数将结果值发送至服务器：

  ```
  void (*_evaluate_superaggregate_extfn)
  (a_v3_extfn_aggregate_context *cntxt, void *args_handle);
  ```

- **NULL fields** – 将以下字段初始化为 NULL：

  ```
  void *    reserved1_must_be_null;
  void *    reserved2_must_be_null;
  void *    reserved3_must_be_null;
  void *    reserved4_must_be_null;
  void *    reserved5_must_be_null;
  ```

- **Status indicator bit field** – 包含允许引擎优化用于处理集合的算法的指示符的位字段。

  ```
  a_sql_uint32  indicators;
  ```

- **_calculation_context_size** – 服务器为每个 UDF 计算上下文分配的字节数。 服务器可以在查询处理过程中分配多个计算上下文。

  a_v3_extfn_aggregate_context_user_calculation_context 中可使用当前活动组的上下文。

  ```
  short _calculation_context_size;
  ```

- **_calculation_context_alignment** – 指定用户计算上下文的对齐要求。 有效值包括 1、2、4 或 8。

  ```
  short _calculation_context_alignment;
  ```

- **External memory requirments** – 下列字段允许优化程序考虑外部分配的内存的开销。 有了这些值，优化程序就能考虑可以执行多个并发计算的程度。 这些计数

器应当基于典型的行或组来估计，并且不应为最大值。 如果没有 UDF 分配的内存，则将这些字段设为零。

- external_bytes_per_group – 分配到每个集合开头的组的内存量。 通常为所有在 reset() 调用期间分配的内存。
- external_bytes_per_row — UDF 为组中每一行分配的内存量。通常为 next_value() 期间所分配的内存量。

```
double            external_bytes_per_group;
double            external_bytes_per_row;
```

- **Reserved fields for future use** – 初始化下列字段：

```
a_sql_uint64      reserved6_must_be_null;
a_sql_uint64      reserved7_must_be_null;
a_sql_uint64      reserved8_must_be_null;
a_sql_uint64      reserved9_must_be_null;
a_sql_uint64      reserved10_must_be_null;
```

- **Closing syntax** — 用下列语法完成描述符：

```
//---------- For Server Internal Use Only ----------
void  * _for_server_internal_use;
} a_extfn_aggregate;
```

计算上下文

_user_calculation_context 字段允许服务器对多组数据同时执行计算。

UDAF 必须在处理行时为计算保留中间计数器。管理这些计数器的简单模型是，在开始 API 函数中分配内存，将它的指针存储在集合上下文的 _user_data 字段中，然后在集合的完成 API 中释放内存。替代的方法为，根据 _user_calculation_context 字段来允许服务器对多组数据同时执行计算。

_user_calculation_context 字段是服务器分配的内存指针，由服务器为每个并发处理的组创建。 服务器确保 _user_calculation_context 始终为当前正在处理的那一组行指向正确的计算上下文。 在 UDF API 调用之间，服务器可能会根据数据分配新的 _user_calculation_context 值。 服务器可能会在处理查询期间将计算上下文区域保存并恢复到磁盘。

UDF 将所有中间计算值存储在此字段中。以下说明一个典型的用法：

```
struct my_average_context
{
        int    sum;
        int    count;
};

reset(a_v3_aggregate_context *context)
{
    mycontext  = (my_average_context *) context-
>_user_calculation_context;
    mycontext->count = 0;
    mycontext->sum = 0;
}

    next_value(a_v3_aggregate_context *context, void *args_handle)
```

```
{
    mycontext  = (my_average_context *) context-
>_user_calculation_context;
    mycontext->count++;
    ..
}
```

在此模型中，_user_data 字段仍然可用，但其中不能存储任何有关中间结果计算的值。_user_calculation_context 在开始和完成入口点处均为 NULL。

要使用 _user_calculation_context 启用并发处理，UDF 必须为其计算上下文指定大小和对齐要求，定义一个容纳其值的结构，并将 a_v3_extfn_aggregate 和 _calculation_context_size 设置为该结构的 sizeof()。

UDF 还必须通过 _calculation_context_alignment 指定 _user_calculation_context 的数据对齐要求。 如果 user_calculation_context 内存只包含一个字符字节数组，则无需特别的对齐，并且可以指定 1 字节对齐。同样，双精度浮点数值可能需要 8 字节对齐。对齐要求因平台和数据类型的不同而异。 指定比所需的大的对齐始终都是可行的；但若使用最小的对齐，内存使用效率会更高。

UDAF 上下文结构

集合 UDF 上下文结构 a_v3_extfn_aggregate_context 具有和标量 UDF 上下文结构完全相同的回调函数指针集。

此外，与标量 UDF 上下文类似，它具有读/写 _user_data 指针，还具有一组描述当前使用情况和位置的只读数据字段。 一个语句中的每个 UDF 唯一实例都具有一个集合 UDF 上下文实例，它在调用时传递到集合 UDF 描述符结构中指定的每一个函数。 集合上下文结构定义为：

- **typedef struct a_v3_extfn_aggregate_context** — 为在查询中引用的外部函数的每一个实例创建一个上下文。如果用在查询内并行的子树中，则并行子树会具有单独的上下文。
- **Callbacks available via the context** — 回调例程的常见参数包括：
  - **arg_handle** — 由服务器提供的函数实例和参数的句柄。
  - **arg_num** — 参数数目。返回值为 0 到 N。
  - **data** — 参数数据的指针。

  上下文必须在 get_piece 之前调用 get_value，但仅在 piece_len 小于 total_len 时才需要调用 get_piece。

```
short (SQL_CALLBACK *get_value)(
  void *          arg_handle,
        a_sql_uint32    arg_num,
        an_extfn_value *value
        );
short (SQL_CALLBACK *get_piece)(
        void *          arg_handle,
        a_sql_uint32    arg_num,
        an_extfn_value *value,
```

```
        a_sql_uint32    offset
        );
```

- **Determining whether an argument is a constant** – UDF 可以询问给定的参数是否为常量。这非常有用。例如，允许工作在第一次调用 _next_value 函数时执行一次，而不是在每次调用 _next_value 函数时执行。

```
short (SQL_CALLBACK *get_value_is_constant)(
  void *          arg_handle,
        a_sql_uint32    arg_num,
        a_sql_uint32 *  value_is_constant
        );
```

- **Returning a null value** — 要返回空值，可将 an_extfn_value 中的“data”设为 NULL。调用 set_value 时会忽略 total_len 字段。如果 append 为 FALSE，则提供的数据会变为参数的值；否则，该数据会附加到参数的当前值中。预期的情形为，在使用 append=TRUE 为一个参数调用 set_value 前，先使用 append=FALSE 为该参数进行调用。对于固定长度的数据类型（换句话说，所有数字数据类型），会忽略 append 字段。

```
short (SQL_CALLBACK *set_value)(
  void *          arg_handle,
        an_extfn_value *value,
        short              append
        );
```

- **Determining whether the statement was interrupted** – 如果 UDF 入口点执行工作的时间较长（许多秒），则可能的话，它应当每秒或每两秒调用一次 get_is_cancelled 回调，以查看用户是否中断了当前的语句。如果语句已被中断，则返回非零值，并且 UDF 入口点应立即执行。最后，调用 _finish_extfn 函数执行任何必要的清理，但随后不再调用任何其它的 UDF 入口点。

```
 a_sql_uint32 (SQL_CALLBACK *get_is_cancelled)
(a_v3_extfn_aggregate_context * cntxt);
```

- **Sending error messages** – 如果 UDF 入口点遇到某个错误，该错误可导致向用户发回错误消息并且关闭当前语句，则应调用 set_error 回调例程。set_error 导致当前语句回退；用户会看到 `Error from external UDF: <error_desc_string>`，并且 SQLCODE 是 <error_number> 的负值形式。调用 set_error 之后，UDF 入口点会立即执行返回。最后，调用 _finish_extfn 函数执行任何必要的清理，但随后不再调用任何其它的 UDF 入口点。

```
void (SQL_CALLBACK *set_error)(
  a_v3_extfn_aggregate_context * cntxt,
        a_sql_uint32    error_number,
        // use error_number values >17000 & <100000
        const char *    error_desc_string
        );
```

- **Writing messages to the message log** — 长度超过 255 字节的消息会被截断。

```
void (SQL_CALLBACK *log_message)(
  const char *msg,
        short msg_length
  );
```

- **Converting one data type to another** – 对于输入：

- **an_extfn_value.data** – 输入数据指针。
- **an_extfn_value.total_len** – 输入数据的长度。
- **an_extfn_value.type** — 输入的 DT_ 数据类型。

对于输出：

- **an_extfn_value.data** — UDF 提供的输出数据指针。
- **an_extfn_value.piece_len** – 输出数据的最大长度。
- **an_extfn_value.total_len** – 服务器设置的转换输出长度。
- **an_extfn_value.type** — 所需输出的 DT_ 数据类型。

```
short (SQL_CALLBACK *convert_value)(
  an_extfn_value   *input,
        an_extfn_value   *output
        );
```

- **Fields reserved for future use** — 这些留待将来使用：

```
void  *   reserved1;
void  *   reserved2;
void  *   reserved3;
void  *   reserved4;
void  *   reserved5;
```

- **Data available from the context** — 此数据指针可通过任何用法使用外部例程所需的任何上下文数据填充。UDF 会分配和释放此内存。每个语句都有处于活动状态的单个 _user_data 实例。不要将此内存用于中间结果值。

```
void  * _user_data;
```

- **Currently active calculation context** — UDF 应使用此内存位置存储计算集合的中间值。此内存由服务器根据 a_v3_extfn_aggregate 中请求的大小进行分配。中间计算必须存储在此内存中，因为引擎可能会对一个以上的组执行并发计算。在每个 UDF 入口点之前，服务器会确保正确的上下文数据处于活动状态。

```
void  * _user_calculation_context;
```

- **Other available aggregate information** – 在包括 start_extfn 在内的所有外部函数入口点上均可用。 零表示未知或不适用的值。 每个分区或每组中估计的平均行数。
  - **a_sql_uint64 _max_rows_in_frame;** — 计算窗口构架中定义的最大行数。对于基于范围的窗口，这表示唯一值。零表示未知或不适用的值。
  - **a_sql_uint64 _estimated_rows_per_partition;** – 显示每个分区或每组中估计的平均行数。 0 表示未知或不适用的值。
  - **a_sql_uint32 _is_used_as_a_superaggregate;** – 标识此实例为普通集合还是超级集合。 如果实例为普通集合，则返回的结果为 0。
- **Determining window specifications** — 查询上存在窗口时的窗口规范：
  - **a_sql_uint32 _is_window_used;** — 确定语句是否为窗口化的。
  - **a_sql_uint32 _window_has_unbounded_preceding;** – 返回值为 0，表示窗口没有未绑定的之前行。
  - **a_sql_uint32 _window_contains_current_row;** – 返回值为 0，表示窗口不包含当前行。

  - **a_sql_uint32 _window_is_range_based;** – 如果返回码为 1，则窗口基于范围。 如果返回码为 0，则窗口基于行。
- **Available at reset_extfn() calls** – 返回当前分区中的实际行数；或者，若为非窗口集合，则返回 0。

```
a_sql_uint64 _num_rows_in_partition;
```

- **Available only at evaluate_extfn() calls for windowed aggregates** — 分区中当前求值的行号（从 1 开始）。这在未受限制窗口的求值阶段很有用。

```
a_sql_uint64 _result_row_from_start_of_partition;
```

- **Closing syntax** — 用下列语法完成上下文：

```
//---------- For Server Internal Use Only ----------
  void  * _for_server_internal_use;
} a_v3_extfn_aggregate_context;
```

UDAF 示例：my_sum 定义

“my_sum”示例仅对整数进行操作。

my_sum 定义

由于和 SUM 类似，my_sum 可在任何上下文中使用，因而提供了所有已优化的可选入口点。在本例中，也可与 _evaluate_superaggregate_extfn 函数一样使用普通的 _evaluate_extfn 函数。

```
#include "extfnapiv3.h"
#include <stdlib.h>
#include <assert.h>


//  Simple aggregate UDF that adds up a set of
//  integer arguments, and whenever asked returns
//  the resulting big integer total.  For int
//  arguments, the only difference between this
//  UDF and the SUM built-in aggregate is that this
//  UDF will return NULL if there are no input rows.
//
//  The start function creates a little structure for
//  the running total, and the finish function then
//  deallocates it.
//
//  Since there are no aggregate usage restrictions
//  for this UDAF, the corresponding SQL declaration
//  will look like:
//
//              CREATE AGGREGATE FUNCTION my_sum(IN arg1 INT)
//                      RETURNS BIGINT
//                      ON EMPTY INPUT RETURNS NULL
//                      EXTERNAL NAME 'my_integer_sum@libudfex'


typedef struct my_total {
  a_sql_int64   _total;
  a_sql_uint64  _num_nonnulls_seen;
```

```
} my_total;


extern "C"
void my_integer_sum_start(a_v3_extfn_aggregate_context *cntxt)
{
}


extern "C"
void my_integer_sum_finish(a_v3_extfn_aggregate_context *cntxt)
{
}


extern "C"
void my_integer_sum_reset(a_v3_extfn_aggregate_context *cntxt)
{
  my_total *cptr = (my_total *)cntxt->_user_calculation_context;
  cptr->_total = 0;
  cptr->_num_nonnulls_seen = 0;
}


extern "C"
void my_integer_sum_next_value(a_v3_extfn_aggregate_context *cntxt,
                               void *arg_handle)
{
  an_extfn_value  arg;
  a_sql_int32 arg1;

  my_total *cptr = (my_total *)cntxt->_user_calculation_context;

  //  Get the one argument, and if non-NULL then add it to the total
  //
  if (cntxt->get_value( arg_handle, 1, &arg) && arg.data) {
    arg1 = *((a_sql_int32 *)arg.data);
    cptr->_total += arg1;
    cptr->_num_nonnulls_seen++;
  }
}


extern "C"
void my_integer_sum_drop_value(a_v3_extfn_aggregate_context *cntxt,
                               void *arg_handle)
{
  an_extfn_value  arg;
  a_sql_int32 arg1;
  my_total *cptr = (my_total *)cntxt->_user_calculation_context;

  //  Get the one argument, and if non-NULL then subtract it from the
total
  if (cntxt->get_value( arg_handle, 1, &arg) && arg.data) {
    arg1 = *((a_sql_int32 *)arg.data);
    cptr->_total -= arg1;
```

```
    cptr->_num_nonnulls_seen--;
  }
}


extern "C"
void my_integer_sum_evaluate(a_v3_extfn_aggregate_context *cntxt,
                             void *arg_handle)
{
  an_extfn_value  outval;
  my_total *cptr = (my_total *)cntxt->_user_calculation_context;

  //  Set the output result value.  If the inputs
  //  were all NULL, then set the result as NULL.
  //
  outval.type = DT_BIGINT;
  outval.piece_len = sizeof(a_sql_int64);
  if (cptr->_num_nonnulls_seen > 0) {
    outval.data = &cptr->_total;
  } else {
    outval.data = 0;
  }
  cntxt->set_value( arg_handle, &outval, 0 );
}


extern "C"
void my_integer_sum_cum_evaluate(
                        a_v3_extfn_aggregate_context *cntxt,
                        void *arg_handle)
{
  an_extfn_value  outval;
  an_extfn_value  arg;
  int arg1;
  my_total *cptr = (my_total *)cntxt->_user_calculation_context;

  //  Get the one argument, and if non-NULL then add it into the
total.
  //
  if (cntxt->get_value( arg_handle, 1, &arg) && arg.data) {
    arg1 = *((a_sql_int32 *)arg.data);
    cptr->_total += arg1;
    cptr->_num_nonnulls_seen++;
  }

  //  Then set the output result value.  If the inputs
  //  were all NULL, then set the result as NULL.
  //
  outval.type = DT_BIGINT;
  outval.piece_len = sizeof(a_sql_int64);
  if (cptr->_num_nonnulls_seen > 0) {
    outval.data = &cptr->_total;
  } else {
    outval.data = 0;
  }
  cntxt->set_value( arg_handle, &outval, 0 );
```

```
}


extern "C"
void my_integer_sum_next_subagg_value(
                a_v3_extfn_aggregate_context *cntxt,
                void *arg_handle)
{
  an_extfn_value  arg;
  a_sql_int64 arg1;

  my_total *cptr = (my_total *)cntxt->_user_calculation_context;

  //  Get the one argument, and if non-NULL then add it to the total
  //
  if (cntxt->get_value( arg_handle, 1, &arg) && arg.data) {
    arg1 = *((a_sql_int64 *)arg.data);
    cptr->_total += arg1;
    cptr->_num_nonnulls_seen++;
  }
}


extern "C"
void my_integer_sum_drop_subagg_value(
                a_v3_extfn_aggregate_context *cntxt,
                void *arg_handle)
{
  an_extfn_value  arg;
  a_sql_int64 arg1;

  my_total *cptr = (my_total *)cntxt->_user_calculation_context;

  //  Get the one argument, and if non-NULL then subtract it from the
total
  //
  if (cntxt->get_value( arg_handle, 1, &arg) && arg.data) {
    arg1 = *((a_sql_int64 *)arg.data);
    cptr->_total -= arg1;
    cptr->_num_nonnulls_seen--;
  }
}



a_v3_extfn_aggregate my_integer_sum_descriptor =
{
        &my_integer_sum_start,
        &my_integer_sum_finish,
        &my_integer_sum_reset,
        &my_integer_sum_next_value,
        &my_integer_sum_evaluate,
        &my_integer_sum_drop_value,
        &my_integer_sum_cum_evaluate,
        &my_integer_sum_next_subagg_value,
        &my_integer_sum_drop_subagg_value,
```

```
        &my_integer_sum_evaluate,
        NULL, // reserved1_must_be_null
        NULL, // reserved2_must_be_null
        NULL, // reserved3_must_be_null
        NULL, // reserved4_must_be_null
        NULL, // reserved5_must_be_null
        0, // indicators
        ( short )sizeof( my_total ), // context size
        8, // context alignment
        0.0, //external_bytes_per_group
        0.0, // external bytes per row
        0, // reserved6_must_be_null
        0, // reserved7_must_be_null
        0, // reserved8_must_be_null
        0, // reserved9_must_be_null
        0, // reserved10_must_be_null
        NULL // _for_server_internal_use
};

extern "C"
a_v3_extfn_aggregate *my_integer_sum()
{
  return &my_integer_sum_descriptor;
}
```

UDAF 示例：my_bit_xor 定义

“my_bit_xor”示例与 SA 内置的 BIT_XOR 相似，只是 my_bit_xor 仅对不带符号的整数进行操作。

my_bit_xor 定义

由于输入和输出数据类型是相同的，可使用普通的 _next_value_extfn 和 _evaluate_extfn 函数来累积子集合值并生成超级集合结果。

```
#include "extfnapiv3.h"
#include <stdlib.h>
#include <assert.h>


//  Generic aggregate UDF that exclusive-ORs a set of
//  unsigned integer arguments, and whenever asked
//  returns the resulting unsigned integer result.
//
//  The start function creates a little structure for
//  the running result, and the finish function then
//  deallocates it.
//
//  Since there are no aggregate usage restrictions
//  for this UDAF, the corresponding SQL declaration
//  will look like:
//
//            CREATE AGGREGATE FUNCTION my_bit_xor(IN arg1 UNSIGNED
INT)
//                      RETURNS UNSIGNED INT
```

```
//                       ON EMPTY INPUT RETURNS NULL
//                       EXTERNAL NAME 'my_bit_xor@libudfex'


typedef struct my_xor_result {
  a_sql_uint64 _num_nonnulls_seen;
  a_sql_uint32 _xor_result;
} my_xor_result;

#if defined __cplusplus
extern "C" {
#endif

static void my_xor_start(a_v3_extfn_aggregate_context *cntxt)
{
}


static void my_xor_finish(a_v3_extfn_aggregate_context *cntxt)
{
}


static void my_xor_reset(a_v3_extfn_aggregate_context *cntxt)
{
  my_xor_result *cptr = (my_xor_result *)cntxt-
>_user_calculation_context;
  cptr->_xor_result = 0;
  cptr->_num_nonnulls_seen = 0;
}


static void my_xor_next_value(a_v3_extfn_aggregate_context *cntxt,
                              void *arg_handle)
{
  an_extfn_value  arg;
  a_sql_uint32 arg1;

  my_xor_result *cptr = (my_xor_result *)cntxt-
>_user_calculation_context;

  //  Get the one argument, and add it to the total
  if (cntxt->get_value( arg_handle, 1, &arg) && arg.data) {
    arg1 = *((a_sql_uint32 *)arg.data);
    cptr->_xor_result ^= arg1;
    cptr->_num_nonnulls_seen++;
  }
}


static void my_xor_drop_value(a_v3_extfn_aggregate_context *cntxt,
                              void *arg_handle)
{
  an_extfn_value  arg;
  a_sql_uint32 arg1;
  my_xor_result *cptr = (my_xor_result *)cntxt-
```

```
>_user_calculation_context;

  //  Get the one argument, and remove it from the total
  if (cntxt->get_value( arg_handle, 1, &arg) && arg.data) {
    arg1 = *((a_sql_uint32 *)arg.data);
    cptr->_xor_result ^= arg1;
    cptr->_num_nonnulls_seen--;
  }
}


static void my_xor_evaluate(a_v3_extfn_aggregate_context *cntxt,
                            void *arg_handle)
{
  an_extfn_value  outval;
  my_xor_result *cptr = (my_xor_result *)cntxt-
>_user_calculation_context;

  outval.type = DT_UNSINT;
  outval.piece_len = sizeof(a_sql_uint32);
  if (cptr->_num_nonnulls_seen > 0) {
    outval.data = &cptr->_xor_result;
  } else {
    outval.data = 0;
  }
  cntxt->set_value( arg_handle, &outval, 0 );
}


static void my_xor_cum_evaluate(a_v3_extfn_aggregate_context
*cntxt,
                                void *arg_handle)
{
  an_extfn_value  outval;
  an_extfn_value  arg;
  a_sql_uint32 arg1;
  my_xor_result *cptr = (my_xor_result *)cntxt-
>_user_calculation_context;

  //  Get the one argument, and include it in the result,
  //  unless that input value is null.
  //
  if (cntxt->get_value( arg_handle, 1, &arg) && arg.data) {
    arg1 = *((a_sql_uint32 *)arg.data);
    cptr->_xor_result ^= arg1;
    cptr->_num_nonnulls_seen++;
  }

  //  Then set the output result value
  outval.type = DT_UNSINT;
  outval.piece_len = sizeof(a_sql_uint32);
  if (cptr->_num_nonnulls_seen > 0) {
    outval.data = &cptr->_xor_result;
  } else {
    outval.data = 0;
  }
```

```
  cntxt->set_value( arg_handle, &outval, 0 );
}


static a_v3_extfn_aggregate my_xor_descriptor =
    {
        &my_xor_start,
        &my_xor_finish,
        &my_xor_reset,
        &my_xor_next_value,
        &my_xor_evaluate,
        &my_xor_drop_value,
        &my_xor_cum_evaluate,
        &my_xor_next_value,
        &my_xor_drop_value,
        &my_xor_evaluate,
        NULL, // reserved1_must_be_null
        NULL, // reserved2_must_be_null
        NULL, // reserved3_must_be_null
        NULL, // reserved4_must_be_null
        NULL, // reserved5_must_be_null
        0,    // indicators
        ( short )sizeof( my_xor_result ), // context size
        8,    // context alignment
        0.0,  // external_bytes_per_group
        0.0,  // external bytes per row
        0,    // reserved6_must_be_null
        0,    // reserved7_must_be_null
        0,    // reserved8_must_be_null
        0,    // reserved9_must_be_null
        0,    // reserved10_must_be_null
        NULL  // _for_server_internal_use
    };

a_v3_extfn_aggregate *my_bit_xor()
{
  return &my_xor_descriptor;
}

#if defined __cplusplus
}
#endif
```

UDAF 示例：my_bit_or 定义

“my_bit_or”示例与 SA 内置的 BIT_OR 相似，只是 my_bit_or 仅对不带符号的整数进行操作并且只能用作简单集合。

my_bit_or 定义

在某种程度上，“my_bit_or”定义要比“my_bit_xor”示例简单。

```
#include "extfnapiv3.h"
#include <stdlib.h>
#include <assert.h>
```

```
//  A simple (non-OLAP) aggregate UDF that ORs a set
//  of unsigned integer arguments, and whenever asked
//  returns the resulting unsigned integer result.
//
//  The start function creates a little structure for
//  the running result, and the finish function then
//  deallocates it.
//
//  The aggregate usage restrictions for this UDAF
//  only allow its use as a simple aggregate, so the
//  corresponding SQL declaration will look like:
//
//      CREATE AGGREGATE FUNCTION my_bit_or(IN arg1 UNSIGNED INT)
//            RETURNS UNSIGNED INT
//                      ON EMPTY INPUT RETURNS NULL
//            OVER NOT ALLOWED
//            EXTERNAL NAME 'my_bit_or@libudfex'


typedef struct my_or_result {
  a_sql_uint32 _or_result;
  a_sql_uint32 _non_null_seen;
} my_or_result;



#if defined __cplusplus
extern "C" {
#endif

static void my_or_start(a_v3_extfn_aggregate_context *cntxt)
{
}

static void my_or_finish(a_v3_extfn_aggregate_context *cntxt)
{
}

static void my_or_reset(a_v3_extfn_aggregate_context *cntxt)
{
  my_or_result *cptr = (my_or_result *)cntxt-
>_user_calculation_context;
  cptr->_or_result = 0;
  cptr->_non_null_seen = 0;
}


static void my_or_next_value(a_v3_extfn_aggregate_context *cntxt,
                             void *arg_handle)
{
  an_extfn_value  arg;
  a_sql_uint32 arg1;

  my_or_result *cptr = (my_or_result *)cntxt-
>_user_calculation_context;
```

```
  //  Get the one argument, and add it to the total
  if (cntxt->get_value( arg_handle, 1, &arg ) && arg.data)
  {
      arg1 = *((a_sql_uint32 *)arg.data);
      cptr->_or_result |= arg1;
      cptr->_non_null_seen = 1;
  }
}


static void my_or_evaluate(a_v3_extfn_aggregate_context *cntxt,
                           void *arg_handle)
{
  an_extfn_value  outval;
  my_or_result *cptr = (my_or_result *)cntxt-
>_user_calculation_context;

  outval.type = DT_UNSINT;
  outval.piece_len = sizeof(a_sql_uint32);
  if (cptr->_non_null_seen)
  {
      outval.data = &cptr->_or_result;
  }
  else
  {
      // Return null if no values seen
      outval.data = 0;
  }
  cntxt->set_value( arg_handle, &outval, 0 );
}



static a_v3_extfn_aggregate my_or_descriptor =
{
    &my_or_start,
    &my_or_finish,
    &my_or_reset,
    &my_or_next_value,
    &my_or_evaluate,
    NULL, // drop_val_extfn
    NULL, // cume_eval,
    NULL, // next_subaggregate_extfn
    NULL, // drop_subaggregate_extfn
    NULL, // evaluate_superaggregate_extfn
    NULL, // reserved1_must_be_null
    NULL, // reserved2_must_be_null
    NULL, // reserved3_must_be_null
    NULL, // reserved4_must_be_null
    NULL, // reserved5_must_be_null
    0, // indicators
    ( short )sizeof( my_or_result ), // context size
    8, // context alignment
    0.0, //external_bytes_per_group
    0.0, // external bytes per row
```

```
    0, // reserved6_must_be_null
    0, // reserved7_must_be_null
    0, // reserved8_must_be_null
    0, // reserved9_must_be_null
    0, // reserved10_must_be_null
    NULL // _for_server_internal_use
};


extern "C"
a_v3_extfn_aggregate *my_bit_or()
{
  return &my_or_descriptor;
}


#if defined __cplusplus
}
#endif
```

UDAF 示例：my_interpolate 定义

“my_interpolate”示例为 OLAP 样式的 UDAF，它试图通过将任意一组邻近的空值线性插值到每个方向最近的非空值，来将空值填充到序列中。

my_interpolate 定义

为合理控制运算成本，运行 my_interpolate 时必须使用固定宽度行式窗口，但用户可以根据预期的邻近 NULL 值的最大数目来设置窗口的宽度。 如果给定行上的输入不是 NULL，则该行的结果与输入值相同。 此函数采用一组双精度浮点值，并产生一组双精度值结果。

```
#include "extfnapiv3.h"
#include <stdlib.h>
#include <assert.h>

//  MY_INTERPOLATE
//
//  OLAP-style aggregate UDF that accepts a double precision
//  floating point argument.  If the current argument value is
//  not NULL, then the result value is the same as the
//  argument value.  On the other hand, if the current row's
//  argument value is NULL, then the result, where possible,
//  will be the arithmetic interpolation across the nearest
//  preceding and nearest following values that are not NULL.
//  In all cases the result is also a double precision value.
//
//  The start function creates a structure for maintaining the
//  argument values within the window including their NULLness.
//  The finish function then deallocates this structure.
//
//  Since there are some strict aggregate usage restrictions
//  for this aggregate (must be used with a row-based window
//  frame that includes the current row), the corresponding
//  SQL declaration will look like:
```

```
//
//      CREATE AGGREGATE FUNCTION my_interpolate(IN arg1 DOUBLE)
//                       RETURNS DOUBLE
//                       OVER REQUIRED
//                       WINDOW FRAME REQUIRED
//                               RANGE NOT ALLOWED
//                               PRECEDING REQUIRED
//                               UNBOUNDED PRECEDING NOT ALLOWED
//                               FOLLOWING REQUIRED
//                               UNBOUNDED FOLLOWING NOT ALLOWED
//                       EXTERNAL NAME  'my_interpolate@libudfex'


typedef struct my_window {
  int      _allocated_elem;
  int      _first_used;
  int      _next_insert_loc;
  int     *_is_null;
  double  *_dbl_val;
  int      _num_rows_in_frame;
} my_window;



#if defined __cplusplus
extern "C" {
#endif


static void my_interpolate_reset(a_v3_extfn_aggregate_context
*cntxt)
{
  assert(cntxt->_user_data);
  my_window *cptr = (my_window *)cntxt->_user_data;

  cptr->_first_used = 0;
  cptr->_next_insert_loc = 0;
  cptr->_num_rows_in_frame = 0;
  for (int i=0; i<cptr->_allocated_elem; i++) {
    cptr->_is_null[i] = 1;
  }
}


static void my_interpolate_start(a_v3_extfn_aggregate_context
*cntxt)
{
  my_window *cptr = (my_window *)cntxt->_user_data;

  // Make sure function was defined correctly
  if (!cntxt->_is_window_used)
  {
      cntxt->set_error(cntxt, 20001, "Function requires window");
      return;
  }
  if (cntxt->_window_has_unbounded_preceding ||
```

```
      cntxt->_window_has_unbounded_following)
  {
      cntxt->set_error(cntxt, 20002, "Window cannot be unbounded");
      return;
  }
  if (cntxt->_window_is_range_based)
  {
      cntxt->set_error(cntxt, 20003, "Window must be row based");
      return;
  }

  if (!cptr) {
    //
    cptr = (my_window *)malloc(sizeof(my_window));
    if (cptr) {
      cptr->_is_null = 0;
      cptr->_dbl_val = 0;
      cptr->_num_rows_in_frame = 0;
      cptr->_allocated_elem = ( int )cntxt->_max_rows_in_frame;
      cptr->_is_null = (int *)malloc(cptr->_allocated_elem
                                     * sizeof(int));
      cptr->_dbl_val = (double *)malloc(cptr->_allocated_elem
                                        * sizeof(double));
      cntxt->_user_data = cptr;
    }
  }
  if (!cptr || !cptr->_is_null || !cptr->_dbl_val) {
      // Terminate this query
      cntxt->set_error(cntxt, 20000, "Unable to allocate memory");
      return;
  }
  my_interpolate_reset(cntxt);
}


static void my_interpolate_finish(a_v3_extfn_aggregate_context
*cntxt)
{
  if (cntxt->_user_data) {
    my_window *cptr = (my_window *)cntxt->_user_data;
    if (cptr->_is_null) {
      free(cptr->_is_null);
      cptr->_is_null = 0;
    }
    if (cptr->_dbl_val) {
      free(cptr->_dbl_val);
      cptr->_dbl_val = 0;
    }
    free(cntxt->_user_data);
    cntxt->_user_data = 0;
  }
}


static void my_interpolate_next_value(a_v3_extfn_aggregate_context
*cntxt,
```

```
                                        void *arg_handle)
{
  an_extfn_value  arg;
  double arg1;
  my_window *cptr = (my_window *)cntxt->_user_data;

  //  Get the one argument, and stash its value
  //  within the rotating window arrays
  //
  int curr_cell_num = cptr->_next_insert_loc % cptr-
>_allocated_elem;
  if (cntxt->get_value( arg_handle, 1, &arg ) && arg.data != NULL ) {
    arg1 = *((double *)arg.data);
    cptr->_dbl_val[curr_cell_num] = arg1;
    cptr->_is_null[curr_cell_num] = 0;
  } else {
    cptr->_is_null[curr_cell_num] = 1;
  }

  //  Then increment the insertion location and number of rows in
frame
  cptr->_next_insert_loc = ((cptr->_next_insert_loc + 1)
                            % cptr->_allocated_elem);
  cptr->_num_rows_in_frame++;
}


static void my_interpolate_drop_value(a_v3_extfn_aggregate_context
*cntxt,
                                        void * /*arg_handle*/)
{
  my_window *cptr = (my_window *)cntxt->_user_data;

  //  Drop one value from the window by incrementing past it and
  // decrement the number of rows in the frame
  cptr->_first_used = ((cptr->_first_used + 1) % cptr-
>_allocated_elem);
  cptr->_num_rows_in_frame--;
}


static void my_interpolate_evaluate(a_v3_extfn_aggregate_context
*cntxt,
                                     void *arg_handle)
{

  an_extfn_value  outval;
  my_window *cptr = (my_window *)cntxt->_user_data;
  double  result;
  int     result_is_null = 1;
  double  preceding_value;
  int     preceding_value_is_null = 1;
  double  preceding_distance = 0;
  double  following_value;
  int     following_value_is_null = 1;
  double  following_distance = 0;
```

```
  int j;

  //  Determine which cell is the current cell
  int curr_cell_num =
      ((int)(cntxt->_result_row_from_start_of_partition-1))%cptr-
>_allocated_elem;
  int tmp_cell_num;

  int result_row_offset_from_start_of_frame = cptr->_first_used <=
curr_cell_num ?
          ( curr_cell_num - cptr->_first_used ) :
          ( curr_cell_num + cptr->_allocated_elem - cptr-
>_first_used );




  //  Compute the result value
  if (cptr->_is_null[curr_cell_num] == 0) {
    //
    //  If the current rows input value is not NULL, then there is
    //  no need to interpolate, just use that input value.
    //
    result = cptr->_dbl_val[curr_cell_num];
    result_is_null = 0;
    //
  } else {
    //
    //  If the current rows input value is NULL, then we do
    //  need to interpolate to find the correct result value.
    //  First, find the nearest following non-NULL argument
    //  value after the current row.
    //
    int rows_following = cptr->_num_rows_in_frame -
                result_row_offset_from_start_of_frame - 1;
    for (j=0; j<rows_following; j++) {
      tmp_cell_num = ((curr_cell_num + j + 1) % cptr-
>_allocated_elem);
      if (cptr->_is_null[tmp_cell_num] == 0) {
          following_value = cptr->_dbl_val[tmp_cell_num];
          following_value_is_null = 0;
          following_distance = j + 1;
          break;
      }
    }
    //  Second, find the nearest preceding non-NULL
    //  argument value before the current row.
    //
    int rows_before = result_row_offset_from_start_of_frame;
    for (j=0; j<rows_before; j++) {
     tmp_cell_num = ((curr_cell_num + cptr->_allocated_elem - j - 1)
                      % cptr->_allocated_elem);
      if (cptr->_is_null[tmp_cell_num] == 0) {
          preceding_value = cptr->_dbl_val[tmp_cell_num];
          preceding_value_is_null = 0;
          preceding_distance = j + 1;
```

```
          break;
      }
    }
    //  Finally, see what we can come up with for a result value
    //
    if (preceding_value_is_null && !following_value_is_null) {
      //
      //  No choice but to mirror the nearest following non-NULL value
      //  Example:
      //
      //    Inputs:  NULL    Result of my_interpolate:  40.0
      //             NULL                               40.0
      //             40.0                               40.0
      //
      result = following_value;
      result_is_null = 0;
      //
    } else if (!preceding_value_is_null && following_value_is_null) {
      //
      //  No choice but to mirror the nearest preceding non-NULL value
      //  Example:
      //
      //    Inputs:  10.0    Result of my_interpolate:  10.0
      //             NULL                               10.0
      //
      result = preceding_value;
      result_is_null = 0;
      //
    } else if (!preceding_value_is_null && !following_value_is_null)
{
      //
      //  Here we get to do real interpolation based on the
      //  nearest preceding non-NULL value, the nearest following
      //  non-NULL value, and the relative distances to each.
      //  Examples:
      //
      //    Inputs:  10.0    Result of my_interpolate:  10.0
      //             NULL                               20.0
      //             NULL                               30.0
      //             40.0                               40.0
      //
      //    Inputs:  10.0    Result of my_interpolate:  10.0
      //             NULL                               25.0
      //             40.0                               40.0
      //
      result = (  preceding_value
                  + (  (following_value - preceding_value)
                       * (  preceding_distance
                          / (preceding_distance +
following_distance))));
      result_is_null = 0;
    }
  }

  //  And last, pass the result value out
  outval.type = DT_DOUBLE;
```

```
  outval.piece_len = sizeof(double);
  if (result_is_null) {
    outval.data = 0;
  } else {
    outval.data = &result;
  }
  cntxt->set_value( arg_handle, &outval, 0 );
}


static a_v3_extfn_aggregate my_interpolate_descriptor =
    {
        &my_interpolate_start,
        &my_interpolate_finish,
        &my_interpolate_reset,
        &my_interpolate_next_value, //( timeseries_expression )
        &my_interpolate_evaluate,
        &my_interpolate_drop_value,
       NULL, // cume_eval,
       NULL, // next_subaggregate_extfn
       NULL, // drop_subaggregate_extfn
       NULL, // evaluate_superaggregate_extfn
       NULL, // reserved1_must_be_null
       NULL, // reserved2_must_be_null
       NULL, // reserved3_must_be_null
       NULL, // reserved4_must_be_null
       NULL, // reserved5_must_be_null
       0, // indicators
       0, // context size
       0, // context alignment
       0.0, //external_bytes_per_group
       ( double )sizeof( double ), // external bytes per row
       0, // reserved6_must_be_null
       0, // reserved7_must_be_null
       0, // reserved8_must_be_null
       0, // reserved9_must_be_null
       0, // reserved10_must_be_null
       NULL // _for_server_internal_use
    };

a_v3_extfn_aggregate *my_interpolate()
  { return &my_interpolate_descriptor; }

#if defined __cplusplus
  }
#endif
```

集合用户定义的函数的上下文存储

上下文变量控制集合函数的中间结果是否将由 UDF 本身管理（强制 IQ 服务器以串行方式运行 UDF），或者内存是否将由 IQ 服务器管理。 上下文区域用于在同一查询内（尤其是 OLAP 样式的查询内）的 UDF 的多次调用之间传输或交换数据。

如果 _calculation_context_size 设置为 0，则 UDF 需要管理内存中的所有中间结果，强制 IQ 服务器按顺序对数据调用 UDF（而不能够在 OLAP 查询期间并行调用 UDF 的多个实例）。

如果 _calculation_context_size 设置为非零值，则 IQ 服务器管理 UDF 的每次调用的单独上下文区域，从而允许并行调用 UDF 的多个实例。 为了最高效地利用内存，请考虑将 _calculation_context_alignment 设置为比缺省值小的值（根据所需上下文存储的大小）。

有关上下文存储的详细信息，请参见集合 UDF 描述符结构 （第 47 页）“集合 UDF 描述符结构”一节中的 _calculation_context_size 和 _calculation_context_alignment 的说明。 这些变量位于描述符结构的末尾附近。

有关上下文存储的使用的详细讨论，请参见计算上下文 （第 50 页）。

重要： 若要将内存中的中间结果存储在集合 UDF 中，请使用 _start_extfn 函数初始化内存，然后使用 _finish_extfn 函数清除并释放任何内存。

UDF 回调函数和调用模式

调用模式为函数在收集结果时所执行的步骤。

UDF 和 UDAF 回调函数

下列这组回调函数由引擎通过 a_v3_extfn_scalar_context 结构提供，并在用户的 UDF 函数中使用：

- **get_value** – 此函数在求值方法中用于检索每个输入参数的值。 对于窄型参数数据类型（小于 256 字节），调用 get_value 就足以检索整个参数值。 而对于较宽的参数数据类型，如果 an_extfn_value 结构内传递到此回调的 piece_len 字段返回的值小于 total_len 字段中的值，请使用 get_piece 回调来检索输入值的其余部分。
- **get_piece** – 此函数用于检索长参数输入值的后续片段。
- **get_is_constant** – 此函数确定指定的输入参数值是否为常量。 这对优化 UDF 来说比较有用。例如，工作可在首次调用 _evaluate_extfn 函数时执行一次，而不是在每次求值调用时执行。
- **set_value** – 此函数在求值函数中用于将该调用的 UDF 的结果值告知服务器。 如果结果数据类型为窄型，则调用 set_value 就已足够。 但是，如果结果数据值为宽型，则需要多次调用 set_value 来传递整个值，并且除最后一个片段外每个片段的回调附加参数都应为真。 要返回 NULL 结果，UDF 应当将结果值的 an_extfn_value 结构中的数据字段设为空指针。
- **get_is_cancelled** – 此函数用于确定语句是否已被取消。 如果 UDF 入口点正在执行时间较长（许多秒）的工作，则可能的话，它应当每秒或每两秒调用 get_is_cancelled 回调，以查看用户是否中断了当前的语句。 如果语句没有被中断，则返回值为 0。

  Sybase IQ 可以处理极大的数据集，并且一些查询会运行相当长的时间。 有时，查询的执行需要特别长的时间。 如果查询的执行需要的时间太长，而无法完成，SQL 客户端允许用户取消查询。 本地函数跟踪用户何时取消了查询。 UDF 的写入方式还必须跟踪用户是否已取消查询。 换句话说，UDF 应支持用户取消调用 UDF 的长时间运行的查询的能力。
- **set_error** – 此函数可用于将错误传回服务器，并最终传给用户。 如果 UDF 入口点遇到应导致向用户传回错误消息的错误，则可调用此回调例程。 在调用时，set_error 会回退当前语句，用户会收到 `Error from external UDF: error_desc_string`，并且 SQLCODE 是提供的 error_number 的负值形式。 为避免与现有的错误冲突，UDF 应当使用介于 17000 和 99999 之间的 error_number 值。“error_desc_string”的最大长度为 140 个字符。
- **log_message** — 此函数用于将消息发送到服务器的消息日志。字符串必须为不超过 255 字节的可打印文本字符串。

- **convert_value** — 此函数允许在数据类型之间进行数据转换。主要用处为 DT_DATE、DT_TIME、DT_TIMESTAMP 和 DT_TIMESTAMP_STRUCT 之间的转换。输入和输出 an_extfn_value 会传递到此函数。

标量 UDF 调用模式

为标量 UDF 调用模式提供的函数指针的预期调用模式。

```
_start_extfn(if supplied)
_evaluate_extfn (repeated 0 to numerous times)
_finish_extfn(if supplied)
```

集合 UDF 调用模式

与标量调用模式相比，用户提供的集合 UDF 函数的调用模式有很大差别并且更为复杂。

使用下列表定义的示例：

```
create table t (a int, b int, c int)
insert into t values (1, 1, 1)
insert into t values (2, 1, 1)
insert into t values (3, 1, 1)
insert into t values (4, 2, 1)
insert into t values (5, 2, 1)
insert into t values (6, 2, 1)
```

使用下列缩写形式：

RR = a_v3_extfn_aggregate_context._result_row_offset_from_start_of_partition — 此值表示在其中计算值的当前分区中当前的行号。该值在窗口集合期间设置，主要用于未受限制的窗口的求值步骤；它在所有求值调用上均可用。

Sybase IQ 是多用户应用程序。 多个用户可以同时执行同一 UDF。 某些 OLAP 查询在同一查询中多次执行 UDF，有时并行执行。

简单拆组集合

简单拆组集合调用模式对所有行的输入值进行总计，并产生一个结果。

查询

```
select my_sum(a) from t
```

调用模式

```
_start_extfn(cntxt)
_reset_extfn(cntxt)
_next_value_extfn(cntxt, args)  -- input a=1
```

```
_next_value_extfn(cntxt, args)  -- input a=2
_next_value_extfn(cntxt, args)  -- input a=3
_next_value_extfn(cntxt, args)  -- input a=4
_next_value_extfn(cntxt, args)  -- input a=5
_next_value_extfn(cntxt, args)  -- input a=6
_evaluate_extfn(cntxt, args)    -- returns 21
_finish_extfn(cntxt)
```

结果

```
my_sum(a)
21
```

简单分组集合

简单分组集合调用模式对组中所有行的输入值进行总计，并产生一个结果。 **_reset_extfn** 标识组的开头。

查询

```
select b, my_sum(a) from t group by b order by b
```

调用模式

```
_start_extfn(cntxt)
_reset_extfn(cntxt)
_next_value_extfn(cntxt, args)  -- input a=1
_next_value_extfn(cntxt, args)  -- input a=2
_next_value_extfn(cntxt, args)  -- input a=3
_evaluate_extfn(cntxt, args)    -- returns  6
_reset_extfn(cntxt)
_next_value_extfn(cntxt, args)  -- input a=4
_next_value_extfn(cntxt, args)  -- input a=5
_next_value_extfn(cntxt, args)  -- input a=6
_evaluate_extfn(cntxt, args)    -- returns 15
_finish_extfn(cntxt)
```

结果

```
b,   my_sum(a)
1,   6
2,   15
```

带未受限制窗口的 OLAP 样式集合调用模式

对“b”分区会创建与对“b”分组相同的分区。 未受限制的窗口会导致为分区的每行求“a”的值。 由于这是未受限制的查询，在求值循环前会首先将所有值填充到 UDF。 _window_has_unbounded_preceding 和 _window_has_unbounded_following 的上下文指示符设置为 1

查询

```
select b, my_sum(a) over (partition by b rows between
unbounded preceding and
```

```
unbounded following)
from t
```

调用模式

```
_start_extfn(cntxt)
_reset_extfn(cntxt)
_next_value_extfn(cntxt, args)    input a=1
_next_value_extfn(cntxt, args)    input a=2
_next_value_extfn(cntxt, args)    input a=3
_evaluate_extfn(cntxt, args)         rr=1  returns 6
_evaluate_extfn(cntxt, args)         rr=2  returns 6
_evaluate_extfn(cntxt, args)      rr=3  returns 6
_reset_extfn(cntxt)
_next_value_extfn(cntxt, args)    input a=4
_next_value_extfn(cntxt, args)    input a=5
_next_value_extfn(cntxt, args)    input a=6
_evaluate_extfn(cntxt, args)      rr=1  returns 15
_evaluate_extfn(cntxt, args)      rr=2  returns 15
_evaluate_extfn(cntxt, args)      rr=3  returns 15
_finish_extfn(cntxt)
```

结果

```
b,  my_sum(a)
1,  6
1,  6
1,  6
2,  15
2,  15
2,  15
```

OLAP 样式的未优化累计窗口集合

如果未提供 _evaluate_cumulative_extfn，此累计总和通过下列调用模式进行求值，该调用模式的效率没有 _evaluate_cumulative_extfn 的高。

查询

```
select b, my_sum(a) over (partition by b
rows between unbounded preceding and current row)
from t
order by b
```

调用模式

```
_start_extfn(cntxt)
_reset_extfn(cntxt)
_next_value_extfn(cntxt, args)     -- input a=1
_evaluate_extfn(cntxt, args)    -- returns 1
_next_value_extfn(cntxt, args)  -- input a=2
_evaluate_extfn(cntxt, args)    -- returns 3
_next_value_extfn(cntxt, args)  -- input a=3
_evaluate_extfn(cntxt, args)    -- returns 6
_reset_extfn(cntxt)
```

```
_next_value_extfn(cntxt, args)  -- input a=4
_evaluate_extfn(cntxt, args)    -- returns 4
_next_value_extfn(cntxt, args)  -- input a=5
_evaluate_extfn(cntxt, args)    -- returns 9
_next_value_extfn(cntxt, args)  -- input a=6
_evaluate_extfn(cntxt, args)    -- returns 15
_finish_extfn(cntxt)
```

结果

```
b,  my_sum(a)
1,  1
1,  3
1,  6
2,  4
2,  9
2,  15
```

OLAP 样式的优化累计窗口集合

如果提供了 _evaluate_cumulative_extfn，此累计总和会在如下位置求值：next_value/ 求值序列组合到每个分区中每一行的单个 _evaluate_cumulative_extfn 调用中。

查询

```
select b, my_sum(a) over (partition by b rows between unbounded
preceding and current row)
from t
order by b
```

调用模式

```
_start_extnfn(cntxt)
_reset_extfn(cntxt)
_evaluate_cumulative_extfn(cntxt, args) -- input a=1 returns 1
_evaluate_cumulative_extfn(cntxt, args) -- input a=2 returns 3
_evaluate_cumulative_extfn(cntxt, args) -- input a=3 returns 6
_reset_extfn(cntxt)
_evaluate_cumulative_extfn(cntxt, args) -- input a=4 returns 4
_evaluate_cumulative_extfn(cntxt, args) -- input a=5 returns 9
_evaluate_cumulative_extfn(cntxt, args) -- input a=6 returns 15
_finish_extfn(cntxt)
```

结果

```
b,  my_sum(a)
1,  1
1,  3
1,  6
2,  4
2,  9
2,  15
```

OLAP 样式的未优化移动窗口集合

如果未提供 _drop_value_extfn 函数，此移动窗口总和通过如下模式进行求值，这与使用 _drop_value_extfn 相比效率要低得多：

查询

```
select b, my_sum(a) over (partition by b rows between 1 preceding and
current row)
from t
```

调用模式

```
_start_extfn(cntxt)
_reset_extfn(cntxt)
_next_value_extfn(cntxt, args)         input a=1
_evaluate_extfn(cntxt, args)           returns 1
_reset_extfn(cntxt)
_next_value_extfn(cntxt, args)         input a=1
_next_value_extfn(cntxt, args)         input a=2
_evaluate_extfn(cntxt, args)           returns 3
_reset_extfn(cntxt)
_next_value_extfn(cntxt, args )        input a=2
_next_value_extfn(cntxt, args )        input a=3
_evaluate_extfn(cntxt, args)           returns 5
_reset_extfn(cntxt)
_next_value_extfn(cntxt, args)         input a=4
_evaluate_extfn(cntxt, args)              returns 4
_reset_extfn(cntxt)
_next_value_extfn(cntxt, args)         input a=4
_next_value_extfn(cntxt, args)         input a=5
_evaluate_extfn(cntxt, args)              returns 9
_reset_extfn(cntxt)
_next_value_extfn(cntxt, args)         input a=5
_next_value_extfn(cntxt, args)         input a=6
_evaluate_extfn(cntxt, args)              returns 11
_finish_extfn(cntxt)
```

结果

```
b,  my_sum(a)
1,  1
1,  3
1,  5
2,  4
2,  9
2,  11
```

OLAP 样式的优化移动窗口集合

如果提供了 _drop_value_extfn 函数，则此移动窗口总和通过此调用模式进行求值，这要比使用 _drop_value_extfn 效率高

查询

```
select b, my_sum(a) over (partition by b rows between 1 preceding and
current row)
from t
```

调用模式

```
_start_extfn(cntxt)
_reset_extfn(cntxt)
_evaluate_aggregate_extfn(cntxt, args)              -- returns  1
_evaluate_aggregate_extfn(cntxt, args)                 -- returns  3
_drop_value_extfn(cntxt)                             -- input a=1
_next_value_extfn(cntxt, args)      -- input a=3
_evaluate_aggregate_extfn(cntxt, args)                 -- returns  5
_reset_extfn(cntxt)
_next_value_extfn(cntxt, args)      -- input a=4
_evaluate_aggregate_extfn(cntxt, args)                 -- returns  4
_next_value_extfn(cntxt, args)      -- input a=5
_evaluate_aggregate_extfn(cntxt, args)                 -- returns 9
_drop_value_extfn(cntxt)                          -- input a=4
_next_value_extfn(cntxt, args)      -- input a=6
_evaluate_aggregate_extfn(cntxt, args)              -- returns  11
_finish_extfn(cntxt)
```

结果

```
b,  my_sum(a)
1,  1
1,  3
1,  5
2,  4
2,  9
2,  11
```

OLAP 样式未优化移动窗口的下列集合

如果未提供 _drop_value_extfn 函数，此移动窗口总和通过如下调用模式进行求值。此情形与之前的移动窗口示例相似，但被求值的行不是由下一值函数所给的最后一行。

查询

```
select b, my_sum(a) over (partition by b rows between 1 preceding and
1 following)
from t
```

调用模式

```
_start_extfn(cntxt)
_reset_extfn(cntxt)
_next_value_extfn(cntxt, args)         input a=1
_next_value_extfn(cntxt, args)       input a=2
_evaluate_extfn(cntxt, args)         returns 3
_reset_extfn(cntxt)
_next_value_extfn(cntxt, args)       input a=1
_next_value_extfn(cntxt, args)       input a=2
_next_value_extfn(cntxt, args)       input a=3
_evaluate_extfn(cntxt, args)         returns 6
_reset_extfn(cntxt)
_next_value_extfn(cntxt, args)       input a=2
_next_value_extfn(cntxt, args)       input a=3
_evaluate_extfn(cntxt, args)         returns 5
_reset_extfn(cntxt)
_next_value_extfn(cntxt, args)       input a=4
_next_value_extfn(cntxt, args)       input a=5
_evaluate_extfn(cntxt, args)         returns 9
_reset_extfn(cntxt)
_next_value_extfn(cntxt, args)       input a=4
_next_value_extfn(cntxt, args)       input a=5
_next_value_extfn(cntxt, args)       input a=6
_evaluate_extfn(cntxt, args)         returns 15
_reset_extfn(cntxt)
_next_value_extfn(cntxt, args)       input a=5
_next_value_extfn(cntxt, args)       input a=6
_evaluate_extfn(cntxt, args)         returns 11
_finish_extfn(cntxt)
```

结果

```
b,  my_sum(a)
1,  3
1,  6
1,  5
2,  9
2,  15
2,  11
```

OLAP 样式优化移动窗口的下列集合

如果提供了 _drop_value_extfn 函数，此移动窗口总和通过如下调用模式进行求值。同样，此情形与之前的移动窗口示例相似，但被求值的行不是由下一值函数所给的最后一行。

查询

```
select b, my_sum(a) over (partition by b rows between 1 preceding and
1 following)
from t
```

调用模式

```
_start_extfn(cntxt)
_reset_extfn(cntxt)
_next_value_extfn(cntxt, args)        input a=1
_next_value_extfn(cntxt, args)      input a=2
_evaluate_extfn(cntxt, args)        returns 3
_next_value_extfn(cntxt, args)      input a=3
_evaluate_extfn(cntxt, args)        returns 6
_dropvalue_extfn(cntxt)                          input a=1
_evaluate_extfn(cntxt, args)        returns 5
_reset_extfn(cntxt)
_next_value_extfn(cntxt, args)      input a=4
_next_value_extfn(cntxt, args)      input a=5
_evaluate_extfn(cntxt, args)        returns 9
_next_value_extfn(cntxt, args)      input a=6
_evaluate_extfn(cntxt, args)        returns 15
_dropvalue_extfn(cntxt)                             input a=4
_evaluate_extfn(cntxt, args)        returns 11
_finish_extfn(cntxt)
```

结果

```
b,  my_sum(a)
1,  3
1,  6
1,  5
2,  9
2,  15
2,  11
```

OLAP 样式的未优化移动窗口（无当前行）

假设 UDF my_sum 的工作方式与内置 SUM 类似。如果未提供 _drop_value_extfn 函数，此移动窗口计数通过如下调用模式进行求值。此情形与之前的移动窗口示例相似，但是当前行不是窗口构架的一个部分。

查询

```
select b, my_sum(a) over (rows between 3 preceding and 1 preceding)
from t
```

调用模式

```
_start_extfn(cntxt)
_reset_extfn(cntxt)
_evaluate_extfn(cntxt, args)          returns NULL
_reset_extfn(cntxt)
_next_value_extfn(cntxt, args)      input a=1
_evaluate_extfn(cntxt, args)          returns 1
_reset_extfn(cntxt)
_next_value_extfn(cntxt, args)      input a=1
_next_value_extfn(cntxt, args)      input a=2
_evaluate_extfn(cntxt, args)          returns 3
```

```
_reset_extfn(cntxt)
_next_value_extfn(cntxt, args)      input a=1
_next_value_extfn(cntxt, args)      input a=2
_next_value_extfn(cntxt, args)      input a=3
_evaluate_extfn(cntxt, args)          returns 6
_reset_extfn(cntxt)
_next_value_extfn(cntxt, args)      input a=2
_next_value_extfn(cntxt, args)      input a=3
_next_value_extfn(cntxt, args)      input a=4
_evaluate_extfn(cntxt, args)          returns 9
_reset_extfn(cntxt)
_next_value_extfn(cntxt, args)      input a=3
_next_value_extfn(cntxt, args)      input a=4
_next_value_extfn(cntxt, args)      input a=5
_evaluate_extfn(cntxt, args)          returns 12
_finish_extfn(cntxt)
```

结果

```
b           my_sum(a)
----------- -------------
          1          NULL
          1             1
          1             3
          2             6
          2             9
          2            12
```

OLAP 样式的优化移动窗口（无当前行）

如果提供了 _drop_value_extfn 函数，此移动窗口计数通过如下调用模式进行求值。此情形与之前的移动窗口示例相似，但是当前行不是窗口构架的一个部分。

查询

```
select b, my_sum(a) over (rows between 3 preceding and 1 preceding)
from t
```

调用模式

```
_start_extfn(cntxt)
_reset_extfn(cntxt)
_evaluate_extfn(cntxt, args)          returns NULL
_next_value_extfn(cntxt, args)      input a=1
_evaluate_extfn(cntxt, args)          returns 1
_next_value_extfn(cntxt, args)      input a=2
_evaluate_extfn(cntxt, args)          returns 3
_next_value_extfn(cntxt, args)      input a=3
_evaluate_extfn(cntxt, args)          returns 6
_dropvalue_extfn(cntxt)                 input a=1
_next_value_extfn(cntxt, args)      input a=4
_evaluate_extfn(cntxt, args)          returns 9
_dropvalue_extfn(cntxt)                 input a=2
_next_value_extfn(cntxt, args)      input a=5
```

```
_evaluate_extfn(cntxt, args)          returns 12
_finish_extfn(cntxt)
```

结果

```
b           my_sum(a)
----------- -------------
          1          NULL
          1             1
          1             3
          2             6
          2             9
          2            12
```

外部函数原型

使用 Sybase IQ 安装目录的子目录中名为 `extfnapiv3.h` 的头文件定义 API。 此标头文件处理外部函数原型与平台有关的功能。

要通知数据库服务器该库不是使用旧 API 写入的，请按如下方式提供一个函数：

```
uint32 extfn_use_new_api( )
```

此函数返回一个不带符号的 32 位整数。 如果返回值为非零，则数据库服务器假定您用的是新 API。

如果 DLL 不导出此函数，则数据库服务器假定使用的是旧 API。 使用新 API 时，返回的值必须为 `extfnapi.v3h` 中定义的 API 版本号。

每个库应该按如下所示实现并导出此函数：

```
unsigned int extfn_use_new_api(void)
    {
        return EXTFN_V3_API;
    }
```

若存在此函数并且其返回 EXTFN_V3_API，将告知 IQ 引擎该库中包含写入到本书中所述的新 API 的 UDF。

函数原型

函数的名称必须与 **CREATE PROCEDURE** 或 **CREATE FUNCTION** 语句中引用的相匹配。将函数声明为：

```
void function-name ( an_extfn_api *api, void *argument-handle )
```

函数必须返回 void，并且必须将用于传递参数的结构和由 SQL 过程提供的参数的句柄用作参数。

an_extfn_api 结构的格式如下：

```
typedef struct an_extfn_api {
short (SQL_CALLBACK *get_value)(
                    void *          arg_handle,
                    a_sql_uint32    arg_num,
```

```
                    an_extfn_value *value
                    );
 short (SQL_CALLBACK *get_piece)(
                    void *          arg_handle,
                    a_sql_uint32    arg_num,
                    an_extfn_value *value,
                    a_sql_uint32    offset
                    );
 short (SQL_CALLBACK *set_value)(
                    void *          arg_handle,
                    a_sql_uint32    arg_num,
                    an_extfn_value *value
                    short          append
                    );
void (SQL_CALLBACK *set_cancel)(
                    void *    arg_handle,
                    void *    cancel_handle
                    );
} an_extfn_api;
```

an_extfn_value 结构的格式如下：

```
typedef struct an_extfn_value {
void *                data;
  a_sql_uint32         piece_len;
  union {
  a_sql_uint32    total_len;
  a_sql_uint32    remain_len;
} len;
  a_sql_data_type        type;
} an_extfn_value;
```

注意

对 OUT 参数调用 get_value 会返回参数的数据类型，并且将数据返回为 NULL。

任何给定参数的 get_piece 函数只能在为相同参数调用 get_value 函数之后立即调用，要返回 NULL，可将 an_extfn_value 中的数据设为 NULL。

set_value 的 append 字段确定提供的数据是否替换（假）或附加到（真）现有的数据。在为相同参数以 append=TRUE 调用 set_value 前，必须先以 append=FALSE 调用。 对于固定长度的数据类型，会忽略 append 字段。

头文件本身包含额外的注释。

索引

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