SYBASE®
An SAP® Company

システム管理ガイド：第 2 巻

Sybase IQ 15.3

ドキュメント ID： DC01145-01-1530-02
改訂： 2011 年 10 月

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

目次

対象読者

このマニュアルは、Sybase® IQ データベースのデータにアクセスするアプリケーションの開発者を対象にしています。

リレーショナル・データベース・システムの基礎知識と、Sybase IQ のユーザ・レベルの基礎的な経験があることを前提にしています。このマニュアルは、他のマニュアルと併用するように構成されています。

プロシージャとバッチの使用

Sybase IQ で使用するためのプロシージャとバッチを作成します。

プロシージャは、すべてのアプリケーションで使えるように、手続き型 SQL 文をデータベースに格納します。これによりデータベースのセキュリティ、効率、標準化を高めることができます。ユーザ定義関数は、クエリやその他の SQL 文で使うための結果を返すプロシージャの一種です。

さまざまな用途で、サーバ側の JDBC は、SQL ストアド・プロシージャよりも柔軟にデータベースにロジックを構築します。 SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - プログラミング』>「SQL Anywhere データ・アクセス API」>「SQL Anywhere JDBC ドライバ」>「JDBC の概要」を参照してください。

バッチは、データベース・サーバにグループとして送られる SQL 文のセットです。制御文などのプロシージャで使用できる機能の多くは、バッチ内でも使用できます。

プロシージャの概要

プロシージャは、手続き型 SQL 文をすべてのアプリケーションで使えるようにデータベースに格納します。 プロシージャでは、制御文を使用して、SQL 文の繰り返し (**LOOP**) や条件付き実行 (**IF** 文と **CASE** 文) ができます。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - SQL の使用法』>「ストアド・プロシージャとトリガ」>「プロシージャ、トリガ、バッチの使用」>「プロシージャとトリガの概要」を参照してください。

注意： Sybase IQ ではトリガをサポートしていません。 SQL Anywhere マニュアルのトリガについての情報は無視してかまいません。

プロシージャの利点

プロシージャの定義はデータベースに置かれており、各データベース・アプリケーションとは区別されています。この区別には多くの利点があります。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - SQL の使用法』>「ストアド・プロシージャとトリガ」>「プロシージャ、トリガ、バッチの使用」>「プロシージャとトリガの利点」を参照してください。

プロシージャ入門

この項では、使用できるプロシージャとその関数について説明します。

ストアド・プロシージャを扱うときに役立つシステム・ストアド・プロシージャとして、**sp_iqprocedure** と **sp_iqprocparm** の 2 つがあります。 **sp_iqprocedure** ストアド・プロシージャは、データベース内のシステムおよびユーザ定義プロシージャに関する情報を表示します。 **sp_iqprocparm** ストアド・プロシージャは、次のカラムのようなストアド・プロシージャのパラメータに関する情報を表示します。

- proc_name
- proc_owner
- parm_name
- parm_type
- parm_mode
- domain_name
- width、scale
- default

参照：

- *プロシージャの結果* (12 ページ)

プロシージャの作成

プロシージャは、**CREATE PROCEDURE** 文を使用して作成します。プロシージャを作成するには、RESOURCE 権限が必要です。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - SQL の使用法』>「ストアド・プロシージャとトリガ」>「プロシージャ、トリガ、バッチの使用」>「プロシージャの概要」>「プロシージャの作成」を参照してください。

Sybase IQ の例

注意： たとえば、Sybase IQ デモ・データベース `iqdemo.db` を使用します。

```
CREATE PROCEDURE new_dept(IN id INT,
             IN name CHAR(35),
                IN head_id INT)
BEGIN
        INSERT
                INTO GROUPO.departments(DepartmentID,
                DepartmentName,
                DepartmentHeadID)
```

```
                values (id, name, head_id);
END
```

注意： IQ でリモート・プロシージャを作成するには、**CREATE PROCEDURE** の AT *location-string* SQL 構文を使用して、プロキシ・ストアド・プロシージャを作成します。この機能は、現時点では Windows と Sun Solaris でのみ動作確認されています。Sybase Central のリモート・プロシージャ作成ウィザードは、リモート サーバでのみ使用できます。

プロシージャの変更

Sybase Central または Interactive SQL のいずれかを使用して、既存のプロシージャを修正できます。それには、DBA 権限を持っているか、またはプロシージャの所有者でなくてはなりません。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - SQL の使用法』>「ストアド・プロシージャとトリガ」>「プロシージャ、トリガ、バッチの使用」>「プロシージャの概要」>「プロシージャの変更」を参照してください。

データベース・オブジェクト・プロパティの変更については、「Sybase IQ の概要」>「データベースの管理」>「プロシージャの管理」を参照してください。

プロシージャに対するパーミッションの付与または取り消しについては、『システム管理ガイド：第 1 巻』の「ユーザ ID とパーミッションの管理」>「個別のユーザ ID とパーミッションの管理」>「Interactive SQL でプロシージャに対するパーミッションを付与する」、および『システム管理ガイド：第 1 巻』の「ユーザ ID とパーミッションの管理」>「個別のユーザ ID とパーミッションの管理」>「Interactive SQL でユーザ・パーミッションを取り消す」を参照してください。

ALTER PROCEDURE 文を使用して、プロシージャを変更することもできます。

プロシージャの呼び出し

プロシージャの呼び出しには **CALL** 文を使用します。プロシージャは、アプリケーション・プログラムから呼び出すことも、他のプロシージャから呼び出すこともできます。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - SQL の使用法』>「ストアド・プロシージャとトリガ」>「プロシージャ、トリガ、バッチの使用」>「プロシージャの概要」>「プロシージャの呼び出し」を参照してください。

参照：

- *プロシージャを実行するためのパーミッション*(6 ページ)

Sybase Central でのプロシージャのコピー

プロシージャのコードは、あるデータベースから別の接続されたデータベースへコピーできます。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - SQL の使用法』>「ストアド・プロシージャとトリガ」>「プロシージャ、トリガ、バッチの使用」>「プロシージャの概要」>「Sybase Central におけるプロシージャのコピー」を参照してください。

プロシージャの削除

いったん作成したプロシージャは、誰かが明示的に削除するまではデータベースに残っています。プロシージャの所有者か DBA 権限を持つユーザだけがプロシージャをデータベースから削除できます。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - SQL の使用法』>「ストアド・プロシージャとトリガ」>「プロシージャ、トリガ、バッチの使用」>「プロシージャの概要」>「プロシージャの削除」を参照してください。

プロシージャを実行するためのパーミッション

プロシージャの所有者はそれを作成したユーザです。所有者はパーミッションなしでそのプロシージャを実行できます。

プロシージャを実行するパーミッションを他のユーザに付与するには、**GRANT EXECUTE** コマンドを使用します。 たとえば、プロシージャ **new_dept** の所有者は、次の文を使用して **new_dept** の実行パーミッションを **another_user** に付与できます。

```
GRANT EXECUTE ON new_dept TO another_user
```

パーミッションを取り消す文は、次のようになります。

```
REVOKE EXECUTE ON new_dept FROM another_user
```

『システム管理ガイド：第 1 巻』の「ユーザ ID とパーミッションの管理」>「個別のユーザ ID とパーミッションの管理」>「Interactive SQL でプロシージャに対するパーミッションを付与する」を参照してください。

参照：

- *プロシージャの呼び出し* (5 ページ)

プロシージャの結果をパラメータとして返す

プロシージャは、呼び出し元の環境に結果を返します。

プロシージャは、次のいずれかの方法で結果を返します。

- OUT または INOUT パラメータとして個々の値を返す。
- 結果セットを返す。
- **RETURN** 文を使用して結果を 1 つ返す。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - SQL の使用法』>「ストアド・プロシージャとトリガ」>「プロシージャ、トリガ、バッチの使用」>「プロシージャの概要」>「プロシージャの結果をパラメータとして返す」を参照してください。

Sybase IQ の例

注意： たとえば、Sybase IQ デモ・データベース `iqdemo.db` を使用します。

```
CREATE PROCEDURE SalaryList (IN department_id INT)
RESULT ( "Employee ID" INT, "Salary" NUMERIC(20,3) )
BEGIN
            SELECT EmployeeID, Salary
            FROM Employees
            WHERE Employees.DepartmentID = department_id;
END
```

プロシージャの結果を結果セットとして返す

プロシージャは、個別のパラメータとして呼び出しを行った環境に結果を返すだけでなく、結果セットとして情報を返すこともできます。通常、結果セットになるのはクエリの結果です。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - SQL の使用法』>「ストアド・プロシージャとトリガ」>「プロシージャ、トリガ、バッチの使用」>「プロシージャの概要」>「プロシージャの結果を結果セットとして返す」を参照してください。

テンポラリ・テーブルの作成とテーブルからの選択
ストアド・プロシージャ内でテンポラリ・テーブルを動的に作成した後、そのテーブルに対する SELECT 文を実行する場合は、**EXECUTE IMMEDIATE WITH RESULT SET ON** 構文を使用して、`Column not found` というエラーが起きないようにします。

次に例を示します。

```
CREATE PROCEDURE p1 (IN @t varchar(30))
   BEGIN
      EXECUTE IMMEDIATE
      'SELECT * INTO #resultSet FROM ' || @t;
      EXECUTE IMMEDIATE WITH RESULT SET ON
      'SELECT * FROM #resultSet';   END
```

ユーザ定義関数入門

ユーザ定義関数は、呼び出しを行った環境に単一の値を返すプロシージャのクラスです。ここでは、ユーザ定義関数の作成、使用、削除について説明します。

ユーザ定義関数の作成

ユーザ定義関数を作成するには、**CREATE FUNCTION** 文を使用します。ただし、RESOURCE 権限を持っていなくてはなりません。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - SQL の使用法』>「ストアド・プロシージャとトリガ」>「プロシージャ、トリガ、バッチの使用」>「ユーザ定義関数の概要」>「ユーザ定義関数の作成」を参照してください。

パフォーマンスの考慮事項や SQL Anywhere と IQ の違いなど、**CREATE FUNCTION** 構文の詳細については、『リファレンス：文とオプション』の「SQL 文」>「CREATE FUNCTION 文」を参照してください。

ユーザ定義関数の呼び出し

ユーザ定義関数は、パーミッションがあれば、集合関数以外の組み込み関数を使用できるどの場所でも使用できます。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - SQL の使用法』>「ストアド・プロシージャとトリガ」>「プロシージャ、トリガ、バッチの使用」>「ユーザ定義関数の概要」>「ユーザ定義関数の呼び出し」を参照してください。

Sybase IQ の例

注意： たとえば、Sybase IQ デモ・データベース `iqdemo.db` を使用します。

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

fullname (GivenName, SurName)

Fran Whitney Matthew Cobb Philip Chin...

ユーザ定義関数の削除

ユーザ定義関数が作成されると、明示的に削除されるまでデータベースに存在します。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - SQL の使用法』>「ストアド・プロシージャとトリガ」>「プロシージャ、トリガ、バッチの使用」>「ユーザ定義関数の概要」>「ユーザ定義関数の削除」を参照してください。

ユーザ定義関数を実行するためのパーミッション

ユーザ定義関数の所有者はそれを作成したユーザです。所有者はパーミッションなしでそれを実行できます。

ユーザ定義関数の所有者は、**GRANT EXECUTE** コマンドを使用して、他のユーザにパーミッションを付与できます。

たとえば、関数 `fullname` の作成者は、次の文を使用して `fullname` を使用するパーミッションを `another_user` に付与できます。

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

パーミッションを取り消す文は、次のようになります。

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

『システム管理ガイド：第 1 巻』の「ユーザ ID とパーミッションの管理」>「個別のユーザ ID とパーミッションの管理」>「Interactive SQL でプロシージャに対するパーミッションを付与する」を参照してください。

バッチ入門

簡単なバッチは、セミコロン (;) で区切られた SQL 文のセットから構成されます。

たとえば、次の一連の文はバッチを構成します。このバッチは、Eastern Sales という部署を作成し、マサチューセッツ (MA) のすべての営業担当者をこの部署に移動します。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - SQL の使用法』>「ストアド・プロシージャとトリガ」>「プロシージャ、トリガ、バッチの使用」>「バッチの概要」を参照してください。

Sybase IQ の例

注意： たとえば、Sybase IQ デモ・データベース `iqdemo.db` を使用します。

```
INSERT
INTO Departments ( DepartmentID, DepartmentName )
VALUES ( 220, 'Eastern Sales' ) ;
UPDATE Employees

SET DepartmentID = 220
WHERE DepartmentID = 200
AND state = 'GA' ;

COMMIT ;
```

制御文

プロシージャの本文またはバッチの中には、論理フローや意思決定のための制御文が多く含まれています。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - SQL の使用法』>「ストアド・プロシージャとトリガ」>「プロシージャ、トリガ、バッチの使用」>「制御文」を参照してください。

各文の詳細については、『リファレンス：文とオプション』の「SQL 文」を参照してください。

複合文の使用

複合文はネストが可能です。また、他の制御文と組み合わせて、プロシージャ内またはバッチ内の実行フローを定義できます。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - SQL の使用法』>「ストアド・プロシージャとトリガ」>「プロシージャ、トリガ、バッチの使用」>「制御文」>「複合文の使用」を参照してください。

参照：

- *プロシージャで使用可能な SQL 文* (11 ページ)
- *プロシージャの構造* (11 ページ)
- *プロシージャでのトランザクションとセーブポイント* (17 ページ)

複合文での宣言

複合文中のローカル宣言文は、キーワード **BEGIN** のすぐ後に続きます。このローカル宣言は複合文中だけに存在します。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - SQL の使用法』>「ストアド・プロシージャとトリガ」>「プロシージャ、トリガ、バッチの使用」>「制御文」>「複合文での宣言」を参照してください。

アトミックな複合文

「アトミック」な文とは、完全に実行されたか、まったく実行されなかったかのいずれかの文です。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - SQL の使用法』>「ストアド・プロシージャとトリガ」>「プロシージャ、トリガ、バッチの使用」>「制御文」>「アトミックな複合文」を参照してください。

プロシージャの構造

プロシージャの本文は複合文で構成されます。

複合文は、一連の SQL 文を囲む **BEGIN** と **END** で構成されています。各文はセミコロンで区切られています。

参照：
- *プロシージャで使用可能な SQL 文* (11 ページ)
- *プロシージャでのトランザクションとセーブポイント* (17 ページ)
- *複合文の使用* (10 ページ)

プロシージャで使用可能な SQL 文

プロシージャでは、次に示すように、ほぼすべての SQL 文を使用できます。

- **SELECT**、**UPDATE**、**DELETE**、**INSERT**、**SET VARIABLE**
- 他のプロシージャを実行する **CALL** 文
- 制御文
- CURSOR 文
- 例外処理文
- **EXECUTE IMMEDIATE** 文

一部の SQL 文はプロシージャ内で使用できません。たとえば次のものです。

- **CONNECT** 文
- **DISCONNECT** 文

プロシージャ内で **COMMIT** 文、**ROLLBACK** 文、**SAVEPOINT** 文を使用できますが、特定の制限が適用されます。

『リファレンス：文とオプション』の「SQL 文」にある各文の「使用法」を参照してください。

参照：
- *プロシージャの構造* (11 ページ)
- *プロシージャでのトランザクションとセーブポイント* (17 ページ)
- *複合文の使用* (10 ページ)

プロシージャ・パラメータの宣言

プロシージャ・パラメータは、**CREATE PROCEDURE** 文でリストとして記述します。

パラメータ名は、カラム名など他のデータベース識別子に関するルールに従って付けてください。 パラメータには有効なデータ型を指定します。また、`IN`、`OUT`、または `INOUT` のいずれかのキーワードを先頭に指定してください。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - SQL の使用法』>「ストアド・プロシージャとトリガ」>「プロシージャ、トリガ、バッチの使用」>「プロシージャとトリガの構造」>「プロシージャ・パラメータの宣言」を参照してください。

パラメータをプロシージャに渡す

ストアド・プロシージャ・パラメータのデフォルト値は、**CALL** 文の 2 種類の形式のどちらでも使用できます。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - SQL の使用法』>「ストアド・プロシージャとトリガ」>「プロシージャ、トリガ、バッチの使用」>「プロシージャとトリガの構造」>「パラメータをプロシージャに渡す」を参照してください。

パラメータを関数に渡す

ユーザ定義関数 (UDF) は、**CALL** 文で呼び出すのではなく、組み込み関数と同じように使用します。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - SQL の使用法』>「ストアド・プロシージャとトリガ」>「プロシージャ、トリガ、バッチの使用」>「プロシージャとトリガの構造」>「パラメータを関数に渡す」を参照してください。

プロシージャの結果

プロシージャは、結果を 1 つまたは複数のデータ・ローとして返します。

単一のローのデータからなる結果の場合は、プロシージャへの引数で返すことができます。複数のローのデータからなる結果の場合は、結果セットで返します。また、プロシージャは **RETURN** 文で 1 つの値を返すこともできます。

プロシージャから結果を返す簡単な例については、次の項を参照してください。詳細については、次の項を参照してください。

参照：

- *プロシージャ入門*(4 ページ)

RETURN 文を使って値を返す

RETURN 文は、呼び出し元の環境に単一の整数値を返し、プロシージャを即座に終了します。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - SQL の使用法』>「ストアド・プロシージャとトリガ」>「プロシージャ、トリガ、バッチの使用」>「プロシージャから返される結果」>「RETURN 文を使って値を返す」を参照してください。

結果をプロシージャのパラメータとして返す

プロシージャは、プロシージャのパラメータで呼び出しを実行した環境に結果を返すことができます。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - SQL の使用法』>「ストアド・プロシージャとトリガ」>「プロシージャ、トリガ、バッチの使用」>「プロシージャから返される結果」>「結果をプロシージャのパラメータとして返す」を参照してください。

プロシージャから結果セットを返す

結果セットを使うと、プロシージャから複数ローの結果を呼び出し元の環境に返すことができます。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - SQL の使用法』>「ストアド・プロシージャとトリガ」>「プロシージャ、トリガ、バッチの使用」>「プロシージャから返される結果」>「プロシージャから結果セットを返す」を参照してください。

プロシージャから複数の結果セットを返す

プロシージャから呼び出元の環境に、複数の結果セットを返すことができます。

複数の結果セットを返すための方法は、**dbisql** と **dbisqlc** では異なります。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - SQL の使用法』>「ストアド・プロシージャとトリガ」>「プロシージャ、トリガ、バッチの使用」>「プロシージャから返される結果」>「プロシージャから複数の結果セットを返す」を参照してください。

プロシージャから変数結果セットを返す

プロシージャで RESULT 句を省略することもできます。RESULT 句を省略すると、実行方法に応じてさまざまなカラム数やカラム型を使った異なる結果セットを返すプロシージャを記述できます。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - SQL の使用法』>「ストアド・プロシージャとトリガ」>「プロシージャ、トリガ、バッチの使用」>「プロシージャから返される結果」>「プロシージャから変数結果セットを返す」を参照してください。

プロシージャでのカーソル

カーソルは、結果セットに複数のローがあるクエリやストアド・プロシージャからローを 1 つずつ取り出します。

「カーソル」は、クエリまたはプロシージャに対するハンドルまたは識別子で、結果セットの中の現在の位置を示します。

カーソル管理の概要

カーソル管理はプログラミング言語のファイル管理に似ています。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - SQL の使用法』>「ストアド・プロシージャとトリガ」>「プロシージャ、トリガ、バッチの使用」>「プロシージャとトリガでのカーソルの使用」>「カーソル管理の概要」を参照してください。

sp_iqcursorinfo ストアド・プロシージャは、サーバ上で現在開いているカーソルに関する情報を表示します。 詳細については、『リファレンス：ビルディング・ブロック、テーブル、およびプロシージャ』の「システム・プロシージャ」>「sp_iqcursorinfo プロシージャ」を参照してください。

カーソル位置

カーソルの位置は、非常に柔軟に指定できます。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - プログラミング』>「SQL Anywhere でのプログラミングの概要」>「アプリケーションでの SQL の使用」>「カーソルを使用した操作」>「カーソル位置」を参照してください。

注意： Sybase IQ は、FIRST、LAST、ABSOLUTE の各オプションを結果セットの先頭から開始するものとして扱います。 負数のロー・カウントの RELATIVE は、現在の位置から開始するものとして扱います。

プロシージャでのカーソルと SELECT 文

`TopCustomerValue` プロシージャは、**SELECT** 文でカーソルを使用します。これは、`ListCustomerValue` プロシージャで使用する同じクエリに基づいて行われます。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - SQL の使用法』>「ストアド・プロシージャとトリガ」>「プロシージャ、トリガ、バッチの使用」>「プロシージャとトリガでのカーソルの使用」>「プロシージャの SELECT 文でのカーソルの使用」を参照してください。

プロシージャでのエラーと警告

アプリケーション・プログラムでは、SQL 文を実行した後、*「リターン・コード」* (ステータス・コード) でエラーをチェックできます。

リターン・コードは文が正しく実行されたかどうかを表示して、エラーの場合はその理由を提示します。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - SQL の使用法』>「ストアド・プロシージャとトリガ」>「プロシージャ、トリガ、バッチの使用」>「プロシージャとトリガでのエラーと警告」を参照してください。

注意： Sybase IQ ではトリガをサポートしていません。 SQL Anywhere マニュアルのトリガについての情報は無視してかまいません。

プロシージャでのデフォルトのエラー処理

Sybase IQ は、プロシージャの実行中に発生したエラーを処理します。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - SQL の使用法』>「ストアド・プロシージャとトリガ」>「プロシージャ、トリガ、バッチの使用」>「プロシージャとトリガでのエラーと警告」>「プロシージャとトリガでのデフォルトのエラー処理」を参照してください。

注意： Sybase IQ ではトリガをサポートしていません。 SQL Anywhere マニュアルのトリガについての情報は無視してかまいません。

ON EXCEPTION RESUME を使用したエラー処理

ON EXCEPTION RESUME 句は、**CREATE PROCEDURE** 文で使用します。

エラーが発生したときに、このプロシージャは文をチェックします。文がエラーを処理する場合は、エラーが起きても制御が呼び出しを行った環境に戻りません。エラーを起こした文の次の文から実行を再開します。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - SQL の使用法』>「ストアド・プロシージャとトリガ」>「プロシージャ、トリガ、バッチの使用」>「プロシージャとトリガでのエラーと警告」>「ON EXCEPTION RESUME を使ったエラー処理」を参照してください。

プロシージャでのデフォルトのエラー処理と警告処理

エラーと警告は、プロシージャでの処理方法が異なります。

デフォルトのエラー処理では、エラーが発生した場合、*SQLSTATE* 変数と *SQLCODE* 変数に値が設定され、呼び出し元の環境に制御が戻されます。一方、デフォルトの警告処理では、*SQLSTATE* と *SQLCODE* に値が設定され、プロシージャの実行が継続されます。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - SQL の使用法』>「ストアド・プロシージャとトリガ」>「プロシージャ、トリガ、バッチの使用」>「プロシージャとトリガでのエラーと警告」>「プロシージャとトリガでのデフォルトのエラー処理」と「プロシージャとトリガでのデフォルトの警告処理」を参照してください。

注意： Sybase IQ ではトリガをサポートしていません。 SQL Anywhere マニュアルのトリガについての情報は無視してかまいません。

プロシージャでの例外ハンドラの使用

特定のタイプのエラーは、呼び出し元の環境に戻すのではなく、プロシージャ内で検知して処理できます。それには「*例外ハンドラ*」を使用します。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - SQL の使用法』>「ストアド・プロシージャとトリガ」>「プロシージャ、トリガ、バッチの使用」>「プロシージャとトリガでのエラーと警告」>「プロシージャとトリガでの例外ハンドラの使用」を参照してください。

注意： Sybase IQ ではトリガをサポートしていません。 SQL Anywhere マニュアルのトリガについての情報は無視してかまいません。

ネストされた複合文と例外ハンドラ

ネストされた複合文を使用すると、エラーの後に実行する文と実行しない文を制御しやすくなります。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - SQL の使用法』>「ストアド・プロシージャとトリガ」>「プロシージャ、トリガ、バッチの使用」>「プロシージャとトリガでのエラーと警告」>「ネストされた複合文と例外処理」を参照してください。

プロシージャでの EXECUTE IMMEDIATE 文の使用

EXECUTE IMMEDIATE 文を使用すると、引用符で囲んだリテラル文字列と変数を使用してプロシージャ内に文を組み立てることができます。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - SQL の使用法』>「ストアド・プロシージャとトリガ」>「プロシージャ、トリガ、バッチの使用」>「プロシージャでの EXECUTE IMMEDIATE 文の使用」を参照してください。

プロシージャでのトランザクションとセーブポイント

プロシージャまたはトリガ内の SQL 文は、現在のトランザクションの一部です。

1 つのトランザクション中で複数のプロシージャを呼び出したり、1 つのプロシージャ中に複数のトランザクションを保持したりできます。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - SQL の使用法』>「ストアド・プロシージャとトリガ」>「プロシージャ、トリガ、バッチの使用」>「プロシージャとトリガでのトランザクションとセーブポイント」を参照してください。

注意： Sybase IQ ではトリガをサポートしていません。 SQL Anywhere マニュアルのトリガについての情報は無視してかまいません。

詳細については、『システム管理ガイド：第 1 巻』の「トランザクションとバージョン管理」>「トランザクション内のセーブポイント」を参照してください。

参照：

- *プロシージャで使用可能な SQL 文* (11 ページ)
- *プロシージャの構造* (11 ページ)
- *複合文の使用* (10 ページ)

プロシージャ、関数、ビューの内容を隠す

場合によっては、アプリケーションとデータベースを配布するときに、プロシージャ、関数、トリガ、ビューの中身のロジックを隠す方がよい可能性があります。

ALTER PROCEDURE、**ALTER FUNCTION**、**ALTER VIEW** 文では、セキュリティ強化策の一環として、SET HIDDEN 句を使用してこれらのオブジェクトの内容を隠すことができます。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - SQL の使用法』>「ストアド・プロシージャとトリガ」>「プロシージャ、トリガ、バッチの使用」>「プロシージャ、関数、トリガ、ビューの内容を隠す」を参照してください。

注意： Sybase IQ ではトリガをサポートしていません。 SQL Anywhere マニュアルのトリガについての情報は無視してかまいません。

詳細については、『リファレンス：文とオプション』の `ALTER FUNCTION` 文、`ALTER PROCEDURE` 文、`ALTER VIEW` 文に関する説明を参照してください。

バッチで使用できる文

バッチでは大部分の SQL 文を使用できますが、一部例外があります。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - SQL の使用法』>「ストアド・プロシージャとトリガ」>「プロシージャ、トリガ、バッチの使用」>「プロシージャ、トリガ、イベント、バッチで使用できる文」を参照してください。

注意： Sybase IQ ではトリガをサポートしていません。 SQL Anywhere マニュアルのトリガについての情報は無視してかまいません。

バッチでの SELECT 文の使用

バッチでは、1 つ以上の **SELECT** 文を使用できます。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - SQL の使用法』>「ストアド・プロシージャとトリガ」>「プロシージャ、トリガ、バッチの使用」>「プロシージャ、トリガ、イベント、バッチで使用できる文」>「バッチでの SELECT 文の使用」を参照してください。

Sybase IQ の例

注意： たとえば、Sybase IQ デモ・データベース `iqdemo.db` を使用します。

```
IF EXISTS(
            SELECT * FROM SYSTAB
                        WHERE table_name='Employees' )
THEN
            SELECT Surname AS LastName,
                        GivenName AS FirstName
            FROM Employees;
            SELECT Surname, GivenName
            FROM Customers;
            SELECT Surname, GivenName
            FROM Contacts;
ELSE
            MESSAGE 'The Employees table does not exist'
```

```
            TO CLIENT;
END IF
```

IQ UTILITIES を使用した独自のストアド・プロシージャの作成

Sybase IQ に用意されているシステム・ストアド・プロシージャは、この章で説明する手法を使用して、SQL で実装されています。

ローカル・テンポラリ・テーブルと **IQ UTILITIES** 文は、システム・ストアド・プロシージャとまったく同じように使用します。

警告！ この規則に違反すると、IQ サーバやデータベースに重大な問題が生じる可能性があります。

プロシージャのすべての SQL コードは暗号化され、共有ライブラリ (UNIX では `libiqscripts15_r.so` ファイル、Windows では `iqscripts15.dll` ファイル) に集約されます。

Sybase Central を使用するか、Interactive SQL で **sp_helptext 'owner.procname'** を入力することで、ストアド・プロシージャ・コードを確認できます。

IQ UTILITIES の構文は、次のとおりです。

```
IQ UTILITIES
MAIN
INTO
local-temp-table-name
 arguments
```

IQ モニタへの **IQ UTILITIES** コマンドは、『リファレンス：文とオプション』にのみ掲載されています。使用に厳密な規則があることと、不正に使用した場合にシステムの処理にリスクを及ぼすことがその理由です。

これらのプロシージャのいくつかから派生した独自のプロシージャを作成できます。それには次のような方法があります。

1. システム・ストアド・プロシージャを呼び出すプロシージャを作成します。
2. システム・ストアド・プロシージャとは別個に、同様の機能を実行するプロシージャを作成します。
3. システム・ストアド・プロシージャと同じ構造を使用し、かつ独自の機能を追加したプロシージャを作成します。 たとえば、フロントエンド・ツールやブラウザで、テキストではなくグラフィカル形式でプロシージャの結果を表示するなどです。

4. 2つ目や3つ目の方法を使用する場合は、**IQ UTILITIES** 文と、その使用方法の厳密な規則を理解する必要があります。

IQ による IQ UTILITIES コマンドの使用

IQ UTILITIES は、ほとんどの IQ のシステム・プロシージャで、実行時に内部で動作する文です。多くの場合、**IQ UTILITIES** が動作していることは、ユーザの関知するところではありません。**IQ UTILITIES** をユーザが直接実行するのは、IQ バッファ・キャッシュ・モニタを起動するときのみです。

IQ UTILITIES を使用することで、IQ のシステム・テーブルに保持されている情報を体系的に収集およびレポートできます。一般的なユーザ・インタフェースはありません。既存のシステム・プロシージャと同様に、**IQ UTILITIES** のみで使用できます。

システム・プロシージャは、情報を格納するローカル・テンポラリ・テーブルを宣言します。システム・プロシージャは **IQ UTILITIES** を実行してシステム・テーブルから情報を取得し、それをローカル・テンポラリ・テーブルに格納します。システム・プロシージャは、ローカル・テンポラリ・テーブルから情報をレポートするだけのこともあれば、追加的な処理を実行することもあります。

システム・プロシージャの中には、事前に定義された番号を **IQ UTILITIES** 文の引数の1つとして指定するものもあります。この番号に応じて特定の機能が実行されます。たとえば、システム・テーブルの情報から値を得るなど。**IQ UTILITIES** の引数として使用する番号のリストについては、次の項を参照してください。

呼び出すプロシージャの選択

データベースの情報をレポートする、作成済みのシステム・プロシージャについては、**IQ UTILITIES** を使用して独自バージョンを作成するのが安全です。

たとえば、**sp_iqspaceused** は、IQ メイン・ストアと IQ テンポラリ・ストアの使用済み領域と空き領域についての情報を表示します。作成したプロシージャの所有者をシステム・ストアド・プロシージャに照らし、正しい所有者となっていることを確認してください。

IQ の処理を制御するシステム・プロシージャについては、独自バージョンを作成しないでください。IQ の処理を制御するプロシージャを修正すると、重大な問題につながる可能性があります。

IQ UTILITIES で使用される番号

次の表では、**IQ UTILITIES** コマンドで引数として使用される番号と、各番号を使用するシステム・プロシージャをリストしています。

これらのプロシージャの機能については、『リファレンス：ビルディング・ブロック、テーブル、およびプロシージャ』の「システム・プロシージャ」を参照してください。

表 1 : システム・プロシージャで使用される IQ UTILITIES の番号

番号	プロシージャ	コメント
10000	sp_iqtransaction	
20000	sp_iqconnection、sp_iqmpxcountdbremote	
30000	sp_iqspaceused	
40000	sp_iqspaceinfo	
50000	sp_iqlocks	
60000	sp_iqmpxversionfetch	使用できません。
70000	sp_iqmpxdumptlvlog	
80000	sp_iqcontext	
100000	sp_iqindexfragmentation	
110000	sp_iqrowdensity	

プロシージャのテスト

作成したプロシージャのテストは、必ず開発環境から始めてください。 運用環境で実行する前にプロシージャをテストすることで、IQ サーバとデータベースの安定性を確保できます。

OLAP の使用

オンライン分析処理 (OLAP: Online Analytical Processing) は、リレーショナル・データベースに格納されている情報を効率的にデータ分析するための手法です。

OLAP を使用すると、データをさまざまな次元で分析し、小計ローを含んだ結果セットを取得し、データを多次元キューブに編成するという処理をすべて 1 つの SQL クエリで行うことができます。また、フィルタを使用してデータを絞り込み、結果セットを迅速に返すことができます。 この章では、Sybase IQ がサポートする SQL/OLAP 関数について説明します。

注意： OLAP の例に示されているテーブルは、`iqdemo` データベースにあります。

OLAP について

この分析関数を使用すると、複雑なデータ分析を 1 つの SQL 文で実行できます。これは、オンライン分析処理 (OLAP) と呼ばれるソフトウェア・テクノロジ分類に基づいています。OLAP の関数には、次のようなものが含まれています。

- **GROUP BY** 句の拡張 - **CUBE** と **ROLLUP**
- 分析関数：
  - 単純な集合関数 - **AVG**、**COUNT**、**MAX**、 **MIN**、**SUM**、**STDDEV**、**VARIANCE**

    注意： **Grouping()** 以外の単純な集合関数は OLAP ウィンドウ関数と併用できます。
  - ウィンドウ関数
    - ウィンドウ集合関数 - **AVG**、**COUNT**、**MAX**、 **MIN**、**SUM**
    - ランク付け関数 - **RANK**、**DENSE_RANK**、**PERCENT_RANK**、**NTILE**
    - 統計関数 - **STDDEV**、**STDDEV_SAMP**、**STDDEV_POP**、**VARIANCE**、**VAR_POP**、 **VAR_SAMP**、 **REGR_AVGX**、**REGR_AVGY**、**REGR_COUNT**、**REGR_INTERCEPT**、**REGR_R2**、**REGR_SLOPE**、**REGR_SXX**、**REGR_SXY**、**REGR_SYY**、**CORR**、**COVAR_POP**、**COVAR_SAMP**、**CUME_DIST**、**EXP_WEIGHTED_AVG**、**WEIGHTED_AVG**
    - 分散統計関数 - **PERCENTILE_CONT** と **PERCENTILE_DISC**
  - 数値関数 - **WIDTH_BUCKET**、**CEIL**、**LN**、**EXP**、**POWER**、**SQRT**、**FLOOR**

1999 年の SQL 標準の改正で、ANSI SQL 標準に複雑なデータ分析機能を含めるための拡張が導入されました。 Sybase IQ では、これらの SQL 拡張機能の一部が取り入れられており、これらの拡張を包括的に追加サポートしています。

データベース製品によっては、OLAP モジュールが独立しており、分析前にデータをデータベースから OLAP モジュールに移動しなければならないものもあります。 一方、Sybase IQ では OLAP 機能がデータベースそのものに組み込まれているため、ストアド・プロシージャなど他のデータベース機能との配備や統合を簡単かつシームレスに行うことができます。

OLAP の利点

OLAP 関数を **GROUPING**、**CUBE**、**ROLLUP** という拡張機能と組み合わせて使用すると、2 つの大きな利点があります。

第一に、多次元のデータ分析、データ・マイニング、時系列分析、傾向分析、コストの割り当て、ゴール・シーク、一時的な多次元構造変更、非手続き型モデリング、例外の警告を多くの場合 1 つの SQL 文で実行できます。第二に、OLAP のウィンドウおよびレポート集合関数では、*ウィンドウ*という関係演算子を使用することができ、これはセルフジョインや相関サブクエリを使用するセマンティック的に等価なクエリよりも効率的に実行できます。OLAP を使用して取得した結果セットには小計ローを含めることができ、この結果セットを多次元キューブに編成することもできます。 詳細については、次の項を参照してください。

さまざまな期間での移動平均と移動和を計算したり、選択したカラムの値が変化したときに集計とランクをリセットしたり、複雑な比率を単純な言葉で表現したりできます。1 つのクエリ式のスコープ内で、それぞれ独自の分割ルールを持ついくつかの異なる OLAP 関数を定義することができます。

参照：

- *分散統計関数* (64 ページ)
- *OLAP の評価* (24 ページ)
- *ランク付け関数* (53 ページ)
- *統計集合関数* (59 ページ)
- *ウィンドウ* (40 ページ)
- *ウィンドウ集合関数* (57 ページ)
- *OLAP 関数の BNF 文法* (79 ページ)

OLAP の評価

OLAP の評価は、最終的な結果に影響を及ぼすクエリ実行のいくつかのフェーズとして概念化できます。

OLAP の実行フェーズは、クエリ内の対応する句によって識別されます。 たとえば、SQL クエリの指定にウィンドウ関数が含まれている場合は、**WHERE**、**JOIN**、**GROUP BY**、**HAVING** 句が先に処理されます。 GROUP BY 句でグループが定義され

た後、クエリの **ORDER BY** 句に含まれる最後の **SELECT** リストが評価される前に、パーティションが作成されます。

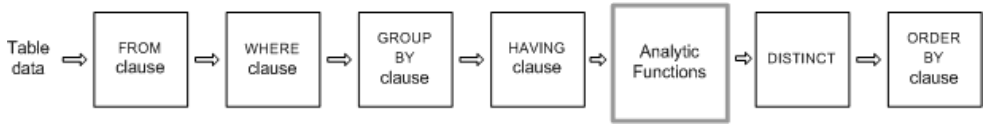
グループ化の際には、NULL 値はすべて同じグループと見なされます (それぞれの NULL 値が等しくない場合でも同様です)。

HAVING 句は、**WHERE** 句に類似しており、GROUP BY 句の結果に対するフィルタとして機能します。

ANSI SQL 標準に基づく SQL 文と **SELECT**、**FROM**、**WHERE**、**GROUP BY**、**HAVING** 句を含んだ単純なクエリ仕様のセマンティックを考えてみます。

1. クエリにより、**FROM** 句のテーブル式を満たすロー・セットが取得されます。
2. **WHERE** 句の述部が、テーブルから取得したローに適用されます。 **WHERE** 句の条件を満たさない (条件が true にならない) ローが除外されます。
3. 残りの各ローについて、**SELECT** リストおよび GROUP BY 句に含まれている式 (集合関数を除く) が評価されます。
4. GROUP BY 句の式の重複しない値に基づいて、結果のローがグループ化されます (NULL は各ドメインで特殊な値として扱われます)。 **PARTITION BY** 句がある場合、GROUP BY 句の式はパーティション・キーとして使用されます。
5. 各パーティションについて、**SELECT** リストまたは **HAVING** 句の集合関数が評価されます。いったん集合関数を適用すると、中間の結果セットには個々のテーブル・ローが含まれなくなります。 新しい結果セットには、**GROUP BY** の式と、各パーティションについて計算した集合関数の値が含まれます。
6. **HAVING** 句の条件が結果グループに適用されます。 **HAVING** 句の条件を満たさないグループが除外されます。
7. **PARTITION BY** 句で定義された境界に基づいて結果が分割されます。 結果ウィンドウについて、OLAP ウィンドウ関数 (ランク付け関数および集合関数) が計算されます。

図 1 : OLAP の SQL 処理



詳細については、次の項を参照してください。 次の項目も参照してください。

参照：

- *分散統計関数* (64 ページ)
- *OLAP の利点* (24 ページ)
- *ランク付け関数* (53 ページ)

- *統計集合関数* (59 ページ)
- *ウィンドウ* (40 ページ)
- *ウィンドウ集合関数* (57 ページ)
- *OLAP 関数の BNF 文法* (79 ページ)

GROUP BY 句の拡張

GROUP BY 句を拡張することで、次のような処理を行う複雑な SQL 文を記述できます。

- 入力ローを複数の次元に分割し、結果グループの複数のサブセットを組み合わせる。
- 「データ・キューブ」を作成し、データ・マイニング分析のための疎密度の多次元結果セットを用意する。
- 元のグループを含んだ結果セットを作成する (必要に応じて、小計ローと合計ローを含める場合もある)。

ROLLUP や **CUBE** など、OLAP の Grouping() 操作は、プレフィクスや小計ローとして概念化できます。

プレフィクス

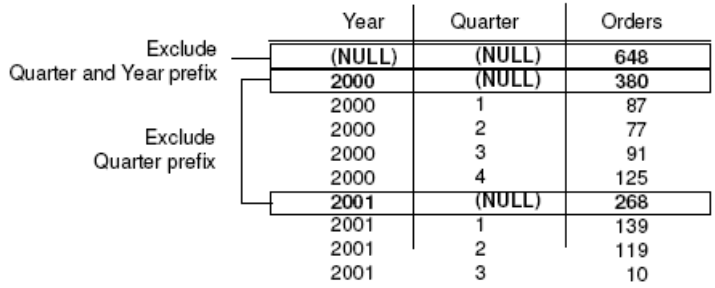
GROUP BY 句を含むクエリでは、*「プレフィクス」*のリストが作成されます。 プレフィクスとは、**GROUP BY** 句の項目のサブセットであり、クエリの **GROUP BY** 句の項目のうち最も右にある 1 つ以上の項目を除外することで作成されます。残りのカラムは*プレフィクス・カラム*と呼ばれます。

ROLLUP の例 1 - 次の **ROLLUP** のクエリ例では、**GROUP BY** リストに 2 つの変数 (*Year* と *Quarter*) が含まれています。

```
SELECT year (OrderDate) AS Year, quarter(OrderDate)
  AS Quarter, COUNT(*) Orders
FROM SalesOrders
GROUP BY ROLLUP(Year, Quarter)
ORDER BY Year, Quarter
```

このクエリには次の 2 つのプレフィクスがあります。

- Quarter を除外するプレフィクス ― 一連のプレフィクス・カラムには 1 つのカラム (Year) が含まれます。
- Quarter と Year の両方を除外するプレフィクス ― プレフィクス・カラムは存在しません。



注意： **GROUP BY** リストには、項目と同じ数のプレフィクスが含まれます。

GROUP BY での ROLLUP と CUBE

ROLLUP と **CUBE** は、一般的なグループ化プレフィクスを指定する構文簡略化パターンです。

GROUP BY ROLLUP

ROLLUP 演算子では、グループ化式が順に並べられたリストを引数として指定する必要があります。

ROLLUP 構文。

```
            SELECT … [ GROUPING (column-name) … ] …
GROUP BY [ expression [, …]
| ROLLUP ( expression [, …] ) ]
```

GROUPING は、カラム名をパラメータとして受け取り、次の表に示すようにブール値を返します。

表 2 : ROLLUP 演算子が指定された GROUPING によって返される値

結果値の種類	GROUPING の戻り値
ROLLUP 処理によって作成された NULL	1 (真)
ローが小計であることを示す NULL	1 (真)
ROLLUP 処理によって作成されたもの以外の NULL	0 (偽)
格納されていた NULL	0 (偽)

ROLLUP は、**GROUP BY** 句に指定された標準の集合値を最初に計算します。 次に、**ROLLUP** はグループ化を行うカラムのリストを右から左に移動し、より高いレベルの小計を連続して作成します。最後に総計が作成されます。 グループ化するカラムの数が *n* 個の場合、**ROLLUP** は *n* に 1 を加えたレベルの小計を作成します。

SQL 構文の例	定義されるセット
`GROUP BY ROLLUP (A, B, C);`	(A, B, C) (A, B) (A) ( )

ROLLUP と小計ロー

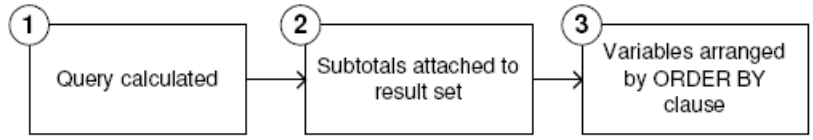
ROLLUP は、一連の **GROUP BY** クエリの **UNION** に等しくなります。次の 2 つのクエリの結果セットは等しくなります。 **GROUP BY** (A, B) の結果セットは、A と B に定数が含まれているすべてのローについての小計から成ります。UNION を可能にするために、カラム C には NULL が割り当てられます。

ROLLUP クエリの例	ROLLUP を使用せずに記述した同じ内容のクエリ
`select year(orderdate) as year, quarter(orderdate) as Quarter, count(*) Ordersfrom SalesOrdersgroup by Rollup (year, quarter)order by year, quarter`	`Select null,null, count(*) Orders from SalesOrdersunion allSELECT year(orderdate) AS YEAR, NULL, count(*) Orders from SalesOrdersGROUP BY year(orderdate) union allSELECT year(orderdate) as YEAR, quarter(orderdate) as QUATER, count(*) Orders from SalesOrdersGROUP BY year(orderdate), quarter(orderdate)`

小計ローはデータの分析に役立ちます。特に、データが大量にある場合、データにさまざまな次元がある場合、データがさまざまなテーブルに含まれている場合、またはまったく異なるデータベースに含まれている場合に威力を発揮します。たとえば販売マネージャが、売上高についてのレポートを営業担当者別、地域別、四半期別に整理して、売上パターンの理解に役立てることができます。データの小計は、販売マネージャが売上高の全体像をさまざまな視点から分析するのに役立ちます。販売マネージャが比較したいと考える基準に基づいて要約情報が提供されていれば、データの分析を容易に行うことができます。

OLAP を使用すると、ローおよびカラムの小計を分析して計算する処理をユーザの目から隠すことができます。

図 2 : 小計



1. このステップで、まだ **ROLLUP** とは見なされない中間の結果セットが生成されます。
2. 小計が評価され、結果セットに付加されます。
3. クエリ内の **ORDER BY** 句に従ってローが並べられます。

NULL 値と小計ロー

GROUP BY 操作に対する入力でローに NULL が含まれている場合、**ROLLUP** または **CUBE** 操作によって追加された小計ローと、最初の入力データの一部として NULL 値を含んでいるローが混在している可能性があります。

Grouping() 関数は、小計ローをその他のローから区別します。具体的には、**GROUP BY** リストのカラムを引数として受け取り、そのカラムが小計ローであるために NULL になっている場合は 1 を返し、それ以外の場合は 0 を返します。

次の例では、結果セットの中に Grouping() カラムが含まれています。強調表示されているローは、小計ローであるために NULL を含んでいるのではなく、入力データの結果として NULL を含んでいるローです。Grouping() カラムは強調表示されています。 このクエリは、`Employees` テーブルと `SalesOrders` テーブルの間の外部ジョインです。 このクエリでは、テキサス、ニューヨーク、またはカリフォルニアに住んでいる女性従業員を選択しています。営業担当者でない (したがって売上がない) 女性従業員については、カラムに NULL が表示されます。

注意： たとえば、Sybase IQ デモ・データベース `iqdemo.db` を使用します。

```
SELECT Employees.EmployeeID as EMP, year(OrderDate) as
  YEAR, count(*) as ORDERS, grouping(EMP) as
  GE, grouping(YEAR) as GY
  FROM Employees LEFT OUTER JOIN SalesOrders on
  Employees.EmployeeID = SalesOrders.SalesRepresentative
  WHERE Employees.Sex IN ('F') AND Employees.State
  IN ('TX', 'CA', 'NY')
GROUP BY ROLLUP (YEAR, EMP)
ORDER BY YEAR, EMP
```

前述のクエリは、以下を返します。

```
EMP         YEAR      ORDERS     GE     GY
------      ----      ------     --     --
NULL        NULL           5      1      0
NULL        NULL         169      1      1
 102        NULL           1      0      0
```

```
 309        NULL          1      0      0
1062        NULL          1      0      0
1090        NULL          1      0      0
1507        NULL          1      0      0
NULL        2000         98      1      0
 667        2000         34      0      0
 949        2000         31      0      0
1142        2000         33      0      0
NULL        2001         66      1      0
 667        2001         20      0      0
949         2001         22      0      0
1142        2001         24      0      0
```

個々のプレフィクスについて、プレフィクス・カラムに同じ値が含まれているすべてのローに関する*小計ロー*が作成されます。

ROLLUP の結果を具体的に説明するために、前述のクエリの例をもう一度詳しく見ていきます。

```
SELECT year (OrderDate) AS Year, quarter
  (OrderDate) AS Quarter, COUNT (*) Orders
FROM SalesOrders
  GROUP BY ROLLUP (Year, Quarter)
  ORDER BY Year, Quarter
```

このクエリでは、`Year` カラムを含んでいるプレフィクスにより、Year=2000 の合計ローと Year=2001 の合計ローが作成されます。このプレフィクスに関する 1 つの合計ローはカラムを含んでいません。これは、中間の結果セットに含まれているすべてのローの小計です。

小計ローの各カラムの値は、次のようになっています。

- プレフィクスに含まれているカラム ― そのカラムの値です。 たとえば、前述のクエリでは、Year=2000 のローに関する小計の `Year` カラムの値は 2000 になります。
- プレフィクスから除外されたカラム ― NULL です。 たとえば、Year カラムから成るプレフィクスにより生成された小計ローでは、`Quarter` カラムの値は NULL になります。
- 集合関数 ― 除外されているカラムの値を計算した結果です。
  小計値は、集約されたローではなく、基本データのローに対して計算されます。 多くの場合、たとえば **SUM** や **COUNT** などでは結果は等しくなりますが、**AVG**、**STDDEV**、**VARIANCE** などの統計関数では結果が異なってくるため、この区別は重要です。

ROLLUP 演算子には次の制限があります。

- **ROLLUP** 演算子は、**GROUP BY** 句で使用できるすべての集合関数をサポートしています (**COUNT DISTINCT** と **SUM DISTINCT** は除く)。

- **ROLLUP** は、**SELECT** 文のみで使用できます。サブクエリでは **ROLLUP** を使用できません。
- 複数の **ROLLUP**、**CUBE**、**GROUP BY** カラムを同じ **GROUP BY** 句で組み合わせたグループ化の指定は、現在サポートされていません。
- **GROUP BY** のキーに定数式を指定することはできません。

式の一般的なフォーマットについては、『リファレンス：ビルディング・ブロック、テーブル、およびプロシージャ』の「式」、および『リファレンス：ビルディング・ブロック、テーブル、およびプロシージャ』の「SQL 言語の要素」を参照してください。

ROLLUP 例 2 - 次は、**ROLLUP** と **GROUPING** の使用例です。**GROUPING** によって作成される一連のマスク・カラムを表示します。 カラム S、N、C に表示されている数字 0 と 1 は、**GROUPING** からの戻り値であり、**ROLLUP** の結果の値を表現しています。マスクが "011" であれば小計のローであり、"111" であれば総計のローであると特定できます。これを利用して、クエリの結果をプログラムで分析することが可能です。

```
SELECT size, name, color, SUM(quantity),
  GROUPING(size) AS S,
  GROUPING(name) AS N,
  GROUPING(color) AS C
FROM Products
GROUP BY ROLLUP(size, name, color) HAVING (S=1 or N=1 or C=1)
ORDER BY size, name, color;
```

前述のクエリは、以下を返します。

```
size               name           color     SUM  S  N  C
----               -----          ------     ---  -  -  -
(NULL)             (NULL)          (NULL)    496  1  1  1
Large              (NULL)          (NULL)     71  0  1  1
Large              Sweatshirt      (NULL)     71  0  0  1
Medium             (NULL)          (NULL)    134  0  1  1
Medium             Shorts          (NULL)     80  0  0  1
Medium             Tee Shirt       (NULL)     54  0  0  1
One size fits all  (NULL)          (NULL)    263  0  1  1
One size fits all  Baseball Cap    (NULL)    124  0  0  1
One size fits all  Tee Shirt       (NULL)     75  0  0  1
One size fits all  Visor           (NULL)     64  0  0  1
Small              (NULL)          (NULL)     28  0  1  1
Small              Tee Shirt       (NULL)     28  0  1  1
```

注意： ROLLUP 例 2 の結果では、SUM カラムは *SUM(products.quantity)* で表示します。

ROLLUP 例 3 - 次の例は、**GROUPING** を使用して、最初から格納されていた NULL 値と **ROLLUP** 操作によって生成された “NULL” 値を区別する方法を示しています。 このクエリで指定されているとおり、最初から格納されていた NULL 値はカ

ラム prod_id に [NULL] として表示され、**ROLLUP** によって生成された “NULL” 値はカラム PROD_IDS で ALL に置き換えられます。

```
SELECT year(ShipDate) AS Year,
                ProductID, SUM(quantity)AS OSum,
CASE
        WHEN GROUPING(Year) = 1
        THEN 'ALL'
        ELSE
        CAST(Year AS char(8))
END,
CASE
        WHEN GROUPING(ProductID) = 1
        THEN 'ALL'
        ELSE
        CAST(ProductIDas char(8))
END
FROM SalesOrderItems
GROUP BY ROLLUP(Year, ProductID) HAVING OSum > 36
ORDER BY Year, ProductID;
```

前述のクエリは、以下を返します。

```
Year  ProductID  OSum  ...(Year)...  ...(ProductID)...
---------     -------   ---   ----------   --------
NULL           NULL   28359   ALL          ALL
2000           NULL   17642   2000         ALL
2000           300     1476   2000         300
2000           301     1440   2000         301
2000           302     1152   2000         302
2000           400     1946   2000         400
2000           401     1596   2000         401
2000           500     1704   2000         500
2000           501     1572   2000         501
2000           600     2124   2000         600
2000           601     1932   2000         601
2000           700     2700   2000         700
2001           NULL   10717   2001         ALL
2001           300      888   2001         300
2001           301      948   2001         301
2001           302      996   2001         302
2001           400     1332   2001         400
2001           401     1105   2001         401
2001           500      948   2001         500
2001           501      936   2001         501
2001           600      936   2001         600
2001           601      792   2001         601
2001           700     1836   2001         700
```

ROLLUP 例 4 - 次のクエリ例は、注文数を年別および四半期別に集計したデータを返します。

```
SELECT year (OrderDate) AS Year,
quarter(OrderDate) AS Quarter, COUNT (*) Orders
FROM SalesOrders
```

```
GROUP BY ROLLUP (Year, Quarter)
ORDER BY Year, Quarter
```

次の図は、このクエリの結果を示しています。結果セット内の小計ローは強調表示されています。各小計ローでは、その小計の計算対象になったカラムに NULL 値が格納されています。

	Year	Quarter	Orders
①	**(NULL)**	**(NULL)**	**648**
②	**2000**	**(NULL)**	**380**
③	2000	1	87
	2000	2	77
	2000	3	91
	2000	4	235
②	**2001**	**(NULL)**	**268**
③	2001	1	139
	2001	2	119
	2001	3	10

ロー①は、両方の年 (2000 年および 2001 年) のすべての四半期の注文数の合計を示しています。 このローは、`Year` カラムと `Quarter` カラムの両方が NULL であり、すべてのカラムがプレフィクスから除外されています。

注意： すべての **ROLLUP** 操作によって返される結果セットには、集合カラムを除くすべてのカラムが NULL であるローが 1 つ含まれています。このローは、集合関数に対する全カラムの要約を表しています。たとえば、集合関数として SUM を使用している場合は、このローはすべての値の総計を表します。

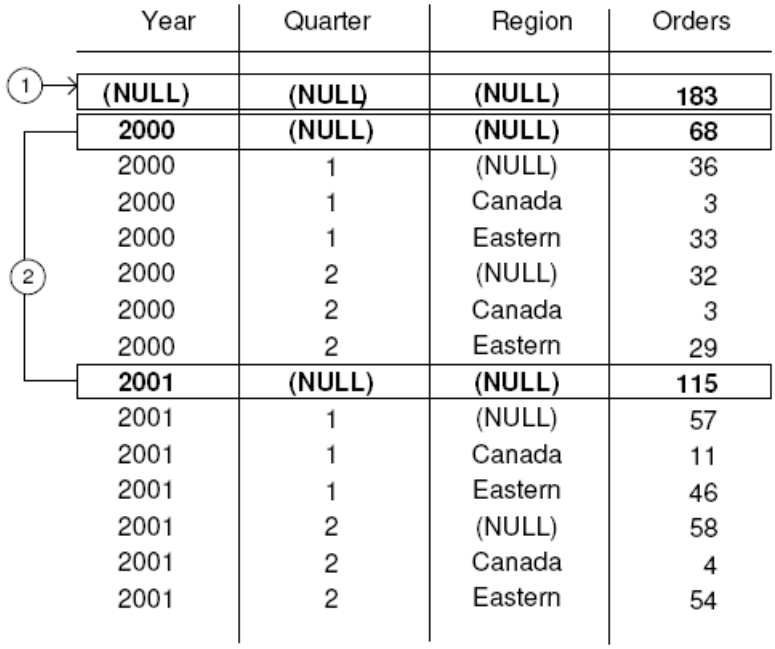
ロー②は、2000 年および 2001 年の注文数の合計をそれぞれ示しています。 どちらのローも、`Quarter` カラムは NULL になっています。これは、このカラムの値を加算して、`Year` の小計を出しているためです。 結果セットにこのようなローが含まれる数は、**ROLLUP** クエリで使用されている変数の数に応じて異なります。

③としてマークされている残りのローは要約情報を示し、それぞれの年の各四半期の注文数の合計を表しています。

ROLLUP 例 5 - **ROLLUP** 操作の例では、年別、四半期別、地域別の注文数を集計する、少し複雑な結果セットを返します。この例では、第 1 および第 2 四半期と 2 つの地域 (カナダと東部地区) だけを分析します。

```
SELECT year(OrderDate) AS Year, quarter(OrderDate)AS Quarter,
region, COUNT(*) AS OrdersFROM SalesOrders WHERE region IN
('Canada','Eastern') AND quarter IN (1, 2)GROUP BY ROLLUP (Year,
Quarter, Region)ORDER BY Year, Quarter, Region
```

次の図は、このクエリの結果セットを示しています。各小計ローでは、その小計の計算対象になったカラムに NULL が格納されています。



	Year	Quarter	Region	Orders
①	**(NULL)**	**(NULL)**	**(NULL)**	**183**
	2000	**(NULL)**	**(NULL)**	**68**
	2000	1	(NULL)	36
	2000	1	Canada	3
	2000	1	Eastern	33
②	2000	2	(NULL)	32
	2000	2	Canada	3
	2000	2	Eastern	29
	2001	**(NULL)**	**(NULL)**	**115**
	2001	1	(NULL)	57
	2001	1	Canada	11
	2001	1	Eastern	46
	2001	2	(NULL)	58
	2001	2	Canada	4
	2001	2	Eastern	54

ロー①はすべてのローの集約結果であり、`Year`、`Quarter`、`Region` カラムに NULL が含まれています。 このローの `Orders` カラムの値は、カナダと東部地区における、2000 年と 2001 年の第 1 および第 2 四半期の注文数の合計を示しています。

②としてマークされているローは、それぞれの年 (2000 年と 2001 年) におけるカナダと東部地区の第 1 および第 2 四半期の注文数の合計を示しています。ロー②の値を足すと、ロー①に示されている総計に等しくなります。

③としてマークされているローは、特定の年および四半期の全地域の注文数の合計を示しています。

Year	Quarter	Region	Orders
(NULL)	(NULL)	(NULL)	183
2000	(NULL)	(NULL)	68
2000	**1**	**(NULL)**	**36**
2000	1	Canada	3
2000	1	Eastern	33
2000	**2**	**(NULL)**	**32**
2000	2	Canada	3
2000	2	Eastern	29
2001	(NULL)	(NULL)	115
2001	**1**	**(NULL)**	**57**
2001	1	Canada	11
2001	1	Eastern	46
2001	**2**	**(NULL)**	**58**
2001	2	Canada	4
2001	2	Eastern	54

③

④としてマークされているローは、結果セット内のそれぞれの年の各四半期の各地域の注文の合計数を示しています。

Year	Quarter	Region	Orders
(NULL)	(NULL)	(NULL)	183
2000	(NULL)	(NULL)	68
2000	1	(NULL)	36
2000	1	Canada	3
2000	1	Eastern	33
2000	2	(NULL)	32
2000	2	Canada	3
2000	2	Eastern	29
2001	(NULL)	(NULL)	115
2001	1	(NULL)	57
2001	1	Canada	11
2001	1	Eastern	46
2001	2	(NULL)	58
2001	2	Canada	4
2001	2	Eastern	54

④

GROUP BY CUBE

GROUP BY 句の **CUBE** 演算子は、データを複数の次元 (グループ化式) でグループ化することでデータを分析します。

CUBE では、次元の順序リストを引数として指定する必要があります。これにより、**SELECT** 文の中で、そのクエリに指定した次元グループの考えられるすべての組み合わせの小計を計算し、選択した複数のカラムのすべての値の組み合わせに関する要約を示す結果セットを生成できます。

CUBE 構文：

```
            SELECT … [ GROUPING (column-name) … ] …
GROUP BY [ expression [,…]
| CUBE ( expression [,…] ) ]
```

GROUPING は、カラム名をパラメータとして受け取り、次の表に示すようにブール値を返します。

表 3 : CUBE 演算子が指定された GROUPING によって返される値

結果値の種類	GROUPING が返す値
CUBE 処理によって作成された NULL	1 (真)
ローが小計であることを示す NULL	1 (真)
CUBE 処理によって作成されたもの以外の NULL	0 (偽)
格納されていた NULL	0 (偽)

CUBE は、同じ階層の一部ではない次元を扱うときに特に有用です。

SQL 構文の例	定義されるセット
`GROUP BY CUBE (A, B, C);`	(A, B, C) (A, B) (A, C) (A) (B, C) (B) (C) ( )

CUBE 演算子には次の制限があります。

- **CUBE** 演算子は、**GROUP BY** 句で使用できるすべての集合関数をサポートしていますが、**COUNT DISTINCT** または **SUM DISTINCT** では、**CUBE** は現在サポートされていません。
- **CUBE** は、逆分散統計関数 (**PERCENTILE_CONT** と **PERCENTILE_DISC**) では現在サポートされていません。
- **CUBE** は、**SELECT** 文のみで使用できます。**SELECT** サブクエリでは **CUBE** を使用できません。
- **ROLLUP**、**CUBE**、**GROUP BY** カラムを同じ GROUP BY 句で組み合わせた **GROUPING** の指定は、現在サポートされていません。
- **GROUP BY** のキーに定数式を指定することはできません。

注意： キューブのサイズがテンポラリ・キャッシュのサイズを超えると、**CUBE** のパフォーマンスが低下します。

GROUPING と **CUBE** 演算子を併用すると、格納されていた NULL 値と **CUBE** によって作成されたクエリ結果の NULL 値を区別できます。

GROUPING 関数を使用して結果を分析する方法については、**ROLLUP** 演算子の説明で示されている例を参照してください。

すべての **CUBE** 操作が返す結果セットには、集合カラムを除くすべてのカラムの値が NULL であるローが、少なくとも 1 つは含まれています。このローは、集合関数に対する全カラムの要約を表しています。

CUBE 例 1 - 次の例は、対象者の州 (地理的な位置)、性別、教育レベル、および収入などで構成される調査データを使用したクエリです。 最初に紹介するクエリには **GROUP BY** 句が指定されています。この句は、クエリの結果を、census テーブルの state、gender、education カラムの値に応じてロー・グループに分類し、収入の平均とローの合計数をグループごとに計算します。 このクエリには **GROUP BY** 句のみを使用し、ローのグループ化に **CUBE** 演算子を使用していません。

```
SELECT State, Sex as gender, DepartmentID,
COUNT(*),CAST(ROUND(AVG(Salary),2) AS NUMERIC(18,2))AS AVERAGEFROM
employees WHERE state IN ('MA' , 'CA')GROUP BY State, Sex,
DepartmentIDORDER BY 1,2;
```

このクエリの結果セットを次に示します。

```
state  gender  DepartmentID  COUNT()     AVERAGE
-----  ------  -------   --------    --------
CA       F       200       2         58650.00
CA       M       200           1         39300.00
```

GROUP BY 句の **CUBE** 拡張機能を使用すると、調査データを 1 回参照するだけで、調査データ全体における州別、性別、教育別の平均収入を計算し、state、gender、education カラムの考えられるすべての組み合わせでの平均収入を計

算できます。 **CUBE** 演算子を使用すると、たとえば、すべての州における全女性の平均収入を計算したり、調査対象者全員の平均収入を、各自の教育別および州別に計算したりすることができます。

CUBE でグループを計算する場合、計算されたグループのカラムに NULL 値が生成されます。 最初からデータベース内に格納されていた NULL であるのか、**CUBE** の結果として生成された NULL であるのかを区別するには、**GROUPING** 関数を使用します。 **GROUPING** 関数は、指定されたカラムが上位レベルのグループにマージされている場合は 1 を返します。

CUBE 例 2 - 次のクエリは、**GROUPING** 関数と **GROUP BY CUBE** を併用した例を示しています。

```
SELECT case grouping(State) WHEN 1 THEN 'ALL' ELSE StateEND AS
c_state, case grouping(sex) WHEN 1 THEN 'ALL'ELSE Sex end AS
c_gender, case grouping(DepartmentID)WHEN 1 THEN 'ALL' ELSE
cast(DepartmentID as char(4)) endAS c_dept, COUNT(*),
CAST(ROUND(AVG(salary),2) ASNUMERIC(18,2))AS AVERAGEFROM employees
WHERE state IN ('MA' , 'CA')GROUP BY CUBE(state, sex,
DepartmentID)ORDER BY 1,2,3;
```

このクエリの結果は次のとおりです。 **CUBE** が生成した小計ローを示す NULL 値が、クエリ内の指定によって小計ローで ALL に置き換えられています。

```
c_state   c_gender  c_dept    COUNT()   AVERAGE
-------   --------  -------    -----     --------
ALL       ALL        200         3       52200.00
ALL       ALL        ALL         3       52200.00
ALL       F          200         2       58650.00
ALL       F          ALL         2       58650.00
ALL       M         200          1       39300.00
ALL       M         ALL          1       39300.00
CA        ALL       200          3       52200.00
CA        ALL       ALL          3       52200.00
CA        F         200          2       58650.00
CA        F         ALL          2       58650.00
CA        M         200          1       39300.00
CA        M         ALL          1       39300.00
```

CUBE 例 3 - この例のクエリは、注文数の合計を要約する結果セットを返し、次に、年別および四半期別の注文数の小計を計算します。

注意： 比較する変数の数が増えると、キューブの計算コストが急激に増大します。

```
SELECT year (OrderDate) AS Year, quarter(OrderDate) AS Quarter, COUNT
(*) OrdersFROM SalesOrdersGROUP BY CUBE (Year, Quarter)ORDER BY Year,
Quarter
```

次の図は、このクエリの結果セットを示しています。この結果セットでは、小計ローが強調表示されています。各小計ローでは、その小計の計算対象になったカラムに NULL が格納されています。

	Year	Quarter	Orders
①	(NULL)	(NULL)	648
②	(NULL)	1	226
	(NULL)	2	196
	(NULL)	3	101
	(NULL)	4	125
③	2000	(NULL)	380
	2000	1	87
	2000	2	77
	2000	3	91
	2000	4	125
③	2001	(NULL)	268
	2001	1	139
	2001	2	119
	2001	3	10

先頭のロー①は、両方の年のすべての四半期の注文数の合計を示しています。`Orders` カラムの値は、③としてマークされている各ローの値の合計です。これは、②としてマークされている 4 つのローの値の合計でもあります。

②としてマークされている一連のローは、両方の年の四半期別の注文数の合計を示しています。③としてマークされている 2 つのローは、それぞれ 2000 年および 2001 年のすべての四半期の注文数の合計を示しています。

分析関数

Sybase IQ では、1 つの SQL 文内で複雑なデータ分析を実行できる機能を備えた、単純な集合関数とウィンドウ集合関数の両方を提供しています。

これらの関数を使用して、たとえば「ダウ工業株 30 種平均の四半期の移動平均はどうなっているか」または「各部署のすべての従業員とその累積給与を一覧表示せよ」というクエリの結果を計算できます。 さまざまな期間における移動平均と累積和を計算し、集計とランクを分割できるため、パーティション値が変化したときに集合計算がリセットされます。1 つのクエリ式のスコープ内で、それぞれ独自の分割ルールを持ついくつかの異なる OLAP 関数を定義することができます。分析関数は 2 つのカテゴリに分けられます。

- 単純な集合関数 (**AVG**、**COUNT**、**MAX**、 **MIN**、**SUM** など) は、データベースに含まれるローのグループのデータを要約します。 **SELECT** 文の **GROUP BY** 句を使用して、グループを作成します。

- 1 つの引数を取る単項の統計集合関数には、**STDDEV**、**STDDEV_SAMP**、**STDDEV_POP**、**VARIANCE**、**VAR_SAMP**、**VAR_POP.** があります。

単純な集合関数でも単項の集合関数でも、データベース内のローのグループに関するデータを要約することができ、ウィンドウ指定と組み合わせて、処理の際に結果セットに対する移動ウィンドウを計算することができます。

注意： 集合関数 **AVG**、**SUM**、**STDDEV**、**STDDEV_POP**、**STDDEV_SAMP**、**VAR_POP**、**VAR_SAMP**、**VARIANCE** は、バイナリ・データ型 (BINARY と VARBINARY) をサポートしていません。

単純な集合関数

単純な集合関数 (**AVG**、**COUNT**、**MAX**、 **MIN**、**SUM** など) は、データベースに含まれるローのグループのデータを要約します。

SELECT 文の **GROUP BY** 句を使用して、グループを作成します。 集合関数は、select リストと、**SELECT** 文の **HAVING** および **ORDER BY** 句の中のみで使用できます。

注意： Grouping() 関数を除き、単純な集合関数と単項の集合関数はどちらも、SQL クエリの指定に「ウィンドウ句」(*ウィンドウ*) を組み込むウィンドウ関数として使用できます。これにより、処理時に結果セットに対して概念的に移動ウィンドウを作成できます。

『リファレンス：ビルディング・ブロック、テーブル、およびプロシージャ』の「SQL 関数」>「集合関数」を参照してください。

ウィンドウ

OLAP に関する ANSI SQL 拡張で導入された主な機能は、「*ウィンドウ*」という名前の構成体です。 このウィンドウ拡張により、ユーザはクエリの結果セット (クエリの論理パーティション) をパーティションと呼ばれるローのグループに分割し、現在のローについて集計するローのサブセットを決定できます。

1 つのウィンドウで 3 つのウィンドウ関数クラス (ランク付け関数、ロー・ナンバリング関数、ウィンドウ集合関数) を使用できます。

```
<WINDOWED TABLE FUNCTION TYPE> ::=
  <RANK FUNCTION TYPE> <LEFT PAREN> <RIGHT PAREN>
  | ROW_NUMBER <LEFT PAREN> <RIGHT PAREN>
  | <WINDOW AGGREGATE FUNCTION>
```

ウィンドウ拡張は、ウィンドウ名または指定に対するウィンドウ関数の種類を指定し、1 つのクエリ式のスコープ内のパーティション化された結果セットに適用されます。 ウィンドウ・パーティションは、特殊な **OVER** 句の 1 つ以上のカラムで定義されている、クエリから返されるローのサブセットです。

```
olap_function() OVER (PARTITION BY col1, col2...)
```

ウィンドウ操作では、パーティション内の各ローのランク付け、パーティション内のローの値の分布、および類似の操作などの情報を設定できます。また、データの移動平均や合計を計算し、データおよびそのデータの操作に対する影響を評価する機能を拡張することもできます。

OLAP ウィンドウの 3 つの重要な側面

OLAP ウィンドウは、ウィンドウ・パーティション、ウィンドウ順序、ウィンドウ・フレームという 3 つの重要な側面から成ります。それぞれの要素は、その時点でウィンドウ内で可視となるデータ・ローに大きな影響を与えます。 また、OLAP の **OVER** 句は、次の 3 つの特徴的な機能により、OLAP 関数を他の分析関数やレポート関数から区別します。

- ウィンドウ・パーティションの定義 (**PARTITION BY** 句)。
- パーティション内でのローの順序付け (**ORDER BY** 句)。
- ウィンドウ・フレームの定義 (ROWS/RANGE 指定)。

複数のウィンドウ関数を指定したり、冗長なウィンドウ定義を避けたりするために、OLAP ウィンドウ指定に関して名前を指定できます。 その場合は、キーワード **WINDOW** の後に少なくとも 1 つのウィンドウ定義を指定します (複数指定する場合はカンマで区切ります)。ウィンドウ定義には、クエリ内でウィンドウを識別するための名前と、ウィンドウ・パーティション、順序、フレームを定義するためのウィンドウ指定の詳細を含めます。

```
<WINDOW CLAUSE> ::= <WINDOW DEFINITION LIST>
```

```
<WINDOW DEFINITION LIST> ::=
  <WINDOW DEFINITION> [ { <COMMA> <WINDOW DEFINITION>
  } . . . ]
```

```
<WINDOW DEFINITION> ::=
  <NEW WINDOW NAME> AS <WINDOW SPECIFICATION>
```

```
<WINDOW SPECIFICATION DETAILS> ::=
  [ <EXISTING WINDOW NAME> ]
  [ <WINDOW PARTITION CLAUSE> ]
  [ <WINDOW ORDER CLAUSE> ]
  [ <WINDOW FRAME CLAUSE> ]
```

ウィンドウ・パーティション内の各ローについて、ウィンドウ・フレームを定義できます。ウィンドウ・フレームにより、パーティションの現在のローに対して計算を実行するときに使用される、ローの指定範囲を変更できます。現在のローは、ウィンドウ・フレームの開始ポイントと終了ポイントを決定するための参照ポイントとなります。

ウィンドウ指定は、物理的なローの数 (ウィンドウ・フレーム単位 ROWS を定義するウィンドウ指定を使用) または論理的な数値の間隔 (ウィンドウ・フレーム単位 RANGE を定義するウィンドウ指定を使用) に基づきます。

OLAP のウィンドウ操作では、次のカテゴリの関数を使用できます。

- ランク付け関数
- ウィンドウ集合関数
- 統計集合関数
- 分散統計関数

参照：
- *分散統計関数* (64 ページ)
- *OLAP の利点* (24 ページ)
- *OLAP の評価* (24 ページ)
- *ランク付け関数* (53 ページ)
- *統計集合関数* (59 ページ)
- *ウィンドウ集合関数* (57 ページ)
- *OLAP 関数の BNF 文法* (79 ページ)

ウィンドウ・パーティション

ウィンドウ・パーティションとは、 **PARTITION BY** 句を使用して、ユーザ指定の結果セット (入力ロー) を分割することです。

パーティションは、カンマで区切られた 1 つ以上の値の式によって定義されます。パーティションに分割されたデータは暗黙的にソートされ、デフォルトのソート順序は昇順 (ASC) になります。

```
<WINDOW PARTITION CLAUSE> ::=
  PARTITION BY <WINDOW PARTITION EXPRESSION LIST>
```

ウィンドウ・パーティション句を指定しなかった場合は、入力が 1 つのパーティションとして扱われます。

注意： 統計関数に対して *「パーティション」* という用語を使用した場合は、結果セットのローを **PARTITION BY** 句に基づいて分割することのみを意味します。

ウィンドウ・パーティションは任意の式に基づいて定義できます。 また、ウィンドウ・パーティションの処理は GROUPING の後に行われるため (**GROUP BY** 句が指定されている場合)、**SUM**、**AVG**、**VARIANCE** などの集合関数の結果をパーティションの式で使用できます。 したがって、パーティションを使用すると、**GROUP BY** 句や **ORDER BY** 句 *とはまた別に*、グループ化と順序付けの操作を実行できます。たとえば、ある数量の最大 **SUM** を求めるなど、集合関数に対して集合関数を計算するクエリを記述できます。

GROUP BY 句がなくても、**PARTITION BY** 句を指定できます。

参照：
- *ウィンドウ・フレーム* (44 ページ)
- *ウィンドウ順序* (43 ページ)

ウィンドウ順序

ウィンドウ順序とは、各ウィンドウ・パーティション内の結果 (ロー) を WINDOW ORDER 句に基づいて並べることです。この句には、1 つ以上の値の式をカンマ区切りで指定します。

WINDOW ORDER 句を指定しなかった場合は、入力ローが任意の順序で処理されることがあります。

```
<WINDOW ORDER CLAUSE> ::= <ORDER SPECIFICATION>
```

OLAP の WINDOW ORDER 句は、非ウィンドウ・クエリの式に指定できる **ORDER BY** 句とは異なります。

OLAP 関数で使用する **ORDER BY** 句は、通常はウィンドウ・パーティション内のローをソートするための式を定義しますが、**PARTITION BY** 句がなくても **ORDER BY** 句を使用できます。この場合、このソート指定によって、意味のある (かつ目的どおりの) 順序で並べられた中間の結果セットに OLAP 関数を確実に適用できます。

OLAP のランク付け関数には順序の指定が必須であり、ランキング値の基準は、ランク付け関数の引数ではなく **ORDER BY** 句で指定します。 OLAP の集合関数では、通常は **ORDER BY** 句の指定は必須ではありませんが、ウィンドウ・フレームを定義するときには必須です。これは、各フレームの適切な集合値を計算する前に、パーティション内のローをソートしなければならないためです。

この **ORDER BY** 句には、昇順および降順のソートを定義するためのセマンティックと、NULL 値の取り扱いに関する規則を指定します。OLAP 関数は、デフォルトでは昇順 (最も小さい値が 1 番目にランク付けされる) を使用します。

これは **SELECT** 文の最後に指定する **ORDER BY** 句のデフォルト動作と同じですが、連続的な計算を行う場合にはわかりにくいかもしれません。 ほとんどの OLAP の計算では、降順 (最も大きい値が 1 番目にランク付けされる) でのソートが必要になります。この要件を満たすには、**ORDER BY** 句に明示的に DESC キーワードを指定する必要があります。

注意： ランク付け関数は、ソートされた入力のみを扱うように定義されているため、「WINDOW ORDER 句」を指定する必要があります。「クエリ指定」の「ORDER BY 句」と同様に、デフォルトのソート順序は昇順です。

「ウィンドウ・フレーム単位」で RANGE を使用する場合も、「WINDOW ORDER 句」を指定する必要があります。 RANGE の場合は、「WINDOW ORDER 句」に 1 つの式のみを指定します。

参照：
- *ウィンドウ・フレーム* (44 ページ)
- *ウィンドウ・パーティション* (42 ページ)

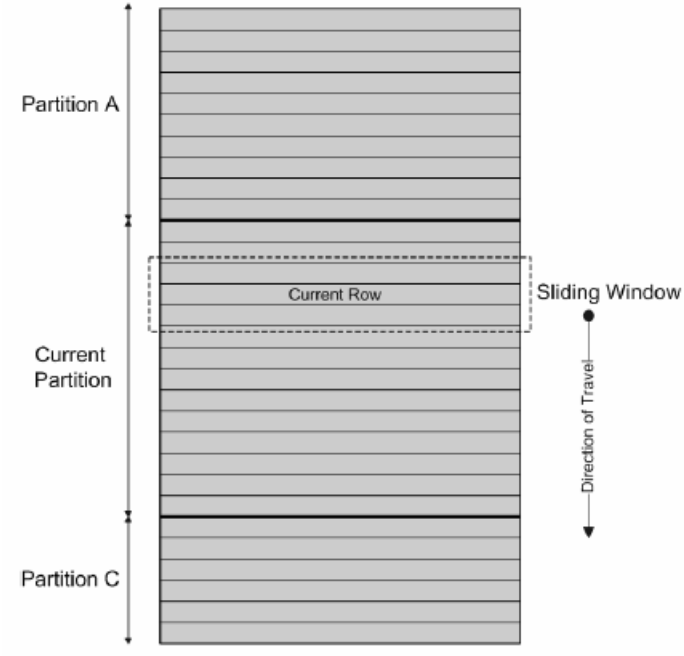
ウィンドウ・フレーム

ランク付け関数を除く OLAP 集合関数では、WINDOW FRAME 句を使用してウィンドウ・フレームを定義できます。この句には、現在のローを基準としてウィンドウの開始位置と終了位置を指定します。

```
<WINDOW FRAME CLAUSE> ::=
  <WINDOW FRAME UNIT>
  <WINDOW FRAME EXTENT>
```

これにより、パーティション全体の固定的な内容ではなく、移動するフレームの内容に対して OLAP 関数を計算できます。定義にもよりますが、パーティションには開始ローと終了ローがあり、ウィンドウ・フレームは開始ポイントからパーティションの終了位置に向けてスライドします。

図 3 : 分割された入力と、3 ロー分の移動ウィンドウ



UNBOUNDED PRECEDING と UNBOUNDED FOLLOWING
ウィンドウ・フレームは、パーティションの先頭 (UNBOUNDED PRECEDING)、最後 (UNBOUNDED FOLLOWING)、または両方まで到達する無制限の集合グループによって定義されます。

UNBOUNDED PRECEDING には、パーティション内の現在のロー以前にあるすべてのローが含まれており、ROWS または RANGE で指定できます。 UNBOUNDED FOLLOWING には、パーティション内の現在のロー以後にあるすべてのローが含まれており、ROWS または RANGE で指定できます。

FOLLOWING の値では、現在のロー以降にあるローの範囲または数を指定します。ROWS を指定する場合、その値には、ローの数を表す正の数を指定します。RANGE を指定する場合、そのウィンドウには、現在のローに指定の数値を足した数よりも少ないローが含まれます。 RANGE を指定する場合、そのウィンドウ値のデータ型は、**ORDER BY** 句のソート・キー式の型に対応している必要があります。 指定できるソート・キー式は 1 つのみで、このソート・キー式のデータ型は「加算」を許可している必要があります。

PRECEDING の値では、現在のロー以前にあるローの範囲または数を指定します。ROWS を指定する場合、その値には、ローの数を表す正の数を指定します。RANGE を指定する場合、そのウィンドウには、現在のローから指定の数値を引いた数よりも少ないローが含まれます。 RANGE を指定する場合、そのウィンドウ値のデータ型は、**ORDER BY** 句のソート・キー式の型に対応している必要があります。 指定できるソート・キー式は 1 つのみで、このソート・キー式のデータ型は「減算」を許可している必要があります。1 つ目のバインドされたグループで CURRENT ROW または FOLLOWING の値を指定している場合は、2 つ目のバインドされたグループにこの句を指定することはできません。

BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING の組み合わせを使用すると、グループ化したクエリとのジョインを構築しなくても、パーティション全体についての集合を計算できます。パーティション全体についての集合は、レポート集合とも呼ばれます。

CURRENT ROW の概念
物理的な集合グループでは、現在のローに対する相対位置に基づき、隣接するローの数に応じて、ローを含めるか除外するかが判断されます。現在のローは、クエリの中間結果における次のローへの参照にすぎません。現在のローが前に進むと、ウィンドウ内に含まれる新しいロー・セットに基づいてウィンドウが再評価されます。現在のローをウィンドウ内に含めるという要件はありません。

WINDOW FRAME 句を指定しなかった場合のデフォルトのウィンドウ・フレームは、WINDOW ORDER 句を指定しているかどうかによって異なります。

- ウィンドウ指定に WINDOW ORDER 句が含まれている場合、ウィンドウの開始ポイントは **UNBOUNDED PRECEDING**、終了ポイントは **CURRENT ROW** になり、累積値の計算に適した可変サイズのウィンドウが定義されます。
- ウィンドウ指定に WINDOW ORDER 句が含まれていない場合、ウィンドウの開始ポイントは **UNBOUNDED PRECEDING**、終了ポイントは **UNBOUNDED FOLLOWING** になり、現在のローに関係なく固定サイズのウィンドウが定義されます。

注意： WINDOW FRAME 句はランク付け関数とは併用できません。

ローベース (ロー指定) または値ベース (範囲指定) のウィンドウ・フレーム単位を指定してウィンドウを定義することもできます。

```
<WINDOW FRAME UNIT> ::= ROWS | RANGE
```

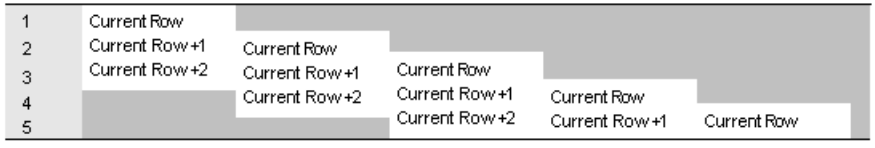
```
<WINDOW FRAME EXTENT> ::= <WINDOW FRAME START> | <WINDOW FRAME
BETWEEN>
```

WINDOW FRAME 句で **BETWEEN** を使用するときは、ウィンドウ・フレームの開始ポイントと終了ポイントを明示的に指定します。

WINDOW FRAME 句でこの 2 つの値のどちらか一方のみ指定した場合は、他方の値がデフォルトで **CURRENT ROW** になります。

ローベースのウィンドウ・フレーム - この例では、ロー [1] ～ [5] は 1 つのパーティションを表しています。それぞれのローは、OLAP のウィンドウ・フレームが前にスライドするにつれて現在のローになります。このウィンドウ・フレームは Between Current Row And 2 Following として定義されているため、各フレームには、最大で 3 つ、最小で 1 つのローが含まれます。フレームがパーティションの終わりに到達したときは、現在のローだけがフレームに含まれます。 網掛けの部分は、各ステップでフレームから除外されているローを表しています。

図 4 : ローベースのウィンドウ・フレーム

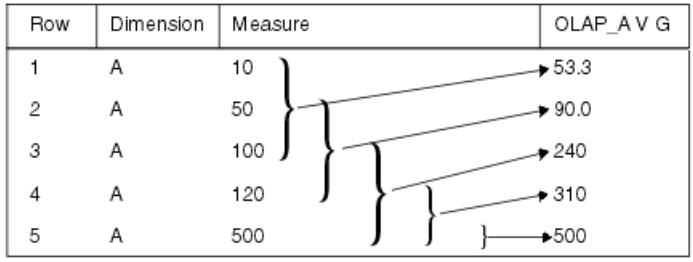


ウィンドウ・フレームは、次のような規則で機能しています。

- ロー [1] が現在のローであるときは、ロー [4] および [5] が除外される。
- ロー [2] が現在のローであるときは、ロー [5] および [1] が除外される。
- ロー [3] が現在のローであるときは、ロー [1] および [2] が除外される。
- ロー [4] が現在のローであるときは、ロー [1]、[2]、[3] が除外される。

- ロー [5] が現在のローであるときは、ロー [1]、[2]、[3]、[4] が除外される。

次の図では、この規則を特定の値セットに適用し、OLAP の **AVG** 関数を使用して各ローを計算しています。スライド計算により、現在のローの位置に応じて、3 つまたはそれ以下のローを範囲として移動平均を算出しています。



次のクエリは、移動ウィンドウの定義の例を示しています。

```
SELECT dimension, measure,
  AVG(measure) OVER(partition BY dimension
    ORDER BY measure
    ROWS BETWEEN CURRENT ROW and 2 FOLLOWING)
    AS olap_avg
FROM ...
```

平均値は次のようにして計算されています。

- ロー [1] = (10 + 50 + 100)/3
- ロー [2] = (50+ 100 + 120)/3
- ロー [3] = (100 + 120 + 500)/3
- ロー [4] = (120 + 500 + NULL)/3
- ロー [5] = (500 + NULL + NULL)/3

結果セット内の以降のすべてのパーティション (たとえば B、C など) についても、同様の計算が実行されます。

現在のウィンドウにローが含まれていない場合、**COUNT** 以外のケースでは、結果は NULL になります。

参照：

- *ウィンドウ順序*(43 ページ)
- *ウィンドウ・パーティション*(42 ページ)

ROWS

ウィンドウ・フレーム単位 ROWS では、現在のローの前後に指定の数のローを含むウィンドウを定義します (現在のローは、ウィンドウの開始ポイントと終了ポイントを決定するための参照ポイントになります)。

それぞれの分析計算は、パーティション内の現在のローに基づいて行われます。ローで表現されるウィンドウを使用して限定的な結果を生成するには、ユニークな順序付けの式を指定する必要があります。

どのウィンドウ・フレームでも、現在のローが参照ポイントになります。SQL/OLAP の構文には、ローベースのウィンドウ・フレームを、現在のローの前または後にある任意の数のロー (または現在のローの前および後ろにある任意の数のロー) として定義するためのメカニズムが用意されています。

ウィンドウ・フレーム単位の代表的な例を次に示します。

- Rows Between Unbounded Preceding and Current Row ― 各パーティションの先頭を開始ポイントとし、現在のローを終了ポイントとするウィンドウを指定します。累積和など、累積的な結果を計算するためのウィンドウを構築するときによく使用されます。
- Rows Between Unbounded Preceding and Unbounded Following ― 現在のローに関係なく、パーティション全体についての固定ウィンドウを指定します。そのため、ウィンドウ集合関数の値は、パーティションのすべてのローで等しくなります。
- Rows Between 1 Preceding and 1 Following ― 3 つの隣接するロー (現在のローとその前後のロー) を含む固定サイズの移動ウィンドウを指定します。このウィンドウ・フレーム単位を使用して、たとえば 3 日間または 3 か月間の移動平均を計算できます。
  ウィンドウ値にギャップがあると、ROWS を使用した場合に意味のない結果が生成されることがあるので注意してください。値セットが連続していない場合は、ROWS の代わりに RANGE を使用することを検討してください。RANGE に基づくウィンドウ定義では、重複する値を含んだ隣接ローが自動的に処理され、範囲内にギャップがあるときに他のローが含まれません。

  注意： 移動ウィンドウでは、入力の最初のローの前、および入力の最後のローの後ろには、NULL 値を含むローが存在することが想定されます。つまり、3 つのローから成る移動ウィンドウの場合は、入力の最後のローを現在のローとして計算するときに、直前のローと NULL 値が計算に含まれます。
- Rows Between Current Row and Current Row ― ウィンドウを現在のローのみに制限します。
- Rows Between 1 Preceding and 1 Preceding ― 現在のローの直前のローのみを含む単一ローのウィンドウを指定します。この指定を、現在のローのみに基づく値

を計算する別のウィンドウ関数と組み合わせると、隣接するロー同士のデルタ (値の差分) を簡単に計算することができます。

参照：

- *RANGE* (49 ページ)

RANGE

範囲ベースのウィンドウ・フレーム - SQL/OLAP 構文では、別の種類のウィンドウ・フレームとして、物理的なローのシーケンスではなく、値ベース (または範囲ベース) のロー・セットに基づいて境界を定義する方法がサポートされています。

値ベースのウィンドウ・フレームは、ウィンドウ・パーティション内で、特定の範囲の数値を含んでいるローを定義します。 OLAP 関数の **ORDER BY** 句では、範囲指定を適用する数値カラムを定義します。このカラムの現在のローの値が、範囲指定の基準となります。範囲指定ではロー指定と同じ構文を使用しますが、構文の解釈の仕方は異なります。

ウィンドウ・フレーム単位 RANGE では、特定の順序付けカラムについて現在のローを基準とする値範囲を指定し、その範囲内の値を持つローを検索して、ウィンドウ・フレームに含めます。これは論理的なオフセットに基づくウィンドウ・フレームと呼ばれ、"3 preceding" などの定数を指定することも、評価結果が数値定数となる任意の式を指定することもできます。 RANGE で定義されているウィンドウを使用するときは、**ORDER BY** 句に数値式を 1 つのみ指定できます。

注意： **ORDER BY** キーは、RANGE ウィンドウ・フレーム内の数値データである必要があります。

たとえば、次のように指定すると、カラムに現在のローの前後数年に当たる *year* 値を含むロー・セットをフレームとして定義できます。

```
ORDER BY year ASC range BETWEEN CURRENT ROW and 1 PRECEDING
```

このクエリ例の 1 PRECEDING という部分は、現在のローの *year* 値から 1 を減算することを意味しています。

このような範囲指定は内包的です。 現在のローの *year* 値が 2000 である場合は、ウィンドウ・パーティション内で、year 値が 2000 および 1999 であるすべてのローがこのフレームに含まれることになります。パーティション内での各ローの物理的な位置は問われません。値ベースのフレームでは、ローを含めたり除外したりする規則が、ローベースのフレームの規則とは大きく異なります (ローベースのフレームの規則は、ローの物理的なシーケンスに完全に依存しています)。

OLAP の **AVG()** 関数の例で考えてみます。次の部分的な結果セットは、値ベースのウィンドウ・フレームの概念を具体的に表しています。前述のように、このフレームには次のローが含まれます。

- 現在のローと同じ year 値を持つロー
- 現在のローから 1 を減算したものと同じ year 値を持つロー

Row	Dimension	Year	Measure	Olap_avg
1	A	1999	10000	10000
2	A	2001	5000	3000
3	A	2001	1000	3000
4	A	2002	12000	5250
5	A	2002	3000	5250

次のクエリは、範囲ベースのウィンドウ・フレーム定義の例を示しています。

```
SELECT dimension, year, measure,
  AVG(measure) OVER(PARTITION BY dimension
    ORDER BY year ASC
    range BETWEEN CURRENT ROW and 1 PRECEDING)
    as olap_avg
FROM ...
```

平均値は次のようにして計算されています。

- ロー [1] = 1999 のため、ロー [2] ～ [5] は除外。したがって AVG = 10,000/1
- ロー [2] = 2001 のため、ロー [1]、[4]、[5] は除外。したがって AVG = 6,000/2
- ロー [3] = 2001 のため、ロー [1]、[4]、[5] は除外。したがって AVG = 6,000/2
- ロー [4] = 2002 のため、ロー [1] は除外。したがって AVG = 21,000/4
- ロー [5] = 2002 のため、ロー [1] は除外。したがって AVG = 21,000/4

値ベースのフレームの昇順と降順 - 値ベースのウィンドウ・フレームを使用する OLAP 関数の **ORDER BY** 句では、範囲指定の対象となる数値カラムを特定するだけではなく、**ORDER BY** 値のソート順序も宣言できます。次の指定により、直前の部分のソート順序 (ASC または DESC) を設定できます。

```
RANGE BETWEEN CURRENT ROW AND n FOLLOWING
```

n FOLLOWING の指定には、次のような意味があります。

- パーティションがデフォルトの昇順 (ASC) でソートされている場合、*n* は正の値として解釈されます。
- パーティションが降順 (DESC) でソートされている場合は、*n* は負の値として解釈されます。

たとえば、year カラムに 1999 ～ 2002 の 4 種類の値が含まれているとします。次の表は、これらの値をデフォルトの昇順でソートした場合 (左側) と降順でソートした場合 (右側) を示しています。

ORDER BY year ASC	ORDER BY year DESC
1999	2002
2000	2001
2001	2000
2002	1999

現在のローが 1999 で、フレームが次のように指定されている場合、このフレームには値 1999 のローと値 1998 のロー (このテーブルには存在しません) が含まれます。

```
ORDER BY year DESC range BETWEEN CURRENT ROW and 1 FOLLOWING
```

注意： **ORDER BY** 値のソート順序は、値ベースのフレームに含まれるローの条件をテストするときに重要な要素です。フレームに含まれるか除外されるかは、数値だけでは決まりません。

無制限ウィンドウの使用 - 次のクエリでは、すべての製品と、すべての製品の総数から成る結果セットが生成されます。

```
SELECT id, description, quantity,
  SUM(quantity) OVER () AS total
FROM products;
```

隣接ロー間のデルタの計算 - 現在のローと前のローをそれぞれ 1 つのウィンドウとして定義し、この 2 つのウィンドウを使用すると、隣接するロー間のデルタ (つまり差分) を直接計算できます。

```
SELECT EmployeeID, Surname, SUM(salary)
OVER(ORDER BY BirthDate rows between current row and current row)
AS curr, SUM(Salary)
OVER(ORDER BY BirthDate rows between 1 preceding and 1 preceding)
AS prev, (curr-prev) as delta
FROM Employees
WHERE State IN ('MA', 'AZ', 'CA', 'CO') AND DepartmentID>10
ORDER BY EmployeeID, Surname;
```

このクエリの結果セットを次に示します。

```
EmployeeID  Surname    curr       prev        delta
------              ---------    --------  ---------  ----------
148             Jordan                   51432.000191
Bertrand                 29800.000            39300.000
-9500.000278          Melkisetian            48500.000
42300.000             6200.000299            Overbey
39300.000             41700.750             -2400.750318
Crow                  41700.750             45000.000
-3299.250586            Coleman                42300.000
46200.000             -3900.000690            Poitras
46200.000              29800.000             16400.000703
Martinez                  55500.800              51432.000
4068.800949            Savarino                 72300.000
55500.800             16799.2001101            Preston
37803.000             48500.000             -10697.0001142
Clark                  45000.000              72300.000
-27300.000
```

ここではウィンドウ関数 **SUM()** を使用していますが、ウィンドウの指定方法により、この合計には現在のローまたは前のローの salary 値のみが含まれています。

また、結果セットの最初のローには前のローが存在しないため、最初のローの prev 値は NULL になります。したがって、delta も NULL になります。

ここまでの例では、**OVER()** 句と一緒に **SUM()** 集合関数を使用しました。

参照：

- *ROWS* (48 ページ)

明示的なウィンドウ句とインラインのウィンドウ句

SQL OLAP では、クエリ内でウィンドウを指定する方法が 2 とおり用意されています。

- 明示的なウィンドウ句。**HAVING** 句の後でウィンドウを定義します。 OLAP 関数を呼び出すときには、このようなウィンドウ句で定義したウィンドウを、ウィンドウの名前を指定して参照します。たとえば、次のようにします。
  ```
  SUM ( ...) OVER w2
  ```
- インラインのウィンドウ指定。クエリ式の **SELECT** リスト内でウィンドウを定義します。 これにより、**HAVING** 句の後のウィンドウ句でウィンドウを定義し、それをウィンドウ関数呼び出しから名前で参照する方法に加えて、関数呼び出しと同時にウィンドウを定義する方法が可能になります。

  注意： インラインのウィンドウ指定を使用する場合は、ウィンドウの名前を指定できません。 1 つの **SELECT** リスト内で複数のウィンドウ関数呼び出しが同じウィンドウを使用する場合は、ウィンドウ句で定義した名前付きウィンドウを参照するか、インラインのウィンドウ定義を繰り返します。

ウィンドウ関数の例 - ウィンドウ関数の例を次に示します。このクエリでは、データを部署別のパーティションに分け、在社年数が最も長い従業員を基点とした従業員の累積給与を計算して、結果セットを返します。この結果セットには、マサチューセッツ在住の従業員だけが含まれます。 Sum_Salary カラムには、従業員の給与の累積和が含まれます。

```
SELECT DepartmentID, Surname, StartDate, Salary,SUM(Salary) OVER
(PARTITION BY DepartmentID ORDER BYstartdate rows between unbounded
preceding andcurrent row) AS sum_salaryFROM EmployeesWHERE State IN
('CA') AND DepartmentID IN (100, 200)ORDER BY DepartmentID;
```

次の結果セットは部署別に分割されています。

```
DepartmentID  Surname   start_date      salary    sum_salary
------------                --------------  ------------
-----------     ------------
200                         Overbey                 1987-02-19
39300.000       39300.000
200                        Savarino                1989-11-07
72300.000      111600.000
200                        Clark                   1990-07-21
45000.000      156600.000
```

ランク付け関数

ランク付け関数を使用すると、データ・セットの値をランク付けされた順序のリストにまとめ、「今年度出荷された製品の中で売上合計が上位 10 位の製品名」または「15 社以上から受注した営業部員の上位 5%」といった要求を満たすクエリを、1 つの SQL 文で作成できます。

SQL/OLAP では、次の 5 つの関数がランク付け関数として分類されています。

```
<RANK FUNCTION TYPE> ::=
  RANK | DENSE_RANK | PERCENT_RANK | ROW_NUMBER | NTILE
```

ランキング関数を使用すると、クエリで指定された順序に基づいて、結果セット内の各ローのランク値を計算することができます。たとえば販売マネージャが、営業成績が最高または最低の営業部員、販売成績が最高または最低の販売地域、または売上が最高または最低の製品を調べたい場合があります。この情報はランキング関数によって入手できます。

参照：

- *分散統計関数* (64 ページ)
- *OLAP の利点* (24 ページ)
- *OLAP の評価* (24 ページ)
- *統計集合関数* (59 ページ)
- *ウィンドウ* (40 ページ)
- *ウィンドウ集合関数* (57 ページ)
- *OLAP 関数の BNF 文法* (79 ページ)

RANK() 関数

RANK 関数は、**ORDER BY** 句で定義されたカラムについて、ローのパーティション内での現在のローのランクを表す数値を返します。

パーティション内の最初のローが 1 位となり、25 のローを含むパーティションでは、パーティション内の最後のローが 25 位となります。**RANK** は構文変換として指定されており、実際に **RANK** を同じ構文に変換することも、変換を行った場合に返される値と同じ結果を返すこともできます。

次の例に出てくる ws1 は、w1 という名前のウィンドウを定義するウィンドウ指定を表しています。

```
RANK() OVER ws
```

これは次の指定に相当します。

```
( COUNT (*) OVER ( ws RANGE UNBOUNDED PRECEDING )
- COUNT (*) OVER ( ws RANGE CURRENT ROW ) + 1 )
```

この **RANK** 関数の変換では、論理的な集合 (RANGE) を使用しています。この結果、同位のロー (順序付けカラムに同じ値が含まれているロー) が複数ある場合は、

それらに同じランクが割り当てられます。パーティション内で異なる値を持つ次のグループには、同位のローのランクよりも 1 以上大きいランクが割り当てられます。たとえば、順序付けカラムに 10、20、20、20、30 という値を含むローがある場合、1 つ目のローのランクは 1 になり、2 つ目のローのランクは 2 になります。3 つ目と 4 つ目のローのランクも 2 になりますが、5 つ目のローのランクは 5 になります。ランクが 3 または 4 のローは存在しません。このアルゴリズムは非連続型ランキング (sparse ranking) とも呼ばれます。

『リファレンス：ビルディング・ブロック、テーブル、およびプロシージャ』の「SQL 関数」>「RANK 関数 [統計]」も参照してください。

DENSE_RANK() 関数

DENSE_RANK DENSE_RANK 関数は、抜けのないランキング値を返します。

同位のローに対しては同じように等しいランク値が割り当てられますが、このローのランクは、個々のローの順位ではなく、順序付けカラムに等しい値を含んでいるローの集まりの順位を表しています。 **RANK** の例と同様に、順序付けカラムに 10、20、20、20、30 という値を含むローがある場合、1 つ目のローのランクは同じく 1 となり、2 つ目のローおよび 3 つ目、4 つ目のローのランクも同じく 2 となります。しかし、最後のローのランクは 5 ではなく 3 になります。

DENSE_RANK も、構文変換を通じて計算されます。

```
DENSE_RANK() OVER ws
```

これは次の指定に相当します。

```
COUNT ( DISTINCT ROW ( expr_1, . . ., expr_n ) )
 OVER ( ws RANGE UNBOUNDED PRECEDING )
```

この例では、*expr_1* から *expr_n* の部分が、ウィンドウ w1 のソート指定リストに含まれている値の式のリストを表しています。

『リファレンス：ビルディング・ブロック、テーブル、およびプロシージャ』の「SQL 関数」>「DENSE_RANK 関数 [統計]」も参照してください。

PERCENT_RANK() 関数

PERCENT_RANK 関数は、小数ではなく、パーセンテージでランクを計算して、0 ～ 1 の 10 進値を返します。

PERCENT_RANK が返すのはローの相対的なランクであり、この数値は、該当するウィンドウ・パーティション内での現在のローの相対位置を表します。 たとえば、順序付けカラムにそれぞれ異なる値を持つ 10 個のローがパーティションに含まれている場合、このパーティションの 3 つ目のローに対する **PERCENT_RANK** の値は 0.222... となります。これは、パーティションの 1 つ目のローに続く 2/9 (22.222...%) のローをカバーしているためです。 次の例に示すとおり、ローの

PERCENT_RANK は、「ローの **RANK** - 1」を「パーティション内のローの数 - 1」で除算したものとして定義されています (“ANT” は、REAL や DOUBLE PRECISION などの概数値の型を表します)。

```
PERCENT_RANK() OVER ws
```

これは次の指定に相当します。

```
CASE
  WHEN COUNT (*) OVER ( ws RANGE BETWEEN UNBOUNDED
  PRECEDING AND UNBOUNDED FOLLOWING ) = 1
  THEN CAST (0 AS ANT)
  ELSE
   ( CAST ( RANK () OVER ( ws ) AS ANT ) -1 /
   ( COUNT (*) OVER ( ws RANGE BETWEEN UNBOUNDED
   PRECEDING AND UNBOUNDED FOLLOWING ) - 1 )
END
```

『リファレンス：ビルディング・ブロック、テーブル、およびプロシージャ』の「SQL 関数」>「PERCENT_RANK 関数 [統計]」も参照してください。

ROW_NUMBER() 関数

ROW_NUMBER 関数は、ローごとにユニークなロー番号を返します。

ウィンドウ・パーティションを定義すると、**ROW_NUMBER** は、各パーティションのロー・ナンバリングを 1 から開始し、ローごとに 1 ずつ増やしていきます。ウィンドウ・パーティションを指定しない場合、**ROW_NUMBER** は、結果セット全体に対して 1 からテーブルの合計カーディナリティまでの番号を付けます。

ROW_NUMBER 関数の構文は次のとおりです。

```
ROW_NUMBER( ) OVER  ( [PARTITION BY
window partition] ORDER BY
window ordering)
```

ROW_NUMBER では引数は必須ではありませんが、カッコは指定してください。

PARTITION BY 句はオプションです。 **OVER** (**ORDER_BY**) 句に、ウィンドウ・フレーム **ROWS**/**RANGE** 指定を含めることはできません。

ランク付けの例

次は、ランク付け関数の一例です。

ランク付けの例 1 - 次の SQL クエリでは、カリフォルニア州在住の男性従業員と女性従業員を検索し、給与を基準として降順にランク付けしています。

```
SELECT Surname, Sex, Salary, RANK() OVER (
ORDER BY Salary DESC) as RANK FROM Employees
WHERE State IN ('CA') AND DepartmentID =200
ORDER BY Salary DESC;
```

このクエリの結果セットを次に示します。

```
Surname      Sex        Salary    RANK
-------      ---        ------    ----
Savarino      F          72300.000  1
Clark         F          45000.000  2
Overbey       M          39300.000  3
```

ランク付けの例 2 - 前述のクエリ例を基にして、データを性別のパーティションに分けることができます。次の例では、性別のパーティションに分けて、従業員の給与を降順にランク付けしています。

```
SELECT Surname, Sex, Salary, RANK() OVER (PARTITION BY Sex
ORDER BY Salary DESC) AS RANK FROM Employees
WHERE State IN ('CA', 'AZ') AND DepartmentID IN (200, 300)
ORDER BY Sex, Salary DESC;
```

このクエリの結果セットを次に示します。

```
Surname           Sex      Salary     RANK
-------           ---      ---------  ----
Savarino           F        72300.000  1
Jordan             F        51432.000  2
Clark              F        45000.000  3
Coleman            M        42300.000  1
Overbey            M        39300.000  2
```

ランク付けの例 3 - この例では、カリフォルニアおよびテキサスの女性従業員のリストを、給与を基準として降順にランク付けしています。**PERCENT_RANK** 関数により、累積和が降順で示されます。

```
SELECT Surname, Salary, Sex, CAST(PERCENT_RANK() OVER
(ORDER BY Salary DESC) AS numeric (4, 2)) AS RANK
FROM Employees WHERE State IN ('CA', 'TX') AND Sex ='F'
ORDER BY Salary DESC;
```

このクエリの結果セットを次に示します。

```
Surname         salary   sex           RANK
---------      ---------   ---   ----------
Savarino       72300.000   F           0.00
Smith          51411.000   F           0.33
Clark          45000.000   F           0.66
Garcia         39800.000   F           1.00
```

ランク付けの例 4 - **PERCENT_RANK** 関数を使用して、データ・セットにおける上位または下位のパーセンタイルを調べることができます。次のクエリは、給与の額がデータ・セットの上位 5% に入る男性従業員を返します。

```
SELECT * FROM (SELECT Surname, Salary, Sex,
CAST(PERCENT_RANK() OVER (ORDER BY salary DESC) as
numeric (4, 2)) AS percent
FROM Employees WHERE State IN ('CA') AND sex ='F' ) AS
DT where percent > 0.5
ORDER BY Salary DESC;
```

このクエリの結果セットを次に示します。

```
Surname          salary    sex     percent
---------      ----------    ---   ---------
Clark                          45000.000     F      1.00
```

ランク付けの例 5 - この例では、**ROW_NUMBER** 関数を使用して、すべてのウィンドウ・パーティション内の各ローについてロー番号を返しています。 このクエリでは、Employees テーブルを部署 ID ごとにパーティションに分け、各パーティションのローを開始日に基づいて順序付けします。

```
SELECT DepartmentID dID, StartDate, Salary ,
ROW_NUMBER()OVER(PARTITION BY dID ORDER BY StartDate)
FROM  Employees ORDER BY 1,2;
```

このクエリの結果セットを次に示します。

```
dID         StartDate     Salary       Row_number()
========   ===========   ==========  ==============
  100      1984-08-28    47500.000       1
  100      1985-01-01    62000.500       2
  100      1985-06-17    57490.000       3
  100      1986-06-07    72995.000       4
  100      1986-07-01    48023.690       5
  ...      ...           ...             ...
  200      1985-02-03    38500.000       1
  200      1985-12-06    54800.000       2
  200      1987-02-19    39300.000       3
  200      1987-07-10    49500.000       4
  ...      ...           ...             ...
  500      1994-02-27     24903.000      9
```

ウィンドウ集合関数

ウィンドウ集合関数を使用すると、複数のレベルの集合を 1 つのクエリで計算できます。

たとえば、支出が平均より少ない四半期をすべて列挙することができます。 集合関数 (単純な集合関数 **AVG**、**COUNT**、**MAX**、 **MIN**、**SUM** など) を使用すると、1 つの文の中でさまざまなレベルで計算した結果を 1 つのローに書き出すことができます。これにより、ジョインや相関サブクエリを使用しなくても、集合値をグループ内のディテール・ローと比較することができます。

これらの関数を使用して、非集合値と集合値を比較することも可能です。たとえば、営業部員が特定の年にある製品に対して平均以上の注文を出した顧客の一覧を作成したり、販売マネージャが従業員の給与をその部署の平均給与と比較したりすることが考えられます。

クエリが **SELECT** 文の中で **DISTINCT** を指定している場合は、ウィンドウ演算子の後に **DISTINCT** 操作が適用されます。 ウィンドウ演算子は、**GROUP BY** 句が処理された後、**SELECT** リストの項目やクエリの **ORDER BY** 句が評価される前に計算されます。

ウィンドウ集合関数の例 1 - 次のクエリは、平均販売数よりも多く売れた製品の一覧を年別に分けて示す結果セットを返します。

```
SELECT * FROM (SELECT Surname AS E_name, DepartmentID ASDept,
CAST(Salary AS numeric(10,2) ) AS Sal,CAST(AVG(Sal) OVER(PARTITION
BY DepartmentID) ASnumeric(10, 2)) AS Average,
CAST(STDDEV_POP(Sal)OVER(PARTITION BY DepartmentID) AS
numeric(10,2)) ASSTD_DEVFROM EmployeesGROUP BY Dept, E_name, Sal) AS
derived_table WHERESal> (Average+STD_DEV )ORDER BY Dept, Sal,
E_name;
```

このクエリの結果セットを次に示します。

```
 E_name    Dept                            Sal    Average   STD_DEV
--------  -----          --------  -------          --------
Lull              100        87900.00            58736.28
16829.59Sheffield          100             87900.00
58736.28            16829.59Scott               100
96300.00           58736.28           16829.59Sterling
200         64900.00             48390.94
13869.59Savarino               200         72300.00
48390.94            13869.59Kelly               200
87500.00            48390.94             13869.59Shea
300         138948.00             59500.00
30752.39Blaikie              400         54900.00
43640.67            11194.02Morris               400
61300.00            43640.67             11194.02Evans
400         68940.00             43640.67
11194.02Martinez              500         55500.80
33752.20             9084.49
```

2000 年の平均注文数は 1,787 であり、4 つの製品 (700、601、600、400) が平均を上回っています。2001 年の平均注文数は 1,048 であり、3 つの製品が平均を上回っています。

ウィンドウ集合関数の例 2 - 次のクエリは、給与の額がそれぞれの部署の平均給与よりも 1 標準偏差以上高い従業員を表す結果セットを返します。標準偏差とは、そのデータが平均からどのぐらい離れているかを示す尺度です。

```
SELECT * FROM (SELECT Surname AS E_name, DepartmentID AS
  Dept, CAST(Salary AS numeric(10,2) ) AS Sal,
  CAST(AVG(Sal) OVER(PARTITION BY dept) AS
  numeric(10, 2)) AS Average, CAST(STDDEV_POP(Sal)
  OVER(PARTITION BY dept) AS numeric(10,2)) AS
  STD_DEV
FROM Employees
GROUP BY Dept, E_name, Sal) AS derived_table WHERE
  Sal> (Average+STD_DEV )
ORDER BY Dept, Sal, E_name;
```

次の結果から、どの部署にも、給与の額が平均を大きく上回っている従業員が 1 人以上いることがわかります。

```
E_name       Dept        Sal    Average      STD_DEV
--------     ----   --------   --------     --------
Lull          100   87900.00   58736.28     16829.59
Sheffield     100   87900.00   58736.28     16829.59
Scott         100   96300.00   58736.28     16829.59
Sterling      200   64900.00   48390.94     13869.59
Savarino      200   72300.00   48390.94     13869.59
Kelly         200   87500.00   48390.94     13869.59
Shea          300  138948.00   59500.00     30752.39
Blaikie       400   54900.00   43640.67     11194.02
Morris        400   61300.00   43640.67     11194.02
Evans         400   68940.00   43640.67     11194.02
Martinez      500   55500.80   33752.20      9084.49
```

従業員 Scott の給与は 96,300.00 ドルで、所属部署の平均給与は 58,736.28 ドルです。この部署の標準偏差は 16,829.00 なので、給与の額が 75,565.88 ドル (58736.28 + 16829.60 = 75565.88) 未満ならば、平均の 1 標準偏差以内の範囲に収まります。

参照：

- *分散統計関数* (64 ページ)
- *OLAP の利点* (24 ページ)
- *OLAP の評価* (24 ページ)
- *ランク付け関数* (53 ページ)
- *統計集合関数* (59 ページ)
- *ウィンドウ* (40 ページ)
- *OLAP 関数の BNF 文法* (79 ページ)

統計集合関数

ANSI SQL/OLAP 拡張機能には、数値データの統計的分析を行うための集合関数がこの他にも数多く用意されています。これには、分散、標準偏差、相関、直線回帰を計算するための関数も含まれます。

標準偏差と分散

SQL/OLAP の一般的な関数の中には、次の構文の太字で表示されている部分のように、1 つの引数を取る関数があります。

```
<SIMPLE WINDOW AGGREGATE FUNCTION TYPE> ::=
  <BASIC AGGREGATE FUNCTION TYPE>
  | STDDEV | STDDEV_POP | STDDEV_SAMP
  | VARIANCE | VARIANCE_POP | VARIANCE_SAMP
```

- **STDDEV_POP** - グループまたはパーティションの各ロー (**DISTINCT** が指定されている場合は、重複が削除された後に残る各ロー) に対して評価される「値の式」についての母標準偏差を計算します。これは、母分散の平方根として定義されます。

- **STDDEV_SAMP** - グループまたはパーティションの各ロー (**DISTINCT** が指定されている場合は、重複が削除された後に残る各ロー) に対して評価される「値の式」についての母標準偏差を計算します。これは、標本分散の平方根として定義されます。
- **VAR_POP** - グループまたはパーティションの各ロー (**DISTINCT** が指定されている場合は、重複が削除された後に残る各ロー) に対して評価される「値の式」についての母分散を計算します。これは、「値の式」と「値の式の平均」との差の 2 乗和を、グループまたはパーティション内の残りのロー数で除算した値として定義されます。
- **VAR_SAMP** - グループまたはパーティションの各ロー (**DISTINCT** が指定されている場合は、重複が削除された後に残る各ロー) に対して評価される「値の式」についての標本分散を計算します。これは、「値の式」の差の 2 乗和を、グループまたはパーティション内の残りのロー数より 1 少ない数で除算した値として定義されます。

これらの関数と **STDDEV** および **VARIANCE** 関数は、クエリの **ORDER BY** 句の指定に従ってローのパーティションについての値を計算できる集合関数です。 **MAX** や **MIN** など、他の基本的な集合関数と同様に、これらの関数は入力データ内の NULL 値を無視します。また、分析される式のドメインに関係なく、分散と標準偏差の計算では必ず IEEE の倍精度浮動小数点数が使用されます。分散関数または標準偏差関数への入力が空のデータ・セットである場合、これらの関数は結果として NULL を返します。 **VAR_SAMP** 関数は 1 つのローに対して計算を行うと NULL を返しますが、**VAR_POP** は値 0 を返します。

相関
相関係数を計算する SQL/OLAP 関数は、次のとおりです。

- **CORR** - 一連の数値ペアの相関係数を返します。

CORR 関数は、ウィンドウ集合関数 (ウィンドウ名または指定に対するウィンドウ関数の種類を指定する関数) としても、OVER 句のない単純な集合関数としても使用できます。

共分散
共分散を計算する SQL/OLAP 関数は、次のとおりです。

- **COVAR_POP** - 一連の数値ペアの母共分散を返します。
- **COVAR_SAMP** - 一連の数値ペアの標本共分散を返します。

共分散関数は、式 1 または式 2 が null 値を持つ、すべてのペアを除外します。

共分散関数は、ウィンドウ集合関数 (ウィンドウ名または指定に対するウィンドウ関数の種類を指定する関数) としても、OVER 句のない単純な集合関数としても使用できます。

累積分布
ローのグループにおける 1 つの値の相対位置を計算する SQL/OLAP 関数は、**CUME_DIST** です。

ウィンドウ指定には ORDER_BY 句が含まれていることが必要です。

CUME_DIST 関数では、複合ソート・キーは許可されません。

回帰分析
回帰分析関数は、直線回帰の方程式を使用し、独立変数と従属変数との間の関係を計算します。SQL/OLAP の直線回帰関数は、次のとおりです。

- **REGR_AVGX** - 回帰線の独立変数の平均を計算します。
- **REGR_AVGY** - 回帰線の独立変数の平均を計算します。
- **REGR_COUNT** - 回帰線の調整に使用される NULL 値以外のペアの数を表す整数を返します。
- **REGR_INTERCEPT** - 従属変数と独立変数に最適な回帰線の y 切片を計算します。
- **REGR_R2** - 回帰線の決定係数 (適合度の統計情報) を計算します。
- **REGR_SLOPE** - NULL 以外のペアに適合する直線回帰の傾きを計算します。
- **REGR_SXX** - 直線回帰モデルに使用される独立した式の 2 乗和の合計を返します。この関数は、回帰モデルの統計的な有効性を評価するときに使用できます。
- **REGR_SXY** - 従変数および独立変数の積和を返します。この関数は、回帰モデルの統計的な有効性を評価するときに使用できます。
- **REGR_SYY** - 回帰モデルの統計的な有効性を評価できる値を返します。

回帰分析関数は、ウィンドウ集合関数 (ウィンドウ名または指定に対するウィンドウ関数の種類を指定する関数) としても、OVER 句のない単純な集合関数としても使用できます。

OLAP の加重集合関数
OLAP の加重集合関数は、加重移動平均を計算します。

- **EXP_WEIGHTED_AVG** - 指数関数的に加重された移動平均を計算します。加重は、平均を構成する各数量の相対的な重要性を決定します。**EXP_WEIGHTED_AVG** の加重は、指数関数的に減少します。 指数関数的な加重は、最新の値に加重を適用し、加重を適用しつつ、古い値の加重を減少させます。
- **WEIGHTED_AVG** - 時間経過とともに等差階級的に加重が減少した、直線的加重移動平均を計算します。 加重は、最新のデータ・ポイントの最高値から減少し、最も古いデータ・ポイントでゼロになります。

ウィンドウ指定には **ORDER_BY** 句を使用します。

データベース業界の標準外の拡張機能
データベース業界で使用される ANSI 以外の SQL/OLAP 集合関数の拡張機能には、**FIRST_VALUE**、**MEDIAN**、**LAST_VALUE** があります。

- **FIRST_VALUE** - 一連の値の最初の値を返します。
- **MEDIAN** - 式から中央値を返します。
- **LAST_VALUE** - 一連の値の最後の値を返します。

FIRST_VALUE および **LAST_VALUE** 関数には、ウィンドウ指定が必要です。
MEDIAN 関数は、ウィンドウ集合関数 (ウィンドウ名または指定に対するウィンドウ関数の種類を指定する関数) としても、OVER 句のない単純な集合関数としても使用できます。

参照：
- *分散統計関数* (64 ページ)
- *OLAP の利点* (24 ページ)
- *OLAP の評価* (24 ページ)
- *ランク付け関数* (53 ページ)
- *ウィンドウ* (40 ページ)
- *ウィンドウ集合関数* (57 ページ)
- *OLAP 関数の BNF 文法* (79 ページ)

Interrow 関数
Interrow 関数 **LAG** および **LEAD** を使用すると、一連のデータ内では前後の値にアクセスでき、テーブル内では複数のローにアクセスできます。

また、Interrow 関数は、セルフジョインなしで同時にパーティション分けします。 LAG では、テーブル内またはパーティション内の **CURRENT ROW** から特定の物理的オフセット分だけ前にあるローにアクセスできます。 LEAD では、テーブル内またはパーティション内の **CURRENT ROW** から特定の物理的オフセット分だけ後ろにあるローにアクセスできます。

LAG と **LEAD** の構文は同じです。 どちらの関数にも **OVER** (**ORDER_BY**) ウィンドウ指定が必要です。例を示します。

```
LAG (value_expr) [, offset [, default]]) OVER ([PARTITION BY
  window partition] ORDER BY
  window ordering)
```

および

```
LEAD (value_expr) [, offset [, default]]) OVER ([PARTITION
```

```
BY
            window partition] ORDER BY
            window ordering)
```

OVER (**ORDER_BY**) 句内の **PARTITION BY** 句はオプションです。 **OVER** (**ORDER_BY**) 句に、ウィンドウ・フレーム **ROWS**/**RANGE** 指定を含めることはできません。

value_expr は、テーブルから返すオフセット・データを定義するテーブル・カラムまたは式です。 *value_expr* では他の関数を定義できます。ただし、分析関数を除きます。

両方の関数について、物理的なオフセットを入力してターゲット・ローを指定します。 *offset* 値は、現在のローより上または下のロー数です。負でない数値式を入力してください (負の値を入力するとエラーが生成されます)。 0 を入力すると、Sybase IQ は現在のローを返します。

オプションの *default* 値は、*offset* 値がテーブルのスコープを超える場合に返される値を定義します。 *default* のデフォルト値は **NULL** です。 *default* のデータ型は、*value_expr* 値のデータ型に暗黙的に変換可能である必要があります。変換可能でない場合、Sybase IQ は変換エラーを生成します。

LAG の例 1 - Interrow 関数は、証券取引などのデータ・ストリームに対して計算を実行する金融サービス・アプリケーションで使用すると有用です。 この例では、**LAG** 関数を使用して、特定の株式取引価格の変化率を計算しています。 次の `stock_trades` という架空のテーブルの取引データについて考えてみます。

```
traded at             symbol   price
--------------------  ------   ------
2009-07-13 06:07:12  SQL      15.84
2009-07-13 06:07:13  TST       5.75
2009-07-13 06:07:14  TST       5.80
2009-07-13 06:07:15  SQL      15.86
2009-07-13 06:07:16  TST       5.90
2009-07-13 06:07:17  SQL      15.86
```

注意： 架空の `stock_trades` テーブルは、`iqdemo` データベースには含まれていません。

このクエリでは、銘柄記号ごとに取引をパーティションに分け、取引時間に基づいて順序付けします。また、**LAG** 関数を使用して、現在の取引価格と前の取引価格を比較し、その増加率または減少率を計算します。

```
select  stock_symbol as 'Stock',
    traded_at    as 'Date/Time of Trade',
    trade_price  as 'Price/Share',
    cast ( ( ( (trade_price
        - (lag(trade_price, 1)
        over (partition by stock_symbol
            order by traded_at)))
```

```
        / trade_price)
    * 100.0) as numeric(5, 2) )
        as '% Price Change vs Previous Price'
from stock_trades
order by 1, 2
```

このクエリは次の結果を返します。

```
Stock   Date/Time of Trade   Price/  % Price Change_vs
symbol                      Share   Previous Price
------  ------------------  -----   -----------------
SQL     2009-07-13 06:07:12  15.84   NULL
SQL     2009-07-13 06:07:15  15.86   0.13
SQL     2009-07-13 06:07:17  15.86   0.00
TST     2009-07-13 06:07:13   5.75   NULL
TST     2009-07-13 06:07:14   5.80   0.87
TST     2009-07-13 06:07:16   5.90   1.72
```

1 番目と 4 番目の出力ローの NULL は、**LAG** 関数が、2 つの各パーティションの最初のローについてはスコープ外であることを示します。 比較対象となる前のローがないため、*default* 変数で指定されているように、Sybase IQ は NULL を返します。

分散統計関数

SQL/OLAP には、順序付きセットを取り扱う関数がいくつか定義されています。

PERCENTILE_CONT および **PERCENTILE_DISC** という 2 つの逆分散統計関数があります。 これらの分析関数は、パーセンタイル値を引数として受け取り、**WITHIN GROUP** 句で指定されたデータのグループまたはデータ・セット全体に対して処理を行います。

これらの関数は、グループごとに 1 つの値を返します。 **PERCENTILE_DISC** (不連続) では、結果のデータ型は **WITHIN GROUP** 句に指定した **ORDER BY** の項目のデータ型と同じになります。 **PERCENTILE_CONT** (連続) では、結果のデータ型は、numeric (**WITHIN GROUP** 句の **ORDER BY** 項目が numeric の場合) または double (**ORDER BY** 項目が整数または浮動小数点の場合) となります。

逆分散統計関数では、**WITHIN GROUP** (**ORDER BY**) 句を指定する必要があります。 例を示します。

```
                PERCENTILE_CONT ( expression1 )
WITHIN GROUP    ( ORDER BY
                expression2 [ ASC | DESC ] )
```

expression1 の値には、numeric データ型の定数を、0 以上 1 以下の範囲で指定します。引数が NULL であれば、"wrong argument for percentile" エラーが返されます。引数の値が 0 よりも小さいか、1 よりも大きい場合は、"data value out of range" エラーが返されます。

必須の **ORDER BY** 句には、パーセンタイル関数の実行対象となる式と、各グループ内でのローのソート順を指定します。 この **ORDER BY** 句は、**WITHIN GROUP** 句の内部のみで使用するものであり、**SELECT** 文の **ORDER BY** とは*異なります*。

WITHIN GROUP 句は、クエリ結果を順序付けられたデータ・セットに分配します。関数は、このデータ・セットに基づいて結果を計算します。

値 *expression2* には、カラム参照を含む 1 つの式でソートを指定します。このソート式に、複数の式やランク付け分析関数、set 関数、またはサブクエリを指定することはできません。

ASC と DESC のパラメータでは、昇順または降順の順序付けシーケンスを指定します。昇順がデフォルトです。

逆分散統計関数は、サブクエリ、**HAVING** 句、ビュー、union で使用できます。逆分散統計関数は、統計を行わない単純な集合関数が使用されるところであれば、どこでも使用できます。逆分散統計関数は、データ・セット内の NULL 値を無視します。

PERCENTILE_CONT の例 - この例では、**PERCENTILE_CONT** 関数を使用して、各地域の自動車販売の 10 番目のパーセンタイル値を求めます。次のようなデータ・セットを使用します。

```
sales       region        dealer_name
-----       ---------     -----------
900         Northeast     Boston
800         Northeast     Worcester
800         Northeast     Providence
700         Northeast     Lowell
540         Northeast     Natick
500         Northeast     New Haven
450         Northeast     Hartford
800         Northwest     SF
600         Northwest     Seattle
500         Northwest     Portland
400         Northwest     Dublin
500         South         Houston
400         South         Austin
300         South         Dallas
200         South         Dover
```

次のクエリ例では、**SELECT** 文に **PERCENTILE_CONT** 関数を使用しています。

```
SELECT region, PERCENTILE_CONT(0.1)
WITHIN GROUP ( ORDER BY ProductID DESC )
FROM ViewSalesOrdersSales GROUP BY region;
```

この **SELECT** 文の結果には、各地域の自動車販売の 10 番目のパーセンタイル値が一覧表示されます。

```
region           percentile_cont
---------        ---------------
```

```
Canada                          601.0
Central                          700.0
Eastern                          700.0

South                           700.0
Western                          700.0
```

PERCENTILE_DISC の例 - この例では、**PERCENTILE_DISC** 関数を使用して、1 つの地域における自動車販売の 10 番目のパーセンタイル値を求めます。次のようなデータ・セットを使用します。

```
sales       region        dealer_name
-----       ---------     -----------
900         Northeast     Boston
800         Northeast     Worcester
800         Northeast     Providence
700         Northeast     Lowell
540         Northeast     Natick
500         Northeast     New Haven
450         Northeast     Hartford
800         Northwest     SF
600         Northwest     Seattle
500         Northwest     Portland
400         Northwest     Dublin
500         South         Houston
400         South         Austin
300         South         Dallas
200         South         Dover
```

次のクエリ例では、**SELECT** 文に **PERCENTILE_DISC** 関数を使用しています。

```
SELECT region, PERCENTILE_DISC(0.1) WITHIN GROUP
  (ORDER BY sales DESC )
FROM carSales GROUP BY region;
```

この **SELECT** 文の結果には、各地域の自動車販売の 10 番目のパーセンタイル値が一覧表示されます。

```
region           percentile_cont
 ---------        ---------------
Northeast        900
Northwest        800
South            500
```

分散統計関数の詳細については、『リファレンス：ビルディング・ブロック、テーブル、およびプロシージャ』の「SQL 関数」>「PERCENTILE_CONT 関数 [統計]」および「PERCENTILE_DISC 関数 [統計]」を参照してください。

参照：

- *OLAP の利点* (24 ページ)
- *OLAP の評価* (24 ページ)
- *ランク付け関数* (53 ページ)
- *統計集合関数* (59 ページ)

- *ウィンドウ* (40 ページ)
- *ウィンドウ集合関数* (57 ページ)
- *OLAP 関数の BNF 文法* (79 ページ)

数値関数

Sybase IQ でサポートされる OLAP 数値関数には、**CEILING** (エイリアスは **CEIL**)、**EXP** (エイリアスは **EXPONENTIAL**)、**FLOOR**、**LN** (エイリアスは **LOG**)、**SQRT**、**WIDTH_BUCKET** があります。

```
<numeric value function> :: =
 <natural logarithm>
| <exponential function>
| <power function>
| <square root>
| <floor function>
| <ceiling function>
| <width bucket function>
```

表 4 : 数値関数の構文

数値関数	構文
自然対数	**LN** (*numeric-expression*)
指数関数	**EXP** (*numeric-expression*)
累乗関数	**POWER** (*numeric-expression1*, *numeric-expression2*)
平方根	**SQRT** (*numeric-expression*)
床関数	**FLOOR** (*numeric-expression*)
天井関数	**CEILING** (*numeric-expression*)
等幅ヒストグラム作成関数	**WIDTH_BUCKET** (*expression*, *min_value*, *max_value*, *num_buckets*)

それぞれの数値関数の機能は次のとおりです。

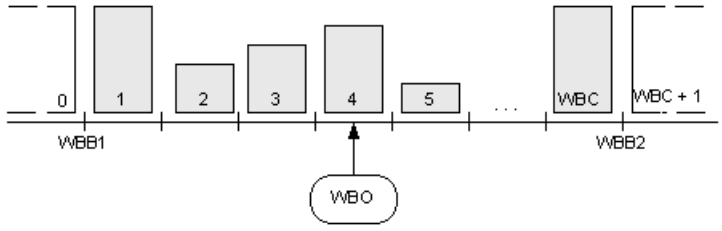
- **LN** - 引数値の自然対数を返します。引数値がゼロまたは負の場合は、エラー状態が発生します。 LN は **LOG** の同意語です。
- **EXP** - *e* (自然対数の底) の値を、引数値で指定された指数まで累乗した結果を返します。
- **POWER** - 1 つ目の引数値を、2 つ目の引数値で指定された指数まで累乗した結果を返します。両方の引数の値が 0 の場合は、1 が返されます。1 つ目の引数が 0 で、2 つ目の引数が正の値である場合は、0 が返されます。1 つ目の引数が 0 で、2 つ目の引数が負の値である場合は、例外が発生します。1 つ目の引数が負の値で、2 つ目の引数が整数でない場合は、例外が発生します。

- **SQRT** - 引数値の平方根を返します。これは、"**POWER** (*expression*, *0.5*)" の構文変換で定義されます。
- **FLOOR** - 引数の値以下で、正の無限大に最も近い整数値を返します。
- **CEILING** - 引数の値以上で、負の無限大に最も近い整数値を返します。 CEIL は CEILING の同意語です。

WIDTH_BUCKET 関数

WIDTH_BUCKET 関数は、他の数値関数よりも少し複雑です。この関数は 4 つの引数を取ります。具体的には、「目的の値」、2 つの範囲境界、そしてこの範囲を何個の等しいサイズ (または可能なかぎり等しいサイズ) の「バケット」に分割するかを指定します。 WIDTH_BUCKET 関数は、範囲の上限から下限までの差のパーセンテージに基づき、目的の値が何番目のバケットに含まれるかを示す数値を返します。最初のバケットが、バケット番号 1 となります。

目的の値が範囲境界の外にある場合のエラーを避けるために、範囲の下限よりも小さい目的の値は、先頭の補助バケット (バケット 0) に配置されます。同様に、範囲の上限よりも大きい目的の値は、末尾の補助バケット (バケット N+1) に配置されます。



たとえば、**WIDTH_BUCKET** (*14, 5, 30, 5*) は 2 を返します。処理の内容は次のとおりです。

- (30-5)/5 = 5 であるため、指定の範囲を 5 つのバケットに分割すると、各バケットの幅は 5 になります。
- 1 つ目のバケットは 0.00 ～ 19.999 …% の値、2 つ目のバケットは 20.00 ～ 39.999 …% の値を表し、以降同様に続き、5 つ目のバケットは 80.00 ～ 100.00% の値を表します。
- 目的の値を含むバケットは、(5*(14-5)/(30-5)) + 1 という計算によって算出されます。これは、バケットの総数と、指定範囲に対する「下限から目的の値までのオフセット」の比率を乗算し、それに 1 を加算するという計算です。実際の数式は (5*9/25) + 1 となり、これを計算すると 2.8 になります。 これはバケット番号 2 (2.0 ～ 2.999...) の範囲に含まれる値であるため、バケット番号 2 が返されます。

WIDTH_BUCKET 例
次の例では、サンプル・テーブル内のマサチューセッツ州の顧客の credit_limit カラムに 10 のバケット・ヒストグラムを作成し、各顧客のバケット数 (“Credit Group”) を返します。最大値を超える限度額が設定されている顧客は、オーバフロー・バケット 11 に割り当てられます。

注意： これは説明用の例であり、iqdemo データベースから生成したものではありません。

```
SELECT customer_id, cust_last_name, credit_limit,
  WIDTH_BUCKET(credit_limit, 100, 5000, 10) "Credit
  Group"
  FROM customers WHERE territory = 'MA'
  ORDER BY "Credit Group";
```

```
CUSTOMER_ID  CUST_LAST_NAME  CREDIT_LIMIT  Credit Group
-----------  --------------  ------------  ------------
825          Dreyfuss                 500             1
826          Barkin                   500             1
853          Palin                    400             1
827          Siegel                   500             1
843          Oates                    700             2
844          Julius                   700             2
835          Eastwood                1200             3
840          Elliott                 1400             3
842          Stern                   1400             3
841          Boyer                   1400             3
837          Stanton                 1200             3
836          Berenger                1200             3
848          Olmos                   1800             4
847           Streep                     5000                 11
```

範囲が逆の場合、バケットはオープン・クローズ間隔になります。たとえば、**WIDTH_BUCKET** (*credit_limit*, *5000*, *0*, *5*) のようになります。 この例では、バケット番号 1 は (4000, 5000)、バケット番号 2 は (3000, 4000)、およびバケット番号 5 は (0, 1000) です。オーバフロー・バケットには 0 (5000,+infinity) の番号が付き、アンダフロー・バケットには 6 (-infinity, 0) の番号が付きます。

参照
『リファレンス：ビルディング・ブロック、テーブル、およびプロシージャ』の「SQL 関数」>「BIT_LENGTH 関数 [文字列]」、「EXP 関数 [数値]」、「FLOOR 関数 [数値]」、「POWER 関数 [数値]」、「SQRT 関数 [数値]」、「WIDTH_BUCKET 関数 [数値]」

OLAP の規則と制限

以下では、OLAP 関数を制御する規則と制限の概要について説明します。

OLAP 関数を使用できる場合
Sybase IQ は SQL OLAP 関数を提供しますが、規則と制限が適用されます。

- **SELECT** リストの中
- 式の中
- スカラ関数の引数として
- 最後の **ORDER BY** 句の中 (クエリ内のどこかで定義されている OLAP 関数のエイリアスまたは位置参照を使用)

OLAP 関数を使用できない場合
OLAP 関数は、次の条件下では使用 *できません*。

- サブクエリの中。
- **WHERE** 句の検索条件の中。
- **SET** (集合) 関数の引数として。たとえば次の式は無効です。
  ```
  SUM(RANK() OVER(ORDER BY dollars))
  ```
- ウィンドウ集合を、他の集合に対する引数として使用することはできません (ただし、内側の集合がビューまたは抽出テーブル内で生成されたものである場合は例外です)。ランキング関数についても同じことが言えます。
- ウィンドウ集合関数と **RANK** 関数は、**HAVING** 句の中では使用できません。
- ウィンドウ集合関数に **DISTINCT** を指定しないでください。
- ウィンドウ関数を他のウィンドウ関数の内部にネストすることはできません。
- 逆分散統計関数は、**OVER** 句ではサポートされていません。
- ウィンドウ定義句では外部参照を使用できません。
- OLAP 関数内での相関参照は認められていますが、相関があるカラムのエイリアスは認められていません。

OLAP 関数から参照するカラムは、その OLAP 関数と **GROUP BY** 句が含まれている同じクエリ・ブロック内のグループ化カラムまたは集合関数である必要があります。 OLAP の処理は、グループ化操作と集合操作の後、最後の **ORDER BY** 句が適用される前に行われます。そのため、中間の結果セットから OLAP 式を導出することもできます。 クエリ・ブロック内に **GROUP BY** 句がない場合、OLAP 関数は select リスト内の他のカラムを参照できます。

Sybase IQ の制限事項
Sybase IQ で SQL OLAP 関数を使用するときの制限事項を次に示します。

- ウィンドウ・フレーム定義内ではユーザ定義関数を使用できません。
- ウィンドウ・フレーム定義内で使用する定数は、符号なし数値である必要があります。また、最大値 `BIG INT` 2^{63-1} を超えることはできません。
- ウィンドウ集合関数と **RANK** 関数は、**DELETE** 文および **UPDATE** 文では使用できません。
- ウィンドウ集合関数と **RANK** 関数は、サブクエリ内では使用できません。
- CUME_DIST は、現時点ではサポートされていません。
- グループ化セットは、現時点ではサポートされていません。
- 相関関数と直線回帰関数は、現時点ではサポートされていません。

その他の OLAP の例

この項では、OLAP 関数を使用したその他の例を紹介します。

ウィンドウの開始ポイントと終了ポイントは、中間の結果ローが処理されるときに変化する可能性があります。たとえば、累積和を計算する場合には、ウィンドウの開始ポイントは各パーティションの最初のローに固定されますが、終了ポイントは現在のローを含めるためにパーティション内のローを移動していきます。

また、ウィンドウの開始ポイントと終了ポイントの両方が可変だが、パーティション全体のローの数は一定であるという例も考えられます。このようなウィンドウを使用すると移動平均を計算するクエリを作成でき、たとえば 3 日間の株価の移動平均を返す SQL クエリを作成できます。

例： クエリ内でのウィンドウ関数

次のクエリは、2005 年の 7 月と 8 月に出荷された全製品と、出荷日別の累積出荷数を一覧にして示します。

```
SELECT p.id, p.description, s.quantity, s.shipdate,

SUM(s.quantity) OVER (PARTITION BY productid ORDER BYs.shipdate rows
between unbounded preceding andcurrent row)FROM SalesOrderItems s
JOIN Products p on(s.ProductID =p.id) WHERE s.ShipDate BETWEEN
'2001-05-01' and'2001-08-31' AND s.quantity > 40ORDER BY p.id;
```

このクエリの結果セットを次に示します。

```
ID   description   quantity   ship_date   sum quantity
---  -----------   --------   ---------   ------------
302  Crew Neck           60  2001-07-02             60
400  Cotton Cap          60  2001-05-26             60
400  Cotton Cap          48  2001-07-05            108
401  Wool cap            48  2001-06-02             48
401  Wool cap            60  2001-06-30            108
401  Wool cap            48  2001-07-09            156
500  Cloth Visor         48  2001-06-21             48
501  Plastic Visor       60  2001-05-03             60
501  Plastic Visor       48  2001-05-18            108
```

```
501  Plastic Visor       48  2001-05-25            156
501  Plastic Visor       60  2001-07-07            216
601  Zipped Sweatshirt   60  2001-07-19             60
700  Cotton Shorts       72  2001-05-18             72
700  Cotton Shorts       48  2001-05-31            120
```

この例では、2 つのテーブルのジョインとクエリの **WHERE** 句を適用した後に、**SUM** ウィンドウ関数の計算が行われます。このクエリではインラインのウィンドウ指定を使用しており、このウィンドウ指定によって、ジョインからの入力ローが次のように処理されています。

1. `prod_id` 属性の値に基づいて入力ローがパーティション (グループ) に分けられます。
2. 各パーティション内で、ローが `ship_date` 属性に基づいてソートされます。
3. パーティション内の各ローの quantity 属性について、**SUM()** 関数が評価されます。このとき、ソートされた各パーティションの最初のローから現在のローまでを含む移動ウィンドウが使用されます。

このクエリを別の方法で記述するには、関数の外でウィンドウを定義し、そのウィンドウを関数呼び出しから参照します。この方法は、同じウィンドウに基づくウィンドウ関数を複数指定する場合に便利です。このウィンドウ関数を使用するクエリを、独立したウィンドウ句を使用する方法で記述すると次のようになります (cumulative というウィンドウを宣言しています)。

```
SELECT p.id, p.description, s.quantity, s.shipdate, SUM(s.quantity)
OVER(cumulative ROWS BETWEEN UNBOUNDED PRECEDING and CURRENT ROW )
cumulative FROM SalesOrderItems s JOIN Products p On (s.ProductID
=p.id)WHERE s.shipdate BETWEEN ‘2001-07-01’ and ‘2001-08-31’Window
cumulative as (PARTITION BY s.productid ORDER BY s.shipdate)ORDER BY
p.id;
```

このクエリ指定では、ウィンドウ句が **ORDER BY** 句の前にあります。ウィンドウ句を使用するときには、次の制限が適用されます。

- インラインのウィンドウ指定に **PARTITION BY** 句を含めることはできません。
- ウィンドウ句で指定されるウィンドウに WINDOW FRAME 句 を含めることはできません。

  ```
  <WINDOW FRAME CLAUSE> ::=
    <WINDOW FRAME UNIT>
    <WINDOW FRAME EXTENT>
  ```
- インラインのウィンドウ指定にもウィンドウ句のウィンドウ指定にも WINDOW ORDER 句を含めることができますが、両方に含めることはできません。

  ```
  <WINDOW ORDER CLAUSE> ::= <ORDER SPECIFICATION>
  ```

例： 複数の関数で使用されるウィンドウ

次のクエリでは、1 つの名前付きウィンドウを定義し、そのウィンドウに基づいて複数の関数を計算できます。

```
SELECT p.ID, p.Description, s.quantity, s.ShipDate,SUM(s.Quantity)
OVER ws1, MIN(s.quantity) OVER ws1FROM SalesOrderItems s JOIN
Products p ON (s.ProductID =p.ID) WHERE s.ShipDate BETWEEN
'2000-01-09' AND'2000-01-17' AND s.Quantity > 40 window ws1
AS(PARTITION BY productid ORDER BY shipdate rowsbetween unbounded
preceding and current row)ORDER BY p.id;
```

このクエリの結果セットを次に示します。

```
ID   Description   quantity   shipDate     SUM   MIN
---  -----------   --------   -----------  ---   ---
400  Cotton Cap          48    2000-01-09   48    48
401  Wool cap            48    2000-01-09   48    48
500  Cloth Visor         60    2000-01-14   60    60
500  Cloth Visor         60    2000-01-15  120    60
501  Plastic Visor           60    2000-01-14   60    60
```

例： 累積和の計算

このクエリでは、**ORDER BY** `start_date` の順序に従って、部署別の給与の累積和を計算します。

```
SELECT dept_id, start_date, name, salary,
  SUM(salary) OVER (PARTITION BY dept_id ORDER BY
  start_date ROWS BETWEEN UNBOUNDED PRECEDING AND
  CURRENT ROW)
FROM emp1
ORDER BY dept_id, start_date;
```

このクエリの結果セットを次に示します。

```
DepartmentID  start_date    name     salary       sum

                                                    (salary)

-------                   ----------    ----   ------   ---------
100                  1996-01-01       Anna
18000            18000
100                   1997-01-01       Mike
28000    46000
100                      1998-01-01    Scott     29000    75000
100                      1998-02-01    Antonia   22000    97000
100                      1998-03-12    Adam      25000   122000
100                      1998-12-01    Amy       18000   140000
200                      1998-01-01    Jeff      18000    18000
200                      1998-01-20    Tim       29000    47000
200                      1998-02-01    Jim       22000    69000
200                      1999-01-10    Tom       28000    97000
300                      1998-03-12    Sandy     55000    55000
```

```
300                     1998-12-01    Lisa      38000    93000
300                     1999-01-10    Peter     48000    141000
```

例： 移動平均の計算

このクエリでは、連続する 3 か月間の売上の移動平均を計算します。 使用するウィンドウ・フレームのサイズは 3 つのローから成り、先行する 2 つのローと現在のローが含まれます。このウィンドウは、パーティションの最初から最後までスライドしていきます。

```
SELECT prod_id, month_num, sales, AVG(sales) OVER
  (PARTITION BY prod_id ORDER BY month_num ROWS
  BETWEEN 2 PRECEDING AND CURRENT ROW)
FROM sale WHERE rep_id = 1
ORDER BY prod_id, month_num;
```

このクエリの結果セットを次に示します。

```
prod_id   month_num      sales       avg(sales)
-------   ---------      ------      ----------
10                1        100          100.00
10                2        120          110.00
10                3        100          106.66
10                4        130          116.66
10                5        120          116.66
10                6        110          120.00
20                1         20           20.00
20                2         30           25.00
20                3         25           25.00
20                4         30           28.33
20                5         31           28.66
20                6         20           27.00
30                1         10           10.00
30                2         11           10.50
30                3         12           11.00
30                4          1            8.00
```

例：ORDER BY の結果

この例では、クエリの最上位の **ORDER BY** 句がウィンドウ関数の最終的な結果に適用されます。 ウィンドウ句に指定されている **ORDER BY** は、ウィンドウ関数の入力データに適用されます。

```
SELECT prod_id, month_num, sales, AVG(sales) OVER
  (PARTITION BY prod_id ORDER BY month_num ROWS
  BETWEEN 2 PRECEDING AND CURRENT ROW)
FROM sale WHERE rep_id = 1
ORDER BY prod_id desc, month_num;
```

このクエリの結果セットを次に示します。

```
prod_id  month_num       sales   avg(sales)
-------  ---------       -----   ----------
30               1          10       10.00
```

```
30                2           11       10.50
30                3           12       11.00
30                4            1        8.00
20                1           20       20.00
20                2           30       25.00
20                3           25       25.00
20                4           30       28.33
20                5           31       28.66
20                6           20       27.00
10                1          100      100.00
10                2          120      110.00
10                3          100      106.66
10                4          130      116.66
10                5          120      116.66
10                    6          110       120.00
```

例：1 つのクエリ内で複数の集合関数を使用

この例では、1 つのクエリ内で、異なるウィンドウに対して複数の集合値を計算しています。

```
SELECT prod_id, month_num, sales, AVG(sales) OVER
  (WS1 ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING) AS
  CAvg, SUM(sales) OVER(WS1 ROWS BETWEEN UNBOUNDED
  PRECEDING AND CURRENT ROW) AS CSum
FROM sale WHERE rep_id = 1 WINDOW WS1 AS (PARTITION BY
  prod_id
ORDER BY month_num)
ORDER BY prod_id, month_num;
```

このクエリの結果セットを次に示します。

```
prod_id  month_num    sales     CAvg     CSum
-------  ---------    -----     ----     ----
10                1      100   110.00      100
10                2      120   106.66      220
10                3      100   116.66      320
10                4      130   116.66      450
10                5      120   120.00      570
10                6      110   115.00      680
20                1       20    25.00       20
20                2       30    25.00       50
20                3       25    28.33       75
20                4       30    28.66      105
20                5       31    27.00      136
20                6       20    25.50      156
30                1       10    10.50       10
30                2       11    11.00       21
30                3       12     8.00       33
30            4        1      6.50       34
```

例： ウィンドウ・フレーム指定の ROWS と RANGE の比較

このクエリでは、ROWS と RANGE を比較しています。 **ORDER BY** 句ごとに、このデータには重複する ROWS が含まれています。

```
SELECT prod_id, month_num, sales, SUM(sales) OVER
  (ws1 RANGE BETWEEN 2 PRECEDING AND CURRENT ROW) AS
    Range_sum, SUM(sales) OVER
  (ws1 ROWS BETWEEN 2 PRECEDING AND CURRENT ROW) AS
    Row_sum
FROM sale window ws1 AS (PARTITION BY prod_id ORDER BY
  month_num)
ORDER BY prod_id, month_num;
```

このクエリの結果セットを次に示します。

```
prod_id  month_num   sales   Range_sum   Row_sum
-------  ---------   -----   ----------  -------
10               1     100         250       100
10               1     150         250       250
10               2     120         370       370
10               3     100         470       370
10               4     130         350       350
10               5     120         381       350
10               5      31         381       281
10               6     110         391       261
20               1      20          20        20
20               2      30          50        50
20               3      25          75        75
20               4      30          85        85
20               5      31          86        86
20               6      20          81        81
30               1      10          10        10
30               2      11          21        21
30               3      12          33        33
30               4       1          25        24
30               4       1          25        14
```

例： 現在のローを除外するウィンドウ・フレーム

この例では、現在のローを除外するウィンドウ・フレームを定義しています。このクエリは、現在のローを除く 4 つのローの合計を計算します。

```
SELECT prod_id, month_num, sales, sum(sales) OVER
  (PARTITION BY prod_id ORDER BY month_num RANGE
  BETWEEN 6 PRECEDING AND 2 PRECEDING)
FROM sale
ORDER BY prod_id, month_num;
```

このクエリの結果セットを次に示します。

```
prod_id   month_num       sales    sum(sales)
-------   ---------       -----    ----------
10                1         100        (NULL)
```

```
10                1          150         (NULL)
10                2          120         (NULL)
10                3          100           250
10                4          130           370
10                5          120           470
10                5           31           470
10                6          110           600
20                1           20         (NULL)
20                2           30         (NULL)
20                3           25            20
20                4           30            50
20                5           31            75
20                6           20           105
30                1           10         (NULL)
30                2           11         (NULL)
30                3           12            10
30                4            1            21
30                    4           1            21
```

例： RANGE のウィンドウ・フレーム

このクエリは、RANGE のウィンドウ・フレームを示しています。合計で使用されるローの数値は、変数です。

```
SELECT prod_id, month_num, sales, SUM(sales) OVER
  (PARTITION BY prod_id ORDER BY month_num RANGE
  BETWEEN 1 FOLLOWING AND 3 FOLLOWING)
FROM sale
ORDER BY prod_id, month_num;
```

このクエリの結果セットを次に示します。

```
prod_id   month_num       sales    sum(sales)
-------   ---------       -----    ----------
10                1         100           350
10                1         150           350
10                2         120           381
10                3         100           391
10                4         130           261
10                5         120           110
10                5          31           110
10                6         110         (NULL)
20                1          20            85
20                2          30            86
20                3          25            81
20                4          30            51
20                5          31            20
20                6          20         (NULL)
30                1          10            25
30                2          11            14
30                3          12             2
30                4           1         (NULL)
30                4           1         (NULL)
```

例： UNBOUNDED PRECEDING と UNBOUNDED FOLLOWING

この例では、パーティション内のすべてのローがウィンドウ・フレームに含まれます。このクエリは、パーティション全体 (各月に重複ローは含まれていません) における売上の最大値を計算します。

```
SELECT prod_id, month_num, sales, SUM(sales) OVER
  (PARTITION BY prod_id ORDER BY month_num ROWS
  BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING)
FROM sale WHERE rep_id = 1
ORDER BY prod_id, month_num;
```

このクエリの結果セットを次に示します。

```
prod_id   month_num       sales    SUM(sales)
-------   ---------       -----    ----------
10                1         100           680
10                2         120           680
10                3         100           680
10                4         130           680
10                5         120           680
10                6         110           680
20                1          20           156
20                2          30           156
20                3          25           156
20                4          30           156
20                5          31           156
20                6          20           156
30                1          10            34
30                2          11            34
30                3          12            34
30                4           1            34
```

このクエリは、次のクエリと同じ意味になります。

```
SELECT prod_id, month_num, sales, SUM(sales) OVER
  (PARTITION BY prod_id )
FROM sale WHERE rep_id = 1
ORDER BY prod_id, month_num;
```

例： RANGE のデフォルトのウィンドウ・フレーム

このクエリは、RANGE のデフォルトのウィンドウ・フレームの例を示しています。

```
SELECT prod_id, month_num, sales, SUM(sales) OVER
  (PARTITION BY prod_id ORDER BY month_num)
FROM sale
ORDER BY prod_id, month_num;
```

このクエリの結果セットを次に示します。

```
prod_id   month_num       sales    SUM(sales)
-------   ---------       -----    ----------
```

```
10                1          100           250
10                1          150           250
10                2          120           370
10                3          100           470
10                4          130           600
10                5          120           751
10                5           31           751
10                6          110           861
20                1           20            20
20                2           30            50
20                3           25            75
20                4           30           105
20                5           31           136
20                6           20           156
30                1           10            10
30                2           11            21
30                3           12            33
30                4            1            35
30                4            1            35
```

このクエリは、次のクエリと同じ意味になります。

```
SELECT prod_id, month_num, sales, SUM(sales) OVER
  (PARTITION BY prod_id ORDER BY month_num RANGE
  BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
FROM sale
ORDER BY prod_id, month_num;
```

OLAP 関数の BNF 文法

次の BNF (Backus-Naur Form) 文法は、さまざまな ANSI SQL 分析関数に関する具体的な構文サポートの概要を示しています。ここに記載されている関数の多くは Sybase IQ で実装されています。

文法規則 1

```
<SELECT LIST EXPRESSION> ::=
  <EXPRESSION>
  | <GROUP BY EXPRESSION>
  | <AGGREGATE FUNCTION>
  | <GROUPING FUNCTION>
  | <TABLE COLUMN>
  | <WINDOWED TABLE FUNCTION>
```

文法規則 2

```
<QUERY SPECIFICATION> ::=
  <FROM CLAUSE>
  [ <WHERE CLAUSE> ]
  [ <GROUP BY CLAUSE> ]
  [ <HAVING CLAUSE> ]
  [ <WINDOW CLAUSE> ]
[ <ORDER BY CLAUSE> ]
```

文法規則 3

```
<ORDER BY CLAUSE> ::= <ORDER SPECIFICATION>
```

文法規則 4

```
<GROUPING FUNCTION> ::=
  GROUPING <LEFT PAREN> <GROUP BY EXPRESSION>
  <RIGHT PAREN>
```

文法規則 5

```
<WINDOWED TABLE FUNCTION> ::=
  <WINDOWED TABLE FUNCTION TYPE> OVER <WINDOW NAME OR
  SPECIFICATION>
```

文法規則 6

```
<WINDOWED TABLE FUNCTION TYPE> ::=
  <RANK FUNCTION TYPE> <LEFT PAREN> <RIGHT PAREN>
  | ROW_NUMBER <LEFT PAREN> <RIGHT PAREN>
  | <WINDOW AGGREGATE FUNCTION>
```

文法規則 7

```
<RANK FUNCTION TYPE> ::=
  RANK | DENSE RANK | PERCENT RANK | CUME_DIST
```

文法規則 8

```
<WINDOW AGGREGATE FUNCTION> ::=
  <SIMPLE WINDOW AGGREGATE FUNCTION>
  | <STATISTICAL AGGREGATE FUNCTION>
```

文法規則 9

```
<AGGREGATE FUNCTION> ::=
  <DISTINCT AGGREGATE FUNCTION>
  | <SIMPLE AGGREGATE FUNCTION>
  | <STATISTICAL AGGREGATE FUNCTION>
```

文法規則 10

```
<DISTINCT AGGREGATE FUNCTION> ::=
  <BASIC AGGREGATE FUNCTION TYPE> <LEFT PAREN>
  <DISTINCT> <EXPRESSION> <RIGHT PAREN>
  | LIST <LEFT PAREN> DISTINCT <EXPRESSION>
  [ <COMMA> <DELIMITER> ]
  [ <ORDER SPECIFICATION> ] <RIGHT PAREN>
```

文法規則 11

```
<BASIC AGGREGATE FUNCTION TYPE> ::=
  SUM | MAX | MIN | AVG | COUNT
```

文法規則 12

```
<SIMPLE AGGREGATE FUNCTION> ::=
  <SIMPLE AGGREGATE FUNCTION TYPE> <LEFT PAREN>
  <EXPRESSION> <RIGHT PAREN>
  | LIST <LEFT PAREN> <EXPRESSION> [ <COMMA>
  <DELIMITER> ]
  [ <ORDER SPECIFICATION> ] <RIGHT PAREN>
```

文法規則 13

```
<SIMPLE AGGREGATE FUNCTION TYPE> ::= <SIMPLE WINDOW AGGREGATE
FUNCTION TYPE>
```

文法規則 14

```
<SIMPLE WINDOW AGGREGATE FUNCTION> ::=
  <SIMPLE WINDOW AGGREGATE FUNCTION TYPE> <LEFT PAREN>
  <EXPRESSION> <RIGHT PAREN>
| GROUPING FUNCTION
```

文法規則 15

```
<SIMPLE WINDOW AGGREGATE FUNCTION TYPE> ::=
  <BASIC AGGREGATE FUNCTION TYPE>
  | STDDEV | STDDEV_POP | STDDEV_SAMP
  | VARIANCE | VARIANCE_POP | VARIANCE_SAMP
```

文法規則 16

```
<STATISTICAL AGGREGATE FUNCTION> ::=
  <STATISTICAL AGGREGATE FUNCTION TYPE> <LEFT PAREN>
  <DEPENDENT EXPRESSION> <COMMA> <INDEPENDENT
  EXPRESSION> <RIGHT PAREN>
```

文法規則 17

```
<STATISTICAL AGGREGATE FUNCTION TYPE> ::=
  CORR | COVAR_POP | COVAR_SAMP | REGR_R2 |
  REGR_INTERCEPT | REGR_COUNT | REGR_SLOPE |
  REGR_SXX | REGR_SXY | REGR_SYY | REGR_AVGY |
  REGR_AVGX
```

文法規則 18

```
<WINDOW NAME OR SPECIFICATION> ::=
  <WINDOW NAME> | <IN-LINE WINDOW SPECIFICATION>
```

文法規則 19

```
<WINDOW NAME> ::= <IDENTIFIER>
```

文法規則 20

```
<IN-LINE WINDOW SPECIFICATION> ::= <WINDOW SPECIFICATION>
```

文法規則 21

```
<WINDOW CLAUSE> ::= <WINDOW WINDOW DEFINITION LIST>
```

文法規則 22

```
<WINDOW DEFINITION LIST> ::=
  <WINDOW DEFINITION> [ { <COMMA> <WINDOW DEFINITION>
  } . . . ]
```

文法規則 23

```
<WINDOW DEFINITION> ::=
  <NEW WINDOW NAME> AS <WINDOW SPECIFICATION>
```

文法規則 24

```
<NEW WINDOW NAME> ::= <WINDOW NAME>
```

文法規則 25

```
<WINDOW SPECIFICATION> ::=
  <LEFT PAREN> <WINDOW SPECIFICATION> <DETAILS> <RIGHT
  PAREN>
```

文法規則 26

```
<WINDOW SPECIFICATION DETAILS> ::=
  [ <EXISTING WINDOW NAME> ]
  [ <WINDOW PARTITION CLAUSE> ]
  [ <WINDOW ORDER CLAUSE> ]
  [ <WINDOW FRAME CLAUSE> ]
```

文法規則 27

```
<EXISTING WINDOW NAME> ::= <WINDOW NAME>
```

文法規則 28

```
<WINDOW PARTITION CLAUSE> ::=
  PARTITION BY <WINDOW PARTITION EXPRESSION LIST>
```

文法規則 29

```
<WINDOW PARTITION EXPRESSION LIST> ::=
  <WINDOW PARTITION EXPRESSION>
  [ { <COMMA> <WINDOW PARTITION EXPRESSION> } . . . ]
```

文法規則 30

```
<WINDOW PARTITION EXPRESSION> ::= <EXPRESSION>
```

文法規則 31

```
<WINDOW ORDER CLAUSE> ::= <ORDER SPECIFICATION>
```

文法規則 32

```
<WINDOW FRAME CLAUSE> ::=
  <WINDOW FRAME UNIT>
  <WINDOW FRAME EXTENT>
```

文法規則 33

```
<WINDOW FRAME UNIT> ::= ROWS | RANGE
```

文法規則 34

```
<WINDOW FRAME EXTENT> ::= <WINDOW FRAME START> | <WINDOW FRAME
BETWEEN>
```

文法規則 35

```
<WINDOW FRAME START> ::=
  UNBOUNDED PRECEDING
  | <WINDOW FRAME PRECEDING>
  | CURRENT ROW
```

文法規則 36

```
<WINDOW FRAME PRECEDING> ::= <UNSIGNED VALUE SPECIFICATION>
PRECEDING
```

文法規則 37

```
<WINDOW FRAME BETWEEN> ::=
  BETWEEN <WINDOW FRAME BOUND 1> AND <WINDOW FRAME
  BOUND 2>
```

文法規則 38

```
<WINDOW FRAME BOUND 1> ::= <WINDOW FRAME BOUND>
```

文法規則 39

```
<WINDOW FRAME BOUND 2> ::= <WINDOW FRAME BOUND>
```

文法規則 40

```
<WINDOW FRAME BOUND> ::=
  <WINDOW FRAME START>
```

```
  | UNBOUNDED FOLLOWING
  | <WINDOW FRAME FOLLOWING>
```

文法規則 41

```
<WINDOW FRAME FOLLOWING> ::= <UNSIGNED VALUE SPECIFICATION>
FOLLOWING
```

文法規則 42

```
<GROUP BY EXPRESSION> ::= <EXPRESSION>
```

文法規則 43

```
<SIMPLE GROUP BY TERM> ::=
  <GROUP BY EXPRESSION>
  | <LEFT PAREN> <GROUP BY EXPRESSION> <RIGHT PAREN>
  | <LEFT PAREN> <RIGHT PAREN>
```

文法規則 44

```
<SIMPLE GROUP BY TERM LIST> ::=
  <SIMPLE GROUP BY TERM> [ { <COMMA> <SIMPLE GROUP BY
  TERM> } . . . ]
```

文法規則 45

```
<COMPOSITE GROUP BY TERM> ::=
  <LEFT PAREN> <SIMPLE GROUP BY TERM>
  [ { <COMMA> <SIMPLE GROUP BY TERM> } . . . ]
  <RIGHT PAREN>
```

文法規則 46

```
<ROLLUP TERM> ::= ROLLUP <COMPOSITE GROUP BY TERM>
```

文法規則 47

```
<CUBE TERM> ::= CUBE <COMPOSITE GROUP BY TERM>
```

文法規則 48

```
<GROUP BY TERM> ::=
  <SIMPLE GROUP BY TERM>
  | <COMPOSITE GROUP BY TERM>
  | <ROLLUP TERM>
  | <CUBE TERM>
```

文法規則 49

```
<GROUP BY TERM LIST> ::=
  <GROUP BY TERM> [ { <COMMA> <GROUP BY TERM> } … ]
```

文法規則 50

```
<GROUP BY CLAUSE> ::= GROUP BY <GROUPING SPECIFICATION>
```

文法規則 51

```
<GROUPING SPECIFICATION> ::=
  <GROUP BY TERM LIST>
  | <SIMPLE GROUP BY TERM LIST> WITH ROLLUP
  | <SIMPLE GROUP BY TERM LIST> WITH CUBE
  | <GROUPING SETS SPECIFICATION>
```

文法規則 52

```
<GROUPING SETS SPECIFICATION> ::=
  GROUPING SETS <LEFT PAREN> <GROUP BY TERM LIST>
  <RIGHT PAREN>
```

文法規則 53

```
<ORDER SPECIFICATION> ::= ORDER BY <SORT SPECIFICATION LIST>
  <SORT SPECIFICATION LIST> ::= <SORT SPECIFICATION>
  [ { <COMMA> <SORT SPECIFICATION> } . . . ]
  <SORT SPECIFICATION> ::= <SORT KEY>
  [ <ORDERING SPECIFICATION> ] [ <NULL ORDERING> ]
  <SORT KEY> ::= <VALUE EXPRESSION>
  <ORDERING SPECIFICATION> ::= ASC | DESC
  <NULL ORDERING> := NULLS FIRST | NULLS LAST
```

参照：

- *分散統計関数* (64 ページ)
- *OLAP の利点* (24 ページ)
- *OLAP の評価* (24 ページ)
- *ランク付け関数* (53 ページ)
- *統計集合関数* (59 ページ)
- *ウィンドウ* (40 ページ)
- *ウィンドウ集合関数* (57 ページ)

データ・サーバとしての Sybase IQ

Sybase IQ は、ODBC または JDBC を介したクライアント・アプリケーション接続をサポートしています。 この章では、Sybase IQ をクライアント・アプリケーションのデータ・サーバとして使用する方法を説明します。

Sybase IQ は、特定のクライアント・アプリケーションに対しては、限定的ながら Open Server としても機能します。この章では、そのようなアプリケーションの作成時と実行時の制限についても簡単に説明します。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - プログラミング』>「SQL Anywhere データ・アクセス API」>「Sybase Open Client API」>「Open Client アーキテクチャ」を参照してください。

この章で説明する機能は、Windows システムや Sun Solaris システムを使用する IQ ユーザに対してリモート・データ・アクセスを提供するものではありません。 リモート・データ・アクセスは、OmniConnect™ の相互運用性機能の核であるコンポーネント統合サービス (CIS) を利用することで実現します。

Sybase IQ とのクライアント／サーバ・インタフェース

単純化するために、Sybase アプリケーションまたはサード・パーティ・アプリケーションのクライアントを Sybase IQ と共に使用してください。

これら 1 つ 1 つがどのように連携しているかを理解することは、データベースの構成やアプリケーションの設定に役立ちます。このセクションでは、これらの要素の位置付けについて説明します。 サード・パーティ製クライアント・アプリケーションの詳細については、『インストールおよび設定ガイド』を参照してください。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - データベース管理』>「レプリケーション」>「Open Server としての SQL Anywhere の使用」>「Open Client、Open Server、TDS」を参照してください。

iqdsedit による IQ Server の設定

Sybase IQ は、ネットワーク上の他の Adaptive Server、Open Server アプリケーション、クライアント・ソフトウェアと通信できます。

クライアントは 1 つ以上のサーバと通信でき、サーバはリモート・プロシージャ・コールを通して他のサーバと通信できます。製品が互いに対話するには、それぞれの製品がネットワークのどこに常駐しているかを互いに認識する必要が

あります。このネットワーク・サービス情報は interfaces ファイルに格納されています。

注意： Sybase IQ では、INSERT...LOCATION を使用可能にするために、次のような制限付き機能を提供する Open Client ユーティリティの一種が提供されています。

- iqisql
- iqdsedit
- iqdscp (UNIX のみ)
- iqocscfg (Windows のみ)

interfaces ファイル

Open Client™ プログラムを使用してデータベース・サーバに接続すると、プログラムは interfaces ファイルでサーバの名前を検索し、そのアドレスを使用してサーバに接続します。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - データベース管理』>「レプリケーション」>「Open Server としての SQL Anywhere の使用」>「Open Server の設定」>「interfaces ファイル」を参照してください。

iqdsedit データベース管理ユーティリティ

iqdsedit ユーティリティは interfaces ファイル (interfaces または SQL.ini) の設定に使用します。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - データベース管理』>「レプリケーション」>「Open Server としての SQL Anywhere の使用」>「Open Server の設定」>「DSEdit ユーティリティの使用」を参照してください。

iqdsedit の起動

Windows では、**iqdsedit** 実行可能ファイルは `%SYBASE%¥IQ-15_3¥bin32` または `%SYBASE%¥IQ-15_3¥bin64` ディレクトリにあります。これは、インストール処理でパスに自動的に追加されます。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - データベース管理』>「レプリケーション」>「Open Server としての SQL Anywhere の使用」>「Open Server の設定」>「DSEdit の起動」を参照してください。

ディレクトリ・サービスのセッションを開く

[ディレクトリ・サービスの選択] ウィンドウでは、Sybase IQ サーバなどのサーバについて、エントリを追加、修正、削除できます。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - データベース管理』>「レプリケーション」>「Open Server としての SQL Anywhere の使用」>「Open Server の設定」>「ディレクトリ・サービスのセッションを開く」を参照してください。

サーバ・エントリの追加
サーバ・エントリが [サーバ] フィールドに表示されます。サーバの属性を指定するには、エントリを修正します。

ここで入力するサーバ名は、Sybase IQ のコマンド・ラインに表示される名前に一致させる必要はありません。 サーバの識別や場所の確認には、サーバ名ではなくアドレスが使用されます。

このサーバ名フィールドは、Open Client が識別の手がかりにするためのものです。 Sybase IQ では、サーバにデータベースが 2 つ以上ロードされている場合、使用するデータベースを IQDSEDIT サーバ名エントリによって識別します。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - データベース管理』>「レプリケーション」>「Open Server としての SQL Anywhere の使用」>「Open Server の設定」>「サーバ・エントリの追加」を参照してください。

サーバ・アドレスの追加または変更
サーバ名を入力した後は、サーバ・アドレスを変更して interfaces ファイル入力を完了する必要があります。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - データベース管理』>「レプリケーション」>「Open Server としての SQL Anywhere の使用」>「Open Server の設定」>「サーバ・アドレスの追加または変更」を参照してください。

ポート番号
入力するポート番号は、Sybase IQ データベース・サーバのコマンド・ラインで指定されているポートに一致させます。 Sybase IQ サーバのデフォルトのポート番号は、2638 です。

次に示すのは有効なサーバ・アドレス・エントリの例です。

```
elora,2638
123.85.234.029,2638
```

参照：
- *Open Server としてのデータベース・サーバの起動* (92 ページ)

サーバ・アドレスの確認
Windows では、[サーバ・オブジェクト] メニューから [サーバに **Ping** を実行] コマンドを使用してネットワーク接続を確認できます。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - データベース管理』>「レプリケーション」>「Open Server としての SQL Anywhere の使用」>「Open Server の設定」>「サーバ・アドレスの確認」を参照してください。

サーバ・エントリの名前の変更

[dsedit] セッション・ウィンドウからサーバ・エントリの名前を変更できます。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - データベース管理』>「レプリケーション」>「Open Server としての SQL Anywhere の使用」>「Open Server の設定」>「サーバ・エントリ名の変更」を参照してください。

サーバ・エントリの削除

[dsedit] セッション・ウィンドウからサーバ・エントリの名前を削除できます。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - データベース管理』>「レプリケーション」>「Open Server としての SQL Anywhere の使用」>「Open Server の設定」>「サーバ・エントリの削除」を参照してください。

Sybase アプリケーションと Sybase IQ

Sybase IQ は Open Server としての役割も果たすため、OmniConnect などの Sybase アプリケーションを Sybase IQ で使用できます。

Open Client ライブラリを使用する場合、クライアント・アプリケーションではサポートされているシステム・テーブル、ビュー、ストアド・プロシージャしか使用できません。

OmniConnect サポート

Sybase OmniConnect は組織内に存在する各種の異質なデータを統一して表示し、データの内容や保管場所がわからなくても複数のデータ・ソースにアクセスできるようにします。 OmniConnect はさらに、企業全体のデータの異機種間ジョインを実行し、DB2、Sybase Adaptive Server® Enterprise、SQL Anywhere、Oracle、VSAM など、ターゲットのプラットホームを問わないテーブル・ジョインを可能にします。

Open Server インタフェースを使用すれば、Sybase IQ を OmniConnect のデータ・ソースとして利用できます。

Open Client アプリケーションと Sybase IQ

PowerSoft Power++™ のような C や C++ プログラミング環境から直接 Open Client ライブラリを使用して、Sybase IQ ベース・テーブルにあるデータにアクセスする Open Client アプリケーションを作成できます。 そのようなアプリケーションでカタログ・テーブル、ビュー、システム・ストアド・プロシージャを参照している場合、これらのオブジェクトは Adaptive Server Enterprise (Transact-SQL™ 構文) と Sybase IQ の両方でサポートされている必要があります。

『リファレンス：ビルディング・ブロック、テーブル、およびプロシージャ』の「付録 A 他の Sybase データベースとの互換性」を参照してください。

Open Client の設定

Open Client を使用して Sybase IQ に接続する場合、または **INSERT...LOCATION** 構文を使用する場合、Open Client 実行時設定 (`.cfg`) ファイルを使用して、各種の Open Client 設定パラメータを設定できます。

たとえば、**CS_MAX_CONNECT** オプションの値で制御されている最大接続数のデフォルトを変更できます。

INSERT...LOCATION のアプリケーション名は `Sybase IQ` です (単語間のスペースは必須です)。このアプリケーション名は、Open Client のコンテキスト・レベルではなく、Open Client の接続レベルで設定されます。 Open Client 実行時設定ファイルおよび使用可能なオプションの使用については、Open Client の『Client-Library/C リファレンス・マニュアル』を参照してください。

`.cfg` を有効にするには、Sybase IQ サーバを停止し、再起動します。 また、**INSERT...LOCATION** コマンド・ラインで指定できる設定パラメータもあります。 **INSERT...LOCATION** で設定されたパラメータは、設定ファイルで設定されたパラメータに置き換えられます。

リモート・サーバとして使用された場合、Sybase IQ は Tabular Data Stream (TDS) のパスワード暗号化をサポートします。 Sybase IQ サーバは、クライアントが送信した、暗号化されたパスワードのある接続を受け入れます。 パスワードの暗号化を設定するための接続プロパティについては、Open Server 15.5 の『Open Client Client-Library/C リファレンス・マニュアル』>「Client-Library トピックス」>「セキュリティ機能」>「Adaptive Server Enterprise のセキュリティ機能」>「セキュリティ・ハンドシェイク：暗号化されたパスワード」を参照してください。

注意： パスワードの暗号化には Open Client 15.0 が必要です。 TDS のパスワード暗号化では、Open Client 15.0 ESD #7 以降が必要です。

Sybase IQ サーバが、暗号化されたパスワードのある jConnect 接続を受け入れるようにするには、jConnect **ENCRYPT_PASSWORD** 接続プロパティを true に設定します。

Open Server としての Sybase IQ

ここでは、Open Client アプリケーションからの接続を受けるように Sybase IQ サーバを設定する方法について説明します。

システムの稼働条件

Sybase IQ を Open Server として使用するためには、クライアント側とサーバ側でそれぞれ稼働条件があります。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - データベース管理』>「レプリケーション」>「Open Server としての SQL Anywhere の使用」>「SQL Anywhere を Open Server として設定する」>「システムの稼働条件」を参照してください。

注意： OmniConnect を使用してローカル SQL Anywhere Enterprise サーバからリモート Sybase IQ に接続する場合は、次のサーバ・クラスを使用してください。

- Sybase IQ 12 以降に接続するには、サーバ・クラス `asaodbc` と `sajdbc` を使用します。
- Sybase IQ 11.x に接続するには、サーバ・クラス `asiq` を使用します。

Open Server としてのデータベース・サーバの起動

Sybase IQ を Open Server として使用する場合は、TCP/IP プロトコルを使用して起動してください。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - データベース管理』>「レプリケーション」>「Open Server としての SQL Anywhere の使用」>「SQL Anywhere を Open Server として設定する」>「データベース・サーバを Open Server として起動する」を参照してください。

ネットワーク・パケットが正しいアプリケーションに届くようにするために、同じマシン上で TCP/IP を使用するすべてのアプリケーションは、それぞれ別の TCP/IP *「ポート」* を使用します。 Sybase IQ のデフォルトのポートは 2638 です。これは共有メモリ通信に使用されます。 次のように、別のポート番号も指定できます。

```
start_iq -x tcpip{port=2629} -n myserver iqdemo.db
```

参照：

- *サーバ・アドレスの追加または変更* (89 ページ)

Open Client で使用するためのデータベースの設定

データベースは Sybase IQ 12.0 以降を使用してください。

Sybase IQ を Adaptive Server Enterprise と共に使用する場合は、Adaptive Server Enterprise との互換性が可能な限り高くなるように、データベースを作成してください。

Open Server としての Sybase IQ に接続すると、多くの場合、アプリケーション側は Adaptive Server Enterprise の場合と同じサービスが提供されるものと想定されます。これらのサービスは常に存在するわけではありません。

『リファレンス：ビルディング・ブロック、テーブル、およびプロシージャ』の「付録 A 他の Sybase データベースとの互換性」を参照してください。

Open Client および jConnect 接続の特性

Sybase IQ は TDS を通じてアプリケーションからの要求を処理するとき、関連するデータベース・オプションを、SQL Anywhere サーバのデフォルトの動作と互換性のある値に自動的に設定します。それらのオプションは、接続されている間のみ一時的に設定されます。クライアント・アプリケーションはこれらのオプションをいつでも独自に設定して変更できます。

注意： Sybase IQ は、`ANSI_BLANKS`、`FLOAT_AS_DOUBLE`、`TSQL_HEX_CONSTANT` の各オプションをサポートしません。

Sybase IQ では長いユーザ名とパスワードが許可されていますが、TDS クライアントのユーザ名とパスワードは最大で 30 バイトです。 パスワードまたはユーザ ID が 30 バイトを超えている場合、TDS によって (たとえば jConnect を使用して) 接続しようとすると、`Invalid user ID or password` エラーが返されます。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - データベース管理』>「レプリケーション」>「Open Server としての SQL Anywhere の使用」>「SQL Anywhere を Open Server として設定する」>「Open Client と jConnect 接続の特性」を参照してください。

注意： Interactive SQL アプリケーションなどの ODBC アプリケーションは、ODBC 仕様で要求される特定のデータベース・オプションの値を自動的に設定します。これにより、`LOGIN_PROCEDURE` データベース・オプションによる設定が上書きされます。 詳細と対処方法については、『リファレンス：文とオプション』の「LOGIN_PROCEDURE オプション」を参照してください。

複数のデータベースがあるサーバ

サーバに複数のデータベースがある場合、Open Client Library を使用して、接続するデータベースを指定できます。

- `interfaces` ファイル内にサーバのエントリを設定します。
- **start_iq** コマンドで **-n** パラメータを指定し、データベース名のショートカットを設定します。

- **isql** コマンドでデータベース名と共に **-S** *database_name* パラメータを指定します。 このパラメータは接続時に常に必要です。

プログラム自体を変更しなくても、ショートカット名をプログラムに記述し、ショートカット定義を変更するだけで、同じプログラムを複数のデータベースに対して実行できます。

たとえば、次の `live_sales` と `test_sales` の 2 つのサーバ定義は、`interfaces` ファイルから抜粋したものです。

```
live_sales

     query tcp ether myhostname 5555
     master tcp ether myhostname 5555

test_sales

     query tcp ether myhostname 7777
     master tcp ether myhostname 7777
```

サーバを起動して、所定のデータベースのエイリアスを設定します。 次のコマンドは、`live_sales` を `salesbase.db` と等価に設定します。

```
start_iq -n sales_live <other parameters> -x ¥ ‘tcpip{port=5555}’
salesbase.db -n live_sales
```

`live_sales` サーバに接続するには、次のコマンドを使用します。

```
isql -Udba -Psql -Slive_sales
```

サーバ名は `interfaces` ファイル内に一度のみ記述します。 これは、Sybase IQ への接続がデータベース名に基づくようにしており、データベース名はユニークである必要があるためです。 すべてのスクリプトが `salesbase` データベースで機能するように設定されている場合、`live_sales` または `test_sales` を使用して処理するように変更する必要はありません。

リモート・データへのアクセス

Sybase IQ では、他のサーバに置かれているデータに対しても、ローカル・データにアクセスするかのような感覚でアクセスできます。これは、アクセス先が Sybase のサーバでも Sybase 以外のサーバでも同じです。

Sybase IQ とリモート・データ

SQL Anywhere のリモート・データ・アクセスを使用すると、他のデータ・ソースのデータにアクセスできます。この機能を使用して、データを SQL Anywhere データベースに移行できます。また、複数のデータベース内のデータを問い合わせることができます。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - SQL の使用法』>「リモート・データとバルク・オペレーション」>「Open Client と jConnect 接続の特性」を参照してください。

リモート・データにアクセスするための要件

ここでは、リモート・データへのアクセスに必要な基本要素について説明します。

リモート・テーブルのマッピング

Sybase IQ は、テーブル内の全データがアプリケーションの接続先データベースに格納されているかのように、クライアント・アプリケーションにテーブルを提示します。

内部的には、Sybase IQ は、リモート・テーブルが関与するクエリを実行して記憶領域の場所を特定し、リモート・ロケーションにアクセスしてデータを取り出します。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - SQL の使用法』>「リモート・データとバルク・オペレーション」>「リモート・データへのアクセス」>「リモート・テーブルのマッピング」を参照してください。

サーバ・クラス

*サーバ・クラス*は、各リモート・サーバに割り当てられています。サーバ・クラスは、サーバとの対話に使用するアクセス方法を示すものであり、リモート・サーバの種類によって、必要とされるアクセス方法の種類も異なります。

サーバ・クラスは、Sybase IQ に詳細なサーバ機能情報を提供します。 Sybase IQ は、この機能に基づいてリモート・サーバとの対話を調節します。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - SQL の使用法』>「リモート・データとバルク・オペレーション」>「リモート・データへのアクセス」>「サーバ・クラス」を参照してください。

注意： OMNI JDBC クラスは、IPv6 ではサポートされていません。

リモート・サーバ

リモート・オブジェクトを配置するリモート・サーバを定義してから、リモート・オブジェクトをローカル・プロキシ・テーブルにマッピングします。

リモート・サーバの作成

CREATE SERVER 文を使用して、リモート・サーバの定義を設定します。

Sybase IQ や SQL Anywhere を含む一部のシステムでは、各データ・ソースが 1 つのデータベースを表すため、データベースごとにリモート・サーバの定義が必要になります。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - SQL の使用法』>「リモート・データとバルク・オペレーション」>「リモート・データへのアクセス」>「リモート・サーバの使用」>「CREATE SERVER 文を使用して、リモート・サーバを作成します。」を参照してください。

ネイティブ・クラスなしでのリモート・データのロード

DirectConnect を使用したデータのロード

ネイティブ・クラスでは、次の場所にあるリモート・データ・ソースへのアクセスに DirectConnect を使用します。

- 64 ビット UNIX プラットフォーム
- ODBC ドライバを使用できない 32 ビット・プラットフォーム (Microsoft SQL Server など)

UNIX 上の IQ サーバに MS SQL Server データをロード

このリモート・データの例では、UNIX 上の IQ サーバに MS SQL Server データをロードしています。

この例では次の状況を想定します。

- *mssql* という名前の Enterprise Connect Data Access (ECDA) サーバが UNIX ホスト *myhostname*、ポート 12530 上にあります。
- データは、ホスト *myhostname*、ポート 1433 上の MS SQL サーバ (名前は *2000*) から取り出されます。

1. DirectConnect のマニュアルに従って、DirectConnect をデータ・ソースに合わせて設定します。
2. ECDA サーバ (*mssql*) が Sybase IQ の interfaces ファイルにリストされていることを確認します。

```
mssql
master tcp ether myhostname 12530
query tcp ether myhostname 12530
```

3. *mssql* サーバのユーザ ID とパスワードを使用して、新しいユーザを追加します。

```
isql -Udba -Psql -Stst_iqdemo
grant connect to chill identified by chill
grant dba to chill
```

4. 新しいユーザとしてログインし、Sybase IQ 上にローカル・テーブルを作成します。

```
isql -Uchill -Pchill -Stst_iqdemo
create table billing(status char(1), name varchar(20), telno int)
```

5. データを挿入します。

```
insert into billing location ‘mssql.pubs’ { select * from
billing }
```

ネイティブ・クラスなしでデータのクエリを実行

ネイティブ・クラスなしでデータのクエリを実行する場合は、次のガイドラインに従います。

1. DirectConnect 経由で接続するために、ASE/CIS、リモート・サーバ、プロキシを設定します。たとえば、Oracle サーバには DirectConnect for Oracle を使用します。
2. ASEJDBC クラスを ASE サーバに使用して、Sybase IQ とリモート・サーバを設定します (ASE 用の 64 ビット Unix ODBC ドライバが存在しないため、ASEODBC クラスは利用できません)。
3. **CREATE EXISTING TABLE** 文を使用してプロキシ・テーブルを作成します。そのプロキシ・テーブルが指す ASE 内のプロキシ・テーブルを介して、最終的に Oracle を指すようにします。

UNIX 上で DirectConnect とプロキシ・テーブルを使用して、リモート・データのクエリを実行

DirectConnect を使用してデータのクエリを実行します。

この例は、MS SQL Server データへのアクセス方法を示します。この例では次の状況を想定します。

- ホスト *myhostname*、ポート 7594 上に Sybase IQ サーバが存在しています。

- ホスト *myhostname*、ポート 4101 上に Adaptive Server Enterprise サーバが存在しています。
- *mssql* という名前の Enterprise Connect Data Access (ECDA) サーバがホスト *myhostname*、ポート 12530 上にあります。
- データは、ホスト *myhostname*、ポート 1433 上の MS SQL サーバ (名前は *2000*) から取り出されます。

MS SQL Server のクエリを実行できるよう Adaptive Server Enterprise を設定

DirectConnect を介して MS SQL Server のクエリを実行できるよう、Adaptive Server とコンポーネント統合サービス (CIS) を設定します。

たとえば、サーバ名が *jones_1207* であるとします。

1. *mssql* に接続するためのエントリを ASE の interfaces ファイルに追加します。

```
mssql

master tcp ether hostname 12530

query tcp ether hostname 12530
```

2. ASE サーバでの CIS とリモート・プロシージャ・コールの処理を有効にします。たとえば、CIS がデフォルトで有効になっているとします。

```
sp_configure 'enable cis'

Parameter Name Default Memory Used Config Value Run Value

enable cis                          1         0
1             1

(1 row affected)
(return status=0)

sp_configure 'cis rpc handling', 1

Parameter Name Default Memory Used Config Value Run Value

enable cis                          0         0
0             1

(1 row affected)
Configuation option changed. The SQL Server need not be restarted
since the option is dynamic.
```

   この場合、Sybase IQ 12.5 などの古いバージョンによる CIS リモート・プロシージャ・コールの処理を有効にした後で、Adaptive Server Enterprise サーバの再起動が必要になる場合があります。

3. ASE サーバの SYSSERVERS システム・テーブルに DirectConnect サーバを追加します。

```
sp_addserver mssql, direct_connect, mssql
```

```
Adding server 'mssql', physical name 'mssql'
Server added.
(Return status=0)
```

4. ASE に接続するために Sybase IQ で使用するユーザを Adaptive Server Enterprise に作成します。

```
sp_addlogin tst, tsttst
```

```
Password correctly set.
Account unlocked. New login created.
(return status = 0)
```

```
grant role sa_role to tst
use tst_db
sp_adduser tst
```

```
New user added.
(return status = 0)
```

5. master データベースから外部ログインを追加します。

```
use master
sp_addexternlogin mssql, tst, chill, chill
```

```
User 'tst' will be known as 'chill' in remote server 'mssql'.
(return status = 0)
```

6. 目的のデータベースから追加されたユーザとして、ASE プロキシ・テーブルを作成します。

```
isql -Utst -Ttsttst
use test_db
create proxy_table billing_tst at 'mssql.pubs..billing'
select * from billing_tst
```

```
status                name                telno
------                ----------                -----
D                BOTANICALLY                1
B                BOTANICALL                2
(2 rows affected)
```

ASE サーバに接続できるよう Sybase IQ を設定

Adaptive Server Enterprise データのクエリを実行するには、次の手順に従います。

1. Sybase IQ の interfaces ファイルにエントリを追加します。

```
jones_1207
master tcp ether jones 4101
query tcp ether jones 4101
```

2. ASE への接続に使用するユーザを作成します。

```
grant connect to tst identified by tsttst
grant dba to tst
```

3. 追加されたユーザとしてログインし、'asejdbc' サーバ・クラスを作成して外部ログインを追加します。

```
isql -Utst -Ptsttst -Stst_iqdemo
create SERVER jones_1207 CLASS 'asejdbc' USING 'jones:4101/tst_db'
create existing table billing_iq at
'jones_1207.tst_db..billing_txt'
select * from billing_iq
```

```
status                name                telno
------                -----------                -----
D                BOTANICALLY                1
B                BOTANICALL                2
(2 rows affected)
```

リモート・サーバの削除

ISYSSERVER システム・テーブルからリモート・サーバを削除するには、Sybase Central または **DROP SERVER** 文を使用します。

このアクションを正しく実行するには、そのサーバ上で定義されているすべてのリモート・テーブルが削除済みであることが必要です。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - SQL の使用法』>「リモート・データとバルク・オペレーション」>「リモート・データへのアクセス」>「リモート・サーバの使用」>「リモート・サーバの削除」を参照してください。

リモート・サーバの変更

サーバの属性を変更するには、**ALTER SERVER** 文を使用します。これらの変更は、次にリモート・サーバに接続するまで有効になりません。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - SQL の使用法』>「リモート・データとバルク・オペレーション」>「リモート・データへのアクセス」>「リモート・サーバの使用」>「リモート・サーバの変更」を参照してください。

サーバ上のリモート・テーブルをリスト

Sybase IQ を設定するとき、特定のサーバで利用できるリモート・テーブルのリストがあると便利です。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - SQL の使用法』>「リモート・データとバルク・オペレーション」>「リモート・データへのアクセス」>「リモート・サーバの使用」>「サーバ上のリモート・テーブルのリスト」を参照してください。

『リファレンス：ビルディング・ブロック、テーブル、およびプロシージャ』の「システム・プロシージャ」>「sp_remote_tables システム・プロシージャ」も参照してください。

リモート・サーバ機能のリスト

sp_servercaps プロシージャは、リモート・サーバの機能についての情報を表示します。 Sybase IQ は、この機能情報に基づいて、1 つのリモート・サーバに渡すことのできる SQL 文の数を判定します。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - SQL の使用法』>「リモート・データとバルク・オペレーション」>「リモート・データへのアクセス」>「リモート・サーバの使用」>「リモート・サーバの機能のリスト」を参照してください。

『リファレンス：ビルディング・ブロック、テーブル、およびプロシージャ』の「システム・プロシージャ」>「sp_servercaps system システム・プロシージャ」も参照してください。

外部ログイン

Sybase IQ は、クライアントに代わってリモート・サーバに接続するときに、それらのクライアントの名前とパスワードを使用します。ただし、外部ログインを作成すると、この動作を無効にすることができます。

外部ログインとは、リモート・サーバとの通信時に使用される代替ログイン名とパスワードのことです。

Sybase IQ がリモート・サーバに接続するときに、**CREATE EXTERNLOGIN** でリモート・ログインが作成されており、**CREATE SERVER** 文でリモート・サーバが定義されている場合、**INSERT...LOCATION** は現在の接続のユーザ ID にリモート・ログインを使用できます。リモート・サーバが定義されていないか、現在の接続のユーザ ID に対するリモート・ログインが作成されていない場合、IQ は現在の接続のユーザ ID とパスワードを使用して接続します。 リモート・ログインを使用する **INSERT...LOCATION** の詳細と例については、『リファレンス：文とオプション』の「INSERT 文」を参照してください。

統合化ログインを使用する場合、Sybase IQ クライアントの Sybase IQ 名とパスワードは、Sybase IQ のユーザ ID が `syslogins` にマッピングしたデータベースのログイン ID およびパスワードと同じです。

外部ログインの作成

外部ログインを追加または変更できるのは、DBA アカウントまたは USER ADMIN 権限を持つアカウントのみです。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - SQL の使用法』>「リモート・データとバルク・オペレーション」>「リモート・データへのアクセス」>「リモート・サーバの使用」>「外部ログインの作成」を参照してください。

詳細については、『リファレンス：文とオプション』の「SQL 文」>「CREATE EXTERNLOGIN 文」を参照してください。

外部ログインの削除

Sybase IQ のシステム・テーブルから外部ログインを削除するには、**DROP EXTERNLOGIN** 文を使用します。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - SQL の使用法』>「リモート・データとバルク・オペレーション」>「リモート・データへのアクセス」>「リモート・サーバの使用」>「外部ログインの削除」を参照してください。

詳細については、『リファレンス：文とオプション』の「SQL 文」>「DROP EXTERNLOGIN 文」を参照してください。

プロキシ・テーブル

リモート・オブジェクトをマッピングするローカル・プロキシ・テーブルを作成すると、実際にどこにあるかは意識せずにリモート・データを扱えるようになります。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - SQL の使用法』>「リモート・データとバルク・オペレーション」>「リモート・データへのアクセス」>「プロキシ・テーブルの使用」を参照してください。

プロキシ・テーブルのロケーションの指定

CREATE TABLE と **CREATE EXISTING TABLE** のどちらの文でも、既存オブジェクトのロケーションを定義する場合は **AT** キーワードを使用します。

このロケーション文字列には 4 つの要素があり、それぞれをピリオドまたはセミコロンで区切ります。セミコロンを使用すると、database と owner の各フィールドにファイル名と拡張子を指定できます。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - SQL の使用法』>「リモート・データとバルク・オペレーション」>「リモート・データへのアクセス」>「プロキシ・テーブルの使用」>「プロキシ・テーブルのロケーションの指定」を参照してください。

例

次の例は、ロケーション文字列の使用方法を示します。

- Sybase IQ：

```
'testiq..DBA.employee'
```

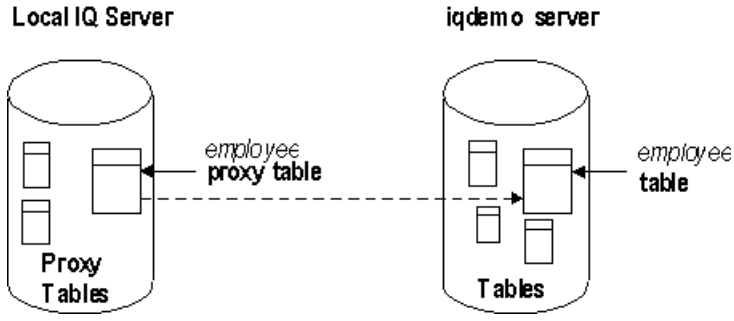
プロキシ・テーブルの作成

CREATE EXISTING TABLE 文は、リモート・サーバ上の既存テーブルをマッピングするプロキシ・テーブルを作成します。

Sybase IQ は、リモート・ロケーションのオブジェクトからカラム属性とインデックス情報を導出します。

例
サーバ `iqdemo1` 上のリモート・テーブル `employee` にマッピングするプロキシ・テーブル `p_employee` を、現在のサーバ上で作成するには、次の構文を使用します。

```
CREATE EXISTING TABLE p_employee
AT 'iqdemo1..DBA.employee'
```



『リファレンス：文とオプション』の「CREATE EXISTING TABLE 文」を参照してください。

CREATE TABLE 文

AT オプションを指定した場合、**CREATE TABLE** 文は、リモート・サーバ上に新しいテーブルを作成し、さらにそのテーブルに対するプロキシ・テーブルを定義します。

カラムは、Sybase IQ データ型を使用して定義されます。 データは Sybase IQ により、リモート・サーバのネイティブ形式に自動的に変換されます。

CREATE TABLE 文を使用してローカル・テーブルとリモート・テーブルの両方を作成し、続いて **DROP TABLE** 文を使用してプロキシ・テーブルを削除すると、リモート・テーブルも削除されます。 ただし、**DROP TABLE** 文を使用して、**CREATE EXISTING TABLE** 文で作成されたプロキシ・テーブルを削除できます。この場合は、リモート・テーブルは削除されません。

例
次に示す文は、リモート・サーバ `iqdemo1` 上にテーブル `Employees` を作成し、リモート・ロケーションにマッピングするプロキシ・テーブル `members` を作成します。

```
CREATE TABLE members
( membership_id INTEGER NOT NULL,
member_name CHAR(30) NOT NULL,
office_held CHAR( 20 ) NULL)
AT 'iqdemo1..DBA.Employees'
```

詳細については、『リファレンス：文とオプション』の「INSERT 文」を参照してください。

リモート・テーブル上のカラムをリスト

sp_remote_columns システム・プロシージャは、リモート・テーブルのカラムのリストとそれらのデータ型についての説明を生成します。

CREATE EXISTING TABLE 文を入力してカラムのリストを指定すると、リモート・テーブルで利用可能なカラムのリストを得るのに便利です。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - SQL の使用法』>「リモート・データとバルク・オペレーション」>「リモート・データへのアクセス」>「プロキシ・テーブルの使用」>「リモート・テーブルのカラムのリスト」を参照してください。

詳細については、『リファレンス：ビルディング・ブロック、テーブル、およびプロシージャ』の「システム・プロシージャ」>「システム・ストアド・プロシージャ」>「カタログ・ストアド・プロシージャ」>「sp_remote_columns システム・プロシージャ」を参照してください。

例：2 つのリモート・テーブルのジョイン

次の図は、デモ・データベースのリモート Sybase IQ テーブルの `employee` と `department` をローカル・サーバ `testiq` にマッピングしている状況を示します。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - SQL の使用法』>「リモート・データとバルク・オペレーション」>「リモート・データへのアクセス」>「リモート・テーブルのジョイン」を参照してください。

複数のローカル・データベース

Sybase IQ サーバでは、同時に複数のローカル・データベースを稼働させることができます。 他のローカル Sybase IQ データベースのテーブルをリモート・テーブルとして定義することで、データベース間のジョインを実行できます。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - SQL の使用法』>「リモート・データとバルク・オペレーション」>「リモート・データへのアクセス」>「複数のローカル・データベースのテーブルのジョイン」を参照してください。

リモート・サーバにネイティブ文を送信

FORWARD TO 文を使用して、1 つ以上の文をネイティブ構文でリモート・サーバに送信します。

この文の使用方法は、次の 2 とおりです。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - SQL の使用法』>「リモート・データとバルク・オペレーション」>「リモート・データへのアクセス」>「ネイティブ文のリモート・サーバへの送信」を参照してください。

リモート・プロシージャ・コール (RPC)

Sybase IQ ユーザは、機能をサポートしているリモート・サーバにプロシージャ・コールを発行できます。

Sybase IQ、SQL Anywhere、Adaptive Server Enterprise、Oracle、DB2 は、この機能をサポートしています。リモート・プロシージャ・コールを発行するのは、ローカル・プロシージャ・コールを使用するのと同様です。

リモート・プロシージャの作成

次のプロシージャのいずれかを使用して、リモート・プロシージャ・コールを発行します。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - SQL の使用法』>「リモート・データとバルク・オペレーション」>「リモート・データへのアクセス」>「リモート・プロシージャ・コール (RPC) の使用」>「リモート・プロシージャの作成」を参照してください。

トランザクション管理とリモート・データ

トランザクションでは、複数の SQL 文をグループ化して 1 つの単位として扱うことができます。つまり、すべての文の実行結果がデータベースにコミットされるか、実行結果が何もコミットされないかの、どちらかになります。

リモート・テーブルとローカル Sybase IQ テーブルでは、トランザクション管理の扱いに若干の違いがあります。 リモート・テーブルのトランザクション管理は、SQL Anywhere の場合とほとんど同様に扱われますが、一部に違いがあります。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - SQL の使用法』>「データベースの作成」>「トランザクションと独立性レベルの使用」を参照してください。

Sybase IQ でのトランザクションの詳細については、『システム管理ガイド：第 1 巻』の「トランザクションとバージョン管理」を参照してください。

リモート・トランザクション管理の概要

リモート・サーバが関与するトランザクションの管理には、*「2 フェーズ・コミット」*プロトコルを使用します。

Sybase IQ には、ほとんどのシナリオでトランザクションの一貫性を維持できる戦略が実装されています。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - SQL の使用法』>「リモート・データとバルク・オペレーション」>「リモート・データへのアクセス」>「トランザクションの管理とリモート・データ」>「リモート・トランザクション管理の概要」を参照してください。

トランザクション管理の制限

トランザクション管理では、セーブポイントおよびネストされた文に制限があります。

トランザクション管理の制限を次に示します。

- セーブポイントは、リモート・サーバには伝達されません。
- リモート・サーバに関わるトランザクションで、**BEGIN TRANSACTION** 文と **COMMIT TRANSACTION** 文の組がネストされている場合は、最も外側の組のみが処理されます。 最も内側の **BEGIN TRANSACTION** 文と **COMMIT TRANSACTION** 文の組は、リモート・サーバには送信されません。

内部操作

ここでは、リモート・サーバ上で SQL Anywhere がクライアント・アプリケーションに代わって実行している基本手順について説明します。

クエリの解析

クライアントから文を受け取ると、文はデータベース・サーバによって解析されます。その文が有効な SQL Anywhere SQL 文でなかった場合は、データベース・サーバでエラーが発生します。

クエリの正規化

クエリの正規化では、参照されているオブジェクトを検証し、データ型の互換性をチェックします。

たとえば、次のようなクエリがあるとします。

```
SELECT *
FROM t1
WHERE c1 = 10
```

このクエリの正規化では、`c1` カラムを持つテーブル `t1` がシステム・テーブルに存在することを検証します。 また、`c1` カラムのデータ型が値 10 と合っているかを確認します。 たとえば、このカラムのデータ型が `DATETIME` であった場合、この文は拒否されます。

クエリの前処理

クエリの前処理は、クエリの最適化の準備をします。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - SQL の使用法』>「リモート・データとバルク・オペレーション」>「リモート・データへのアクセス」>「内部オペレーション」>「クエリの前処理」を参照してください。

サーバ機能

Sybase IQ に定義されている各リモート・サーバには、それに関連付けられた一連の機能があります。 これらの機能は、syscapabilities システム・テーブルに格納されています。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - SQL の使用法』>「リモート・データとバルク・オペレーション」>「リモート・データへのアクセス」>「内部オペレーション」>「サーバの機能」を参照してください。

文の完全なパススルー

文を処理する最も効率的な方法は、元の文をなるべくそのままの形でリモート・サーバに渡すことです。

デフォルトでは、Sybase IQ はこの方針に従って文を渡そうとします。 多くの場合、Sybase IQ に渡された文は、そのままの完全な形でリモート・サーバに渡されます。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - SQL の使用法』>「リモート・データとバルク・オペレーション」>「リモート・データへのアクセス」>「内部オペレーション」>「文の完全なパススルー」を参照してください。

文の部分的なパススルー

1 つの文の中で複数のサーバが参照されている場合、または、リモート・サーバがサポートしていない SQL 機能が指定されている場合、クエリはよりシンプルな要素へと分解されます。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - SQL の使用法』>「リモート・データとバルク・オペレーション」>「リモート・データへのアクセス」>「内部オペレーション」>「文の部分的なパススルー」を参照してください。

リモート・データ・アクセスのトラブルシューティング

ここでは、リモート・サーバにアクセスするときのトラブルシューティングのヒントについて説明します。

リモート・データには使用できない機能

Sybase IQ がまったくサポートしていない機能もあれば、ローカル・データについてのみサポートしている機能もあります。

Sybase IQ では、SQL Anywhere のリストに以下が追加されています。

- Java データ型がサポートされていない。
- 特定の地域でコンポーネント統合サービス (CIS) を使用している場合、接続要求を行うと次のエラーが返される。

```
No Suitable Driver
```

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - SQL の使用法』>「リモート・データとバルク・オペレーション」>「リモート・データへのアクセス」>「リモート・データ・アクセスのトラブルシューティング」>「リモート・データに対してサポートされない機能」を参照してください。

大文字と小文字の区別

Sybase IQ データベースの大文字と小文字の設定は、アクセス先のリモート・サーバの設定と合わせる必要があります。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - SQL の使用法』>「リモート・データとバルク・オペレーション」>「リモート・データへのアクセス」>「リモート・データ・アクセスのトラブルシューティング」>「大文字と小文字の区別」を参照してください。

接続の問題

リモート・サーバに接続できることを確認するには、簡単なパススルー文を実行して、接続とリモート・ログインの設定をチェックします。

例を示します。

```
FORWARD TO testiq {select @@version}
```

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - SQL の使用法』>「リモート・データとバルク・オペレーション」>「リモート・データへのアクセス」>「リモート・データ・アクセスのトラブルシューティング」>「接続のテスト」を参照してください。

クエリ関連の一般的問題

Sybase IQ でリモート・テーブルに対するクエリを処理しているときに、何らかの問題が発生する場合、Sybase IQ が当該クエリを実行する過程を理解することで、問題の解決につながる可能性があります。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - SQL の使用法』>「リモート・データとバルク・オペレーション」>「リモート・データへのアクセス」>「リモート・データ・アクセスのトラブルシューティング」>「クエリに関する一般的な問題」を参照してください。

リモート・データ・アクセス接続の管理

ODBC 経由でリモート・データベースにアクセスするとき、リモート・サーバへの接続には名前が与えられます。

この名前は、リモート要求をキャンセルする方法の 1 つとして接続を切断するときにも使用します。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - SQL の使用法』>「リモート・データとバルク・オペレーション」>「リモート・データへのアクセス」>「ODBC を使用したリモート・データ・アクセスの接続の管理」を参照してください。

リモート・データ・アクセス用のサーバ・クラス

この章では、Sybase IQ がさまざまなサーバ・クラスとインタフェースを取る方法について説明します。

サーバ・クラスの概要

リモート接続の動作は、**CREATE SERVER** 文内のサーバ・クラスによって決定されます。 サーバ・クラスは、Sybase IQ に詳細なサーバ機能情報を提供します。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - SQL の使用法』>「リモート・データとバルク・オペレーション」>「リモート・データ・アクセスのサーバ・クラス」を参照してください。

JDBC ベースのサーバ・クラス

JDBC ベースのサーバ・クラスが使用されるのは、Sybase IQ がリモート・サーバに接続するために Java 仮想マシンと jConnect™ for JDBC™ 5.5 を内部的に使用したときです。

JDBC ベースのサーバ・クラスには次の種類があります。

- Sybase IQ と SQL Anywhere
- Sybase SQL Anywhere と Adaptive Server Enterprise (バージョン 10 以降)

JDBC クラスの設定上の注意事項

JDBC ベース・クラスで定義されたリモート・サーバにアクセスする場合、最適なパフォーマンス、リモート・サーバ・アクセス、リモート・サーバ接続を考慮してください。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - SQL の使用法』>「リモート・データとバルク・オペレーション」>「リモート・データ・アクセスのサーバ・クラス」>「JDBC ベースのサーバ・クラス」>「JDBC クラスの設定上の注意」を参照してください。

サーバ・クラス sajdbc

Sybase IQ や SQL Anywhere のデータ・ソースの設定に関しては、特別な要件はありません。

CREATE SERVER 文のパラメータ値

CREATE SERVER 文の **USING** パラメータは、*hostname:portnumber [/databasename]* という書式で指定します。

- **hostname –** リモート・サーバを実行しているマシンです。
- **portnumber –** リモート・サーバが受信している TCP/IP のポート番号です。 Sybase IQ が受信するデフォルトのポート番号は 2638 です。
- **databasename –** その接続で使用される Sybase IQ データベースです。 これは、サーバ起動時に **-n** スイッチに指定された名前、または **DBN** (**DatabaseName**) 接続パラメータに指定された名前です。

Sybase IQ の例

apple という名前のマシン上にありポート番号 2638 を受信している、`testiq` という Sybase IQ サーバを設定するには、次の文を実行します。

```
CREATE SERVER testiq
CLASS 'sajdbc'
USING 'apple:2638'
```

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - SQL の使用法』>「リモート・データとバルク・オペレーション」>「リモート・データ・アクセスのサーバ・クラス」>「JDBC ベースのサーバ・クラス」>「サーバ・クラス sajdbc」>「CREATE SERVER 文の USING パラメータ」を参照してください。

サーバ・クラス asejdbc

サーバ・クラス asejdbc を持つサーバは、Adaptive Server Enterprise または SQL Anywhere Version 10 以降のいずれかです。

通常は、Adaptive Server Enterprise データ・ソースは特別な設定を必要としませんが、ASE 15.5 は jConnect-6_0 メタデータ・ストアド・プロシージャとテーブルが必要です。 「リモート・データへのアクセス」>「リモート・データ・アクセス用のサーバ・クラス」>「JDBC ベースのサーバ・クラス」>「サーバ・クラス aseodbc」>「jConnect 6.0 メタデータのインストール」を参照してください。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - SQL の使用法』>「リモート・データとバルク・オペレーション」>「リモート・データ・アクセスのサーバ・クラス」>「JDBC ベースのサーバ・クラス」>「サーバ・クラス asejdbc」を参照してください。

データ型変換

プロキシ・テーブルを作成するために **CREATE TABLE** 文を発行すると、Sybase IQ はデータ型を対応する Adaptive Server Enterprise のデータ型に自動的に変換します。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - SQL の使用法』>「リモート・データとバルク・オペレーション」>「リモート・データ・アクセスのサーバ・クラス」>「JDBC ベースのサーバ・クラス」>「サーバ・クラス asejdbc」を参照してください。

jConnect 6.0 メタデータのインストール

Adaptive Server Enterprise 15.5 データのプロキシ・テーブルでは、jConnect 6.0 メタデータが必要です。

jConnect 6.0 メタデータがない場合、**CREATE EXISTING TABLE** 文は次のエラーを返します。`SQL Anywhere Error -667: Could not access column information for the table`.

isql の使用

1. Adaptive Server Enterprise データベースに接続します。
2. 次の形式でコマンドを入力します。

```
isql -I<path to interfaces>
-Usa -P
-S<ASE_server>
-i$SYBASE/jConnect-6_0/sp/sql_server15.0.sql
```

ODBC ベースのサーバ・クラス

Sybase IQ は各種の ODBC ベースのサーバ・クラスをサポートします。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - SQL の使用法』>「リモート・データとバルク・オペレーション」>「リモート・データ・アクセスのサーバ・クラス」>「ODBC ベースのサーバ・クラス」を参照してください。

ODBC 外部サーバ

ODBC ベースのサーバを定義する最も一般的な方法は、ODBC データ・ソースを利用する方法です。

これを実行するには、ODBC アドミニストレータでデータ・ソース名 (DSN) を作成する必要があります。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - SQL の使用法』 >「リモート・データとバルク・オペレーション」>「リモート・データ・アクセスのサーバ・

クラス」>「ODBC ベースのサーバ・クラス」>「ODBC 外部サーバの定義」を参照してください。

Sybase IQ の例
次のようにして、Sybase IQ に接続します。

```
CREATE SERVER testiq
CLASS 'asaodbc'
USING 'driver=adaptive server IQ 12.0;
eng=testasaiq;dbn=iqdemo;links=tcpip{}'
```

Sybase IQ に ODBC データ・ソースを作成する方法の詳細については、『システム管理ガイド：第 1 巻』>「Sybase IQ の接続」>「ODBC データ・ソース」を参照してください。

サーバ・クラス saodbc

複数のデータベースをサポートしている SQL Anywhere データベース サーバにアクセスするには、各データベースへの接続を定義する ODBC データ・ソース名を作成します。 これらの ODBC データ・ソース名ごとに、**CREATE SERVER** 文を発行します。

サーバ・クラス `saodbc` を持つサーバは、次のいずれかです。

- Sybase IQ バージョン 12 以降
- SQL Anywhere

SQL Anywhere や Sybase IQ のデータ・ソースの設定に関しては、特別な要件はありません。

サーバ・クラス aseodbc

Sybase IQ から `aseodbc` クラスを持つリモート Adaptive Server に接続するには、ローカルにインストールされている Adaptive Server Enterprise ODBC ドライバと Open Client コネクティビティ・ライブラリが必要です。 しかし、パフォーマンスは、`asejdbc` クラスを使った場合に比べて高くなります。

サーバ・クラス `aseodbc` を持つサーバは、次のいずれかです。

- Adaptive Server Enterprise
- SQL Anywhere (バージョン 10 以降)

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - SQL の使用法』>「リモート・データとバルク・オペレーション」>「リモート・データ・アクセスのサーバ・クラス」>「ODBC ベースのサーバ・クラス」>「サーバ・クラス aseodbc」を参照してください。

サーバ・クラス db2odbc

サーバ・クラス db2odbc を持つサーバは IBM DB2 です。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - SQL の使用法』>「リモート・データとバルク・オペレーション」>「リモート・データ・アクセスのサーバ・クラス」>「ODBC ベースのサーバ・クラス」>「サーバ・クラス db2odbc」を参照してください。

サーバ・クラス oraodbc

サーバ・クラス oraodbc を使用するサーバは、Oracle バージョン 10.0 以降です。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - SQL の使用法』>「リモート・データとバルク・オペレーション」>「リモート・データ・アクセスのサーバ・クラス」>「ODBC ベースのサーバ・クラス」>「サーバ・クラス oraodbc」を参照してください。

Sybase IQ から Oracle へのデータ型マッピング

CREATE TABLE 文を使用して Oracle サーバにリモート・テーブルを作成すると、IQ のデータ型は、対応する Oracle のデータ型に変換されます。

表 5 : 新しいリモート Oracle テーブルへのデータ・マッピング

Sybase IQ のデータ型	Oracle のデータ型
BIGINT	NUMBER(20,0)
BINARY(n)	n > 255 の場合は LONG RAW、その他の場合は RAW(n)
BIT	NUMBER(1,0)
CHAR(n)	n > 255 の場合は LONG、その他の場合は VARCHAR(n)
CHARACTER VARYING(n)	VARCHAR2(n)
CHARACTER(n)	VARCHAR2(n)
DATE	DATE
DATETIME	DATE
DECIMAL(prec, scale)	NUMBER(prec, scale)
DOUBLE	FLOAT
FLOAT	FLOAT
INT	NUMBER(11,0)

Sybase IQ のデータ型	Oracle のデータ型
LONG BINARY	LONG RAW
LONG VARCHAR	LONG または CLOB
MONEY	NUMBER(19,4)
NUMERIC(prec, scale)	NUMBER(prec, scale)
REAL	FLOAT
SMALLDATETIME	DATE
SMALLINT	NUMBER(5,0)
SMALLMONEY	NUMBER(10,4)
時刻	DATE
TIMESTAMP	DATE
TINYINT	NUMBER(3,0)
UNIQUEIDENTIFIERSTR	CHAR(36)
UNSIGNED BIGINT	NUMBER(20,0)
UNSIGNED INT	NUMBER(11,0)
UNSIGNED INTEGER	NUMBER(11,0)
VARBINARY(n)	n > 255 の場合は LONG RAW、その他の場合は RAW(n)
VARCHAR(n)	VARCHAR2(n)

Oracle から Sybase IQ へのデータ・マッピング

CREATE EXISTING 文を使用して既存の Oracle テーブルにマッピングするためのプロキシ・テーブルを作成すると、Oracle のデータ型は、対応する IQ のデータ型に変換されます。

表 6 : 既存の Oracle テーブルへのデータ・マッピング

Oracle のデータ型	IQ のデータ型
BFILE	LONG BINARY
BLOB	LONG BINARY
CHAR(n)	CHAR(n)
CLOB	LONG VARCHAR
DATE	TIMESTAMP

Oracle のデータ型	IQ のデータ型
DEC(prec, scale)	NUMERIC(prec, scale)
DECIMAL(prec, scale)	NUMERIC(prec, scale)
DOUBLE PRECISION	DOUBLE
FLOAT	DOUBLE
INT	NUMERIC(38,0)
INTEGER	NUMERIC(38,0)
NCHAR(n)	NCHAR(n)
NCLOB	LONG NVARCHAR
NUMBER(prec, scale)	NUMERIC(prec, scale)
NUMERIC(prec, scale)	NUMERIC(prec, scale)
NVARCHAR2(n)	VARCHAR(n)
RAW(n)	VARBINARY(n)
REAL	DOUBLE
SMALLINT	NUMERIC(38,0)
TIMESTAMP	TIMESTAMP
VARCHAR2(n)	VARCHAR(n)

注意：

- Sybase IQ では、プロキシ・テーブルを Oracle ビューにマップできます。 Oracle 識別子は常に大文字で表示されるため、Oracle ビューにマップしたプロキシ・テーブルを作成または参照する場合は、大文字を使用する必要があります。
- SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - SQL の使用法』>「リモート・データとバルク・オペレーション」>「リモート・データ・アクセスのサーバ・クラス」>「ODBC ベースのサーバ・クラス」>「サーバ・クラス oraodbc」を参照してください。

サーバ・クラス mssodbc

サーバ・クラス mssodbc を持つサーバは、Microsoft SQL Server バージョン 6.5、Service Pack 4 です。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - SQL の使用法』>「リモート・データとバルク・オペレーション」>「リモート・データ・アクセスのサーバ・

クラス」>「ODBC ベースのサーバ・クラス」>「サーバ・クラス mssodbc」を参照してください。

サーバ・クラス odbc

独自のサーバ・クラスを持たない ODBC データ・ソースは、odbc というサーバ・クラスを使用します。ODBC ドライバはどれでも使用できます。

Microsoft ODBC ドライバの最新バージョンは、Microsoft のダウンロード センタで配布されている Microsoft Data Access Components (MDAC) を通じて入手することができます。以下に示す Microsoft ドライバのバージョンは、MDAC 2.0 のものです。

Microsoft Excel (Microsoft 3.51.171300)

Excel の各ワークブックは複数のテーブルを持つデータベースと考えることができます。

ワークブックのシートがテーブルに相当します。 ODBC データ・ソース名を ODBC ドライバ・マネージャで設定するときは、そのデータ・ソースに関連するデフォルトのワークブック名を指定します。しかし、**CREATE TABLE** 文を発行するときは、デフォルトを無効にしてロケーション文字列でワークブック名を指定することができます。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - SQL の使用法』>「リモート・データとバルク・オペレーション」>「リモート・データ・アクセスのサーバ・クラス」>「ODBC ベースのサーバ・クラス」>「サーバ・クラス odbc」>「Microsoft Excel (Microsoft 3.51.171300)」を参照してください。

Microsoft Foxpro (Microsoft 3.51.171300)

複数の Foxpro テーブルを 1 つの Foxpro データベース・ファイル (`.dbc`) にまとめて格納することも、各テーブルを独自の `.dbf` ファイルに個別に格納することもできます。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - SQL の使用法』>「リモート・データとバルク・オペレーション」>「リモート・データ・アクセスのサーバ・クラス」>「ODBC ベースのサーバ・クラス」>「サーバ・クラス odbc」>「Microsoft FoxPro (Microsoft 3.51.171300)」を参照してください。

Lotus Notes SQL 2.0 (2.04.0203)

Lotus の Web サイトから Lotus Notes SQL 2.0 (2.04.0203) ドライバを入手できます。

Notes データがどのようにリレーショナル・テーブルにマッピングされるかについては、ドライバに付属のマニュアルを参照してください。IQ テーブルは簡単に Notes 形式にマッピングできます。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - SQL の使用法』>「リモート・データとバルク・オペレーション」>「リモート・データ・アクセスのサーバ・クラス」>「ODBC ベースのサーバ・クラス」>「サーバ・クラス odbc」>「Lotus Notes SQL 2.0」を参照してください。

Address サンプル・ファイルにアクセスできるよう IQ を設定する

Address サンプル・ファイルにアクセスできるよう、IQ を設定します。

1. NotesSQL ドライバを使用して、ODBC データ・ソースを作成します。

   データベースはサンプル名ファイル `c:¥notes¥data¥names.nsf` になります。[特殊文字のマッピング] オプションをオンにしてください。この例では、データ・ソース名は *my_notes_dsn* です。
2. IQ サーバを作成します。

   ```
   CREATE SERVER names
   CLASS 'odbc'
   USING 'my_notes_dsn'
   ```
3. Person フォームを IQ テーブルにマッピングします。

   ```
   CREATE EXISTING TABLE Person
   AT 'names...Person'
   ```
4. テーブルをクエリします。

   ```
   SELECT * FROM Person
   ```

スケジューリングとイベント処理によるタスクの自動化

この章では、Sybase IQ のスケジューリング機能とイベント処理機能を使用して、データベース管理やその他のタスクを自動化する方法について説明します。

スケジューリングとイベント処理の概要

多くのデータベース管理タスクは最適な形で体系的に実行されます。

たとえば、データベースの適切な管理では、定期バックアップの実行が重要な位置を占めています。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - データベース管理』>「データベースの保守」>「スケジュールとイベントの使用によるタスクの自動化」>「スケジュールとイベント処理の概要」を参照してください。

スケジュール

アクティビティをスケジューリングすると、あらかじめ設定しておいた時刻に確実にアクションが実行されるようにできます。スケジューリング情報とイベント・ハンドラは、どちらもデータベース自体の中に格納されます。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - データベース管理』>「データベースの保守」>「スケジュールとイベントの使用によるタスクの自動化」>「スケジュールの概要」を参照してください。

Sybase IQ の例

注意： たとえば、Sybase IQ デモ・データベース `iqdemo.db` を使用します。

```
Create table OrderSummary(c1 date, c2 int);create event
Summarizeschedulestart time '6:00 pm'on ('Mon', 'Tue', 'Wed', 'Thu',
'Fri')handlerbegin   insert into DBA.OrderSummary   select
max(OrderDate), count(*)   from GROUPO.SalesOrders where OrderDate =
current dateend
```

スケジュールの定義

柔軟性を持たせるために、スケジュールの定義は複数のコンポーネントで構成されています。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - データベース管理』>「データベースの保守」>「スケジュールとイベントの使用によるタスクの自動化」>「スケジュールの概要」>「スケジュールの定義」を参照してください。

イベント

データベース・サーバは数種類のシステム・イベントを追跡します。システム・イベントをチェックし、特定の*トリガ条件*が満たされていることがわかると、データベース・サーバはそれに対応するイベント・ハンドラをトリガします。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - データベース管理』>「データベースの保守」>「スケジュールとイベントの使用によるタスクの自動化」>「システム・イベントの概要」を参照してください。

システム・イベントの選択

Sybase IQ は複数のシステム・イベントを追跡します。各システム・イベントにはアクションを割り当てるためのフックがあります。

データベース・サーバはイベントを追跡し、必要に応じて (イベント・ハンドラに定義されているとおりに) アクションを実行します。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - データベース管理』>「データベースの保守」>「スケジュールとイベントの使用によるタスクの自動化」>「システム・イベントの概要」を参照してください。

イベントのトリガ条件の定義

各イベントの定義には、システム・イベントが割り当てられています。さらに、1 つまたは複数のトリガ条件も割り当てられています。

システム・イベントのトリガ条件が満たされると、イベント・ハンドラがトリガされます。

注意： Sybase IQ のイベントに関連付けられているトリガ条件は、SQL Anywhere や Adaptive Server Enterprise のトリガ (ユーザが指定のテーブルで指定のデータ操作文を実行しようとすると自動的に実行される) と同じではありません。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - データベース管理』>「データベースの保守」>「スケジュールとイベントの使用によるタスクの自動化」>「システム・イベントの概要」>「イベントのトリガ条件の定義」を参照してください。

Sybase IQ の例

注意： たとえば、Sybase IQ デモ・データベース `iqdemo.db` を使用します。

```
create event SecurityCheck
type ConnectFailed
handler
begindeclare num_failures int;declare mins int;

insert into FailedConnections( log_time )values ( current
timestamp );
select count( * ) into num_failuresfrom FailedConnectionswhere
log_time >= dateadd( minute, -5,
    current timestamp );if( num_failures >= 3 ) then
    select datediff( minute, last_notification,        current
timestamp ) into mins    from Notification;
    if( mins > 30 ) then        update Notification        set
last_notification = current timestamp;        call
xp_sendmail( recipient='DBAdmin',                subject='Security
Check',"message"=        'over 3 failed connections in last 5
minutes' )    end ifend ifend
```

イベント・ハンドラ

イベント・ハンドラは、イベントをトリガしたアクションとは別の接続を使って動作するので、クライアント・アプリケーションとの対話は行いません。イベント・ハンドラはイベント作成者のパーミッションで動作します。

イベント・ハンドラの開発

イベント・ハンドラは (スケジューリングしたイベント用、システム・イベントの処理用にかかわらず)、複合文を含んでおり、多くの点でストアド・プロシージャに似ています。 ループや条件付き実行などを追加できます。また、Sybase IQ デバッガを使用してイベント・ハンドラをデバッグすることもできます。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - データベース管理』>「データベースの保守」>「スケジュールとイベントの使用によるタスクの自動化」>「システム・イベントの概要」>「イベント・ハンドラの開発」を参照してください。

EVENT_PARAMETER 関数で、イベント・ハンドラのコンテキスト情報を取得できます。『リファレンス： ビルディング・ブロック、テーブル、およびプロシージャ』を参照してください。

イベント処理の使用例については、『システム管理ガイド： 第 1 巻』>「スケジューリングとイベント処理によるタスクの自動化」>「ユーザ・アカウントと接続の管理」を参照してください。

スケジュールとイベントの内容

ここでは、データベース・サーバがスケジュールとイベントの定義を処理する仕組みについて説明します。

データベース・サーバがシステム・イベントをチェックする仕組み

イベントは、**CREATE EVENT** 文に直接指定された*イベント・タイプ*に従って分類されます。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - データベース管理』>「データベースの保守」>「スケジュールとイベントの使用によるタスクの自動化」>「スケジュールとイベントの内部」>「データベース・サーバによるシステム・イベントのチェック」を参照してください。

予定時刻をデータベース・サーバがチェックする仕組み

予定イベント時刻の計算は、データベース・サーバ起動時と、スケジューリングしたイベント・ハンドラが完了するたびに行われます。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - データベース管理』>「データベースの保守」>「スケジュールとイベントの使用によるタスクの自動化」>「スケジュールとイベントの内部」>「データベース・サーバによるスケジュールされたイベントのチェック」を参照してください。

イベント・ハンドラが実行される仕組み

イベント・ハンドラがトリガされると、内部接続が一時的に確立され、その接続でイベント・ハンドラが実行されます。

イベント・ハンドラは、それ自体のトリガ元となった接続の上では実行されません。したがって、クライアント・アプリケーションとの対話を伴う **MESSAGE ... TO CLIENT** などの文をイベント・ハンドラ内に書いても無意味です。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - データベース管理』>「データベースの保守」>「スケジュールとイベントの使用によるタスクの自動化」>「イベント処理タスク」>「データベースへのイベントの追加」を参照してください。

スケジューリングとイベント処理のタスク

ここでは、スケジュールとイベントの自動化に関連するタスクについて説明します。

スケジュールやイベントのデータベースへの追加

スケジュールやイベントは、Sybase Central で SQL を使って追加できます。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - データベース管理』>「データベースの保守」>「スケジュールとイベントの使用によるタスクの自動化」>「イベント処理タスク」>「データベースへのイベントの追加」を参照してください。

手動トリガ・イベントのデータベースへの追加

トリガ元となるスケジュールやシステム・イベントを持たないイベント・ハンドラは、手動でトリガしないかぎり実行されません。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - データベース管理』>「データベースの保守」>「スケジュールとイベントの使用によるタスクの自動化」>「イベント処理タスク」>「データベースへの手動トリガ・イベントの追加」を参照してください。

ALTER EVENT 文を使用してイベントを変更します。『リファレンス： 文とオプション』を参照してください。

イベント・ハンドラのトリガ

どのイベント・ハンドラも、スケジュールやシステム・イベントによっても実行されますが、手動でもトリガできます。イベントの手動トリガは、イベント・ハンドラの開発時に役立つだけでなく、イベントによっては運用環境においても役立つ場合があります。

たとえば、月次売り上げ報告の作成がスケジューリングされているとします。しかし、売り上げ報告が必要なのは月末だけではありません。別の目的で中間報告が必要な場合もあります。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - データベース管理』>「データベースの保守」>「スケジュールとイベントの使用によるタスクの自動化」>「イベント処理タスク」>「イベント・ハンドラのトリガ」を参照してください。

トリガの詳細については、『リファレンス：文とオプション』の `TRIGGER EVENT` 文に関する説明を参照してください。

イベント・ハンドラのデバッグ

デバッグはソフトウェア開発にはつきものの作業です。イベント・ハンドラは開発過程の中でデバッグできます。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - データベース管理』>「データベースの保守」>「スケジュールとイベントの使用によるタスクの自動化」>「イベント処理タスク」>「イベント・ハンドラのデバッグ」を参照してください。

イベントやスケジュールに関する情報の取得

Sybase IQ は、イベント、システム・イベント、スケジュールに関する情報を、SYSEVENT、SYSEVENTTYPE、SYSSCHEDULE の各システム・テーブルに保存しています。

ALTER EVENT 文を使用してイベントを変更するとき、イベント名のほかに、オプションでスケジュール名を指定します。 **TRIGGER EVENT** 文を使用してイベントをトリガするには、イベント名を指定します。

イベント名は、システム・テーブル SYSEVENT をクエリして、一覧表示できます。次に例を示します。

```
SELECT event_id, event_name FROM SYSEVENT
```

スケジュール名は、システム・テーブル SYSSCHEDULE をクエリして、一覧表示できます。次に例を示します。

```
SELECT event_id, sched_name FROM SYSSCHEDULE
```

イベントには固有のイベント ID が付いています。イベントと関連するスケジュールの対応付けには、SYSEVENT と SYSSCHEDULE の event_id カラムを使用します。

JDBC を使用したデータ・アクセス

この付録では、JDBC を使用してデータにアクセスする方法について説明します。

JDBC は、クライアント・アプリケーションから使用できるだけでなく、データベース内でも使用できます。JDBC を使用する Java クラスは、データベースにプログラミング・ロジックを組み込む方法として、SQL ストアド・プロシージャに代わる一層強力な選択肢となります。

JDBC の概要

JDBC は Java アプリケーションに対して SQL インタフェースを提供します。Java からリレーショナル・データにアクセスする場合は、JDBC 呼び出しを使用します。

この付録は、JDBC データベース・インタフェースの細部にわたる網羅的な解説ではありません。ここでは、いくつかの簡単な例を示しながら JDBC の機能を紹介し、サーバ内とサーバ外における JDBC の使用方法を具体的に説明します。さらに、データベース・サーバ内で JDBC を実行するサーバ側での使用については、より詳細に説明します。

具体例では、Sybase IQ において JDBC を使用する場合に特徴的な機能を示します。JDBC プログラミングの詳細については、JDBC プログラミング関連の書籍を参照してください。

JDBC と Sybase IQ

Sybase IQ では、次の 2 つの方法で JDBC を使用できます。

- Java クライアント・アプリケーションは、Sybase IQ に対して JDBC 呼び出しを実行できます。サーバへの接続には Sybase jConnect JDBC ドライバまたは iAnywhere JDBC ドライバが使用されます。
  この付録では、ユーザのマシン上で実行されているアプリケーションと、中間層のアプリケーション・サーバ上で実行されているロジックの両方を「*クライアント・アプリケーション*」と呼びます。
- データベースにインストールされたサーバ側 Java クラスの JDBC は、内部 JDBC ドライバを使用して、JDBC 呼び出しによるデータ・アクセスとデータ変更を実行できます。

この付録では、サーバ側の JDBC に重点を置いて説明します。

JDBC リソース

- 必須ソフトウェア
  Sybase jConnect ドライバを使用するには TCP/IP が必要です。
  Sybase IQ のインストール状況によっては、現状ですでに Sybase jConnect ドライバが使用可能な場合があります。

参照：

- *jConnect ドライバのファイル* (149 ページ)

JDBC ドライバの選択

Sybase IQ では、次の 2 種類の JDBC ドライバが提供されています。

表 7 :

ドライバ	定義
jConnect	jConnect は、100% pure Java ドライバです。 このドライバは Sybase IQ との通信に TDS クライアント／サーバ・プロトコルを使用します。
iAnywhere JDBC ドライバ	このドライバは Sybase IQ との通信に Command Sequence クライアント／サーバ・プロトコルを使用します。 このドライバでは、ODBC、埋め込み SQL、OLEDB のアプリケーションとの動作の一貫性が維持されています。

jConnect のマニュアルについては、『jConnect for JDBC』を参照してください。

使用するドライバは、次の要因を考慮して選択してください。

- 機能—どちらのドライバも JDK 2 に準拠しています。iAnywhere JDBC では自由にスクロール可能なカーソルがサポートされていますが、jConnect ではこの機能は使用できません。
- Pure Java—jConnect ドライバは、pure Java ソリューションです。 iAnywhere JDBC ドライバは Sybase IQ または Adaptive Server の Anywhere ODBC ドライバを必要とするため、pure Java ソリューションではありません。
- パフォーマンス—ほとんどの用途において、パフォーマンスは iAnywhere JDBC ドライバの方が jConnect ドライバより優れています。
- 互換性—jConnect ドライバで使用される TDS プロトコルは、Adaptive Server Enterprise と共通のプロトコルです。 このドライバの動作の一部はこのプロトコルによって制御され、Adaptive Server Enterprise と互換性を持つように設定されます。

どちらのドライバも Windows 95/98/Me と Windows NT/2000/2003/XP で使用できるだけでなく、UNIX と Linux のオペレーティング・システムでもサポートされています。

JDBC の考慮事項
Java アプリケーションを実行するときは、次のことを考慮してください。

- iAnywhere JDBC ドライバを使用して dbisql Java で Sybase IQ 12.5 サーバに接続するときは問題があります。 詳細は、『システム管理ガイド： 第 1 巻』>「トラブルシューティングのヒント」>「データ・トランケーションまたはデータ変換エラー」を参照してください。
- Java アプリケーションを Sybase IQ で実行すると、外部の Sun Java 仮想マシン (JVM) で実行するときよりも速度が遅くなります。 このような制約はありますが、`JAVA_HEAP_SIZE` および `JAVA_NAMESPACE_SIZE` データベース・オプションで IQ JVM の使用可能メモリを増やし、アプリケーションをチューニングすることをおすすめします。『リファレンス： 文とオプション』の「データベース・オプション」>「JAVA_HEAP_SIZE」を参照してください。

JDBC プログラムの構造

JDBC プログラムの構造には、イベントのシーケンスが含まれています。

典型的な JDBC アプリケーションでは、次のような順序で処理が実行されます。

- Connection オブジェクトの作成—**DriverManager** クラスの **getConnection** クラス・メソッドは、**Connection** オブジェクトを作成し、データベースとの接続を確立します。
- Statement オブジェクトの生成—**Connection** オブジェクトは、**Statement** オブジェクトを生成します。
- SQL 文の引き渡し—データベース環境内で実行された SQL 文が **Statement** オブジェクトに渡されます。 渡される SQL 文がクエリである場合は、このアクションによって **ResultSet** オブジェクトが返されます。

  ResultSet オブジェクトには SQL 文から返されたデータが格納されていますが、アプリケーションからは一度に 1 つのローのデータしか扱えません (カーソルの動きと同じです)。
- 結果セット内のローを順次処理するループ—**ResultSet** オブジェクトの **next** メソッドは、次の 2 つのアクションを実行します。
  - 現在のロー (結果セットのローのうち、**ResultSet** オブジェクトにおける現在の処理対象となっているロー) を次のローに進めます。
  - 進んだ位置に実際にローが存在するかどうかを示す Boolean 値 (true/false) を返します。

- 各ローにおける値の取得—値は、カラムの名前または位置のいずれかを特定することによって、**ResultSet** オブジェクトの各カラムにおいて取得されます。現在のローのカラムから日付の値を取得するには、**getDate** メソッドを使用します。

Java オブジェクトは、JDBC オブジェクトを使用することによって、データベースとの対話的操作とデータの取得を実行できます。取得したデータは、データの操作や他のクエリでの使用など、さまざまな目的で各オブジェクトが独自に使用できます。

サーバ側 JDBC の機能

JDBC 1.2 は JDK 1.1 に含まれています。JDBC 2.0 は Java 2 (JDK 1.2) に含まれています。

データベースで使用されている Java は JDK バージョン 1.1 のサブセットであるため、内部 JDBC ドライバでは JDBC バージョン 1.2 がサポートされています。

内部 JDBC ドライバ (`asajdbc`) では、JDBC 2.0 の一部の機能をサーバ側の Java アプリケーションから使用できますが、JDBC 2.0 は完全にはサポートされていません。

データベースで使用されている Java の **java.sql** パッケージに含まれている JDBC クラスは、バージョン 1.2 レベルのクラスです。 サーバ側における JDBC 2.0 レベルの機能は **sybase.sql.ASA** パッケージで実装されています。 JDBC 2.0 の機能を使用するには、JDBC オブジェクトのキャスト時に、**java.sql** パッケージではなく、**sybase.sql.ASA** パッケージの対応クラスにキャストする必要があります。 **java.sql** として宣言されたクラスの機能は、JDBC 1.2 の機能のみに限定されます。

sybase.sql.ASA のクラスは次のとおりです。

JDBC クラス	Sybase 内部ドライバクラス
java.sql.Connection	sybase.sql.ASA.SAConnection
java.sql.Statement	sybase.sql.ASA.SAStatement
java.sql.PreparedStatement	sybase.sql.ASA.SAPreparedStatement
java.sql.CallableStatement	sybase.sql.ASA.SACallableStatement
java.sql.ResultSetMetaData	sybase.sql.ASA.SAResultSetMetaData
java.sql.ResultSet	sybase.sql.SAResultSet
java.sql.DatabaseMetaData	sybase.sql.SADatabaseMetaData

次の関数は、その準備文を実行した場合に得られる結果の **ResultSetMetaData** オブジェクトを返します。実際に文を実行する必要はなく、**ResultSet** も不要です。この関数は、JDBC 標準には含まれていません。

```
ResultSetMetaData sybase.sql.ASA.SAPreparedStatement.describe()
```

JDBC 2.0 の制限
次のクラスは JDBC 2.0 コア・インタフェースに含まれていますが、**sybase.sql.ASA** パッケージでは使用できません。

- java.sql.Blob
- java.sql.Clob
- java.sql.Ref
- java.sql.Struct
- java.sql.Array
- java.sql.Map

次の JDBC 2.0 コア関数は、**sybase.sql.ASA** パッケージでは使用できません。

sybase.sql.ASA のクラス	提供されていない関数
SAConnection	java.util.Map getTypeMap() void setTypeMap(java.util.Map map)
SAPreparedStatement	void setRef(int pidx, java.sql.Ref r) void setBlob(int pidx, java.sql.Blob b) void setClob(int pidx, java.sql.Clob c) void setArray(int pidx, java.sql.Array a)
SACallableStatement	Object getObject(pidx, java.util.Map map) java.sql.Ref getRef(int pidx) java.sql.Blob getBlob(int pidx) java.sql.Clob getClob(int pidx) java.sql.Array getArray(int pidx)

sybase.sql.ASA のクラス	提供されていない関数
SAResultSet	Object getObject(int cidx, java.util.Map map)
	java.sql.Ref getRef(int cidx)
	java.sql.Blob getBlob(int cidx)
	java.sql.Clob getClob(int cidx)
	java.sql.Array getArray(int cidx)
	Object getObject(String cName, java.util.Map map)
	java.sql.Ref getRef(String cName)
	java.sql.Blob getBlob(String cName)
	java.sql.Clob getClob(String cName)
	java.sql.Array getArray(String cName)

クライアント側 JDBC 接続とサーバ側 JDBC 接続の違い

クライアント側の JDBC とデータベース・サーバ側の JDBC の違いは、データベース環境との接続の確立方法にあります。

- クライアント側—クライアント側の JDBC で接続を確立するには、Sybase jConnect JDBC ドライバが必要です。 **DriverManager.getConnection** に引数を渡すことによって、接続が確立されます。クライアント・アプリケーションから見た場合、データベース環境は 1 つの外部アプリケーションと見なされます。
- サーバ側—データベース・サーバ内で JDBC を使用する場合、接続はすでに存在しています。 **jdbc:default:connection** の値が **DriverManager.getConnection** に渡され、それにより、JDBC アプリケーションは、現在のユーザ接続内で操作を実行できるようになります。これは高速で効率的な処理であり、また、クライアント・アプリケーションは接続を確立する段階ですでにデータベース・セキュリティの認証に合格しているので安全でもあります。すでに認証されたユーザ ID とパスワードを再び入力する必要はありません。 asajdbc ドライバで接続できるのは、現在の接続のデータベースに対してのみです。

JDBC クラスでは、ソース・コードにおいて 1 つの条件文を使用して URL の構築方法を使い分けることにより、クライアント側とサーバ側の両方で実行可能なクラスを作成できます。 外部接続にはマシン名とポート番号の指定が必要です。内部接続の場合は **jdbc:default:connection** を指定する必要があります。

JDBC 接続の確立

この項では、Java アプリケーションから JDBC データベース接続を確立するクラスについて説明します。

JDBC クライアント・アプリケーションからの jConnect による接続

JDBC アプリケーションからデータベースのシステム・テーブル (データベース・メタデータ) にアクセスする場合は、jConnect システム・オブジェクトのセットをデータベースに追加する必要があります。

JDBC アプリケーションからデータベースのシステム・テーブル (データベース・メタデータ) にアクセスする場合は、jConnect システム・オブジェクトのセットをデータベースに追加する必要があります。Asajdbc と jConnect は、データベース・メタデータをサポートするための同じストアド・プロシージャを共有しています。これらのプロシージャは、デフォルトですべてのデータベースにインストールされます。 このインストールが実行されないようにするには、**iqinit** において **-i** スイッチを指定します。

-i スイッチは、**iqinit** および SQL Anywhere のユーティリティ **dbinit** に共通です。 **-i** スイッチの詳細については、SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - データベース管理』>「データベースの管理」>「データベース管理ユーティリティ」>「初期化ユーティリティ (dbinit)」を参照してください。

次に例として示す完全な Java アプリケーションは、コマンド・ライン・アプリケーションです。このアプリケーションは、実行中のデータベースに接続し、取得した情報をコマンド・ラインに出力して、終了します。

接続の確立は、データベースのデータを操作する場合にすべての JDBC アプリケーションが最初に実行する不可欠のステップです。

参照：

- *サーバ側 JDBC クラスからの接続の確立* (137 ページ)
- *Sybase jConnect JDBC ドライバ* (148 ページ)
- *外部接続サンプル・アプリケーションの実行* (136 ページ)
- *分散アプリケーションの例* (155 ページ)

外部接続のサンプル・コード

接続を確立するために使用するメソッドのソース・コードです。

main メソッドと **ASAConnect** メソッドのソース・コードは、Sybase IQ インストール・ディレクトリの下の `C:¥Documents and Settings¥All Users¥SybaseIQ¥samples¥SQLAnywhere¥JDBC` ディレクトリ (Windows) または

$SYBASE/IQ-15_3/samples/sqlanywhere/JDBC (UNIX) にある JDBCExamples.java ファイルに記述されています。

```
// Import the necessary classes
import java.sql.*;             // JDBC
import com.sybase.jdbc.*;      // Sybase jConnect
import java.util.Properties;   // Properties
import sybase.sql.*;           // Sybase utilities
import asademo.*;              // Example classes

private static Connection conn;
public static void main(String args[])  {

    conn = null;
    String machineName;
    if ( args.length != 1 ) {
      machineName = "localhost";
    } else {
      machineName = new String( args[0] );
    }

    ASAConnect( "dba", "sql", machineName );
    if( conn!=null ) {
        System.out.println( "Connection successful" );
    }else{
        System.out.println( "Connection failed" );
    }

    try{
      serializeVariable();
      serializeColumn();
      serializeColumnCastClass();
    }
    catch( Exception e ) {
      System.out.println( "Error: " + e.getMessage() );
      e.printStackTrace();
    }
  }
}
private static void ASAConnect( String UserID,
                                String Password,
                                String Machinename ) {
    // uses global Connection variable

    String _coninfo = new String( Machinename );

    Properties _props = new Properties();
    _props.put("user", UserID );
    _props.put("password", Password );

    // Load the Sybase Driver
    try {
      Class.forName("com.sybase.jdbc.SybDriver").newInstance();

      StringBuffer temp = new StringBuffer();
```

```
      // Use the Sybase jConnect driver...
      temp.append("jdbc:sybase:Tds:");
      // to connect to the supplied machine name...
      temp.append(_coninfo);
      // on the default port number for ASA...
      temp.append(":2638");
      // and connect.
      System.out.println(temp.toString());
     conn = DriverManager.getConnection( temp.toString() , _props );
    }
    catch ( Exception e ) {
      System.out.println("Error: " + e.getMessage());
      e.printStackTrace();
    }
  }
```

外部接続サンプル・コードの動作

外部接続の例として示したコードは、Java コマンド・ライン・アプリケーションのものです。

パッケージのインポート

このアプリケーションはいくつかのライブラリを必要とします。必要なライブラリは `JDBCExamples.java` の冒頭でインポートされています。

- **java.sql** パッケージには Sun Microsystems の JDBC クラスが含まれています。これらのクラスは、すべての JDBC アプリケーションにおいて必須です。 このパッケージは、Java サブディレクトリ内の `classes.zip` ファイルに格納されています。
- **com.sybase.jdbc** からインポートされている Sybase jConnect JDBC ドライバは、jConnect を使用して接続するすべてのアプリケーションにおいて必須です。 このパッケージは、Java サブディレクトリ内の `jdbcdrv.zip` ファイルに格納されています。
- このアプリケーションは*「プロパティ・リスト」*を使用します。 プロパティ・リストを扱うには **java.util.Properties** クラスが必要です。 このパッケージは、`Java` サブディレクトリ内の `classes.zip` ファイルに格納されています。
- **sybase.sql** パッケージには、直列化に使用するユーティリティが含まれています。 このパッケージは、`Java` サブディレクトリ内の `asajdbc.zip` ファイルに格納されています。
- **asademo** パッケージには、一部の例で使用するサンプル・クラスが含まれています。 このパッケージは、`java` サブディレクトリ内の `asademo.jar` ファイルに格納されています。

main メソッド

Java アプリケーションには **main** という名前のメソッドを持つクラスが必要です。これはプログラムの起動時に呼び出されるメソッドです。 この簡単な例の場合、**JDBCExamples.main** は、アプリケーションの唯一のメソッドです。

JDBCExamples.main メソッドは、次のタスクを実行します。

- コマンド・ライン引数を処理し、引数が指定されている場合は、それをマシン名として使用します。 デフォルトのマシン名は *localhost* です。このデフォルト名は、個人用のデータベース・サーバに適しています。
- ASAConnect メソッドを呼び出して、接続を確立します。
- いくつかのメソッドを実行して、コマンド・ラインにデータをスクロールしながら表示します。

ASAConnect メソッド

JDBCExamples.ASAConnect メソッドは、次のタスクを実行します。

- Sybase jConnect を使用して、実行中のデフォルトのデータベースに接続します。
  - **Class.forName** は、jConnect をロードします。 **newInstance** メソッドを使用しているのは、一部のブラウザで発生する問題を回避するためです。
  - **StringBuffer** 文を使用して、リテラル文字列と、コマンド・ラインで指定されたマシン名から接続文字列を構築します。
  - 接続文字列を使用して **DriverManager.getConnection** を実行することにより、接続を確立します。
- 呼び出し元メソッドに制御を戻します。

外部接続サンプル・アプリケーションの実行

この項では、外部接続の例を示すサンプル・アプリケーションの実行方法を説明します。

1. システムのコマンド・プロンプトにおいて、Sybase IQ インストール・ディレクトリに移動します。
2. `IQ-15_3/java` サブディレクトリに移動します。
3. CLASSPATH 環境変数の値として、カレント・ディレクトリ (.) とインポート対象の各種 zip ファイルを必ず指定します。たとえば、コマンド・プロンプトから次のように入力して、CLASSPATH 環境変数の値を設定します (このコマンドは全体を 1 行として入力する必要があります)。

   ```
   set classpath=..¥java¥jdbcdrv.zip;.;..¥java
   ¥asajdbc.zip;asademo.jar
   ```

   Java のデフォルトの zip ファイル名は `classes.zip` です。 `classes.zip` という名前のファイルに格納されているクラスを使用するには、そのファイルが

置かれているディレクトリの名前のみを CLASSPATH 変数で指定すれば十分です。zip ファイル自体のファイル名は必要ありません。それ以外の名前のファイルに格納されているクラスについては、zip ファイル名を指定する必要があります。

サンプル・アプリケーションを実行するには、CLASSPATH 変数においてカレント・ディレクトリを指定する必要があります。

4. TCP/IP が実行されているデータベース・サーバ上に、データベースがロードされていることを確認します。 このようなサーバをローカル・マシン上で起動するには、次のコマンドを使用します (`IQ-15_3/samples/sqlanywhere` サブディレクトリから実行します)。

   UNIX の場合： `start_iq .../iqdemo`

   Windows の場合： `start_iq ...¥iqdemo`

5. コマンド・プロンプトで次のコマンドを入力して、サンプル・アプリケーションを実行します。

   ```
   java JDBCExamples
   ```

   別のマシン上で動作しているサーバに対してこのコマンドを実行する場合は、正しいマシン名を入力する必要があります。 デフォルトのマシン名は `localhost` です。これは、現在のマシン名のエイリアスです。

6. コマンド・プロンプトに人名と製品のリストが表示されたことを確認します。

   接続の確立が失敗した場合は、リストの代わりにエラー・メッセージが表示されます。その場合は、必要な上記の手順をすべて正しく実行したことを再確認します。CLASSPATH が正しく設定されていることもチェックしてください。CLASSPATH の指定が不適切な場合は、必要なクラスを見つけられません。

参照：

- *サーバ側 JDBC クラスからの接続の確立* (137 ページ)
- *JDBC クライアント・アプリケーションからの jConnect による接続* (133 ページ)
- *Sybase jConnect JDBC ドライバ* (148 ページ)

サーバ側 JDBC クラスからの接続の確立

JDBC で SQL 文を構築するには、**Connection** オブジェクトの **createStatement** メソッドを使用します。 クラスがサーバ内で実行されている場合でも、**Connection** オブジェクトを作成するために、接続を確立する必要があります。

サーバ側 JDBC クラスからの接続の確立は、外部接続の確立よりも単純明快な処理です。サーバ側のクラスを実行するユーザはすでにサーバに接続しているので、そのクラスでは現在の接続がそのまま使用されます。

JDBC 接続に関する注意

- オートコミット動作—JDBC の仕様では、デフォルトの場合、データを修正する文を使用した後は、毎回 COMMIT を実行する必要があります。現在、サーバ側の JDBC の動作は、コミットを実行するように設定されています。この動作は、次のような文を使用して制御できます。
  ```
  conn.setAutoCommit( false ) ;
  ```
  conn は、現在使用している接続オブジェクトです。
- 接続デフォルト—サーバ側 JDBC では、**getConnection("jdbc:default:connection")** の呼び出しによってデフォルト値を使用した新しい接続が作成されるのは、初回の呼び出し時のみです。後続の呼び出しは、接続プロパティをまったく変更せずに、現在の接続のラッパを返します。 初回の接続時に自動コミットを無効に設定した場合、同じ Java コード内におけるその後の **getConnection** の呼び出しでは、常に、自動コミットが無効に設定された接続が返されます。
  接続を閉じるたびに接続プロパティをデフォルト値に再設定して、その後の呼び出しで標準的な JDBC 値を持つ接続を取得できるようにすることが必要な場合もあります。その場合は、次のようなコードを使用して再設定を実現できます。
  ```
  Connection conn = DriverManager.getConnection("");
  boolean oldAutoCommit = conn.getAutoCommit();
  try {
       // do code here
  }
  finally {
      conn.setAutoCommit( oldAutoCommit );
  }
  ```
  この方法は AutoCommit だけでなく、TransactionIsolation や isReadOnly などのその他の接続プロパティにも適用できます。

参照：
- *JDBC クライアント・アプリケーションからの jConnect による接続*(133 ページ)
- *Sybase jConnect JDBC ドライバ*(148 ページ)
- *外部接続サンプル・アプリケーションの実行*(136 ページ)

サーバ側接続のサンプル・コード

サーバ側接続のサンプルのソース・コードです。

InternalConnect メソッドのソース・コードは、Sybase IQ インストール・ディレクトリの下の `C:¥Documents and Settings¥All Users¥SybaseIQ¥samples¥SQLAnywhere¥JDBC` ディレクトリにある `JDBCExamples.java` に記述されています。

```
public static void InternalConnect() {
  try {
      conn = DriverManager.getConnection("jdbc:default:connection");
      System.out.println("Hello World");
    }
    catch ( Exception e ) {
      System.out.println("Error: " + e.getMessage());
      e.printStackTrace();
    }
  }
}
```

サーバ側接続サンプル・コードの動作

この簡単な例の場合、**InternalConnect()** は、アプリケーションの唯一のメソッドです。

このアプリケーションに必要なライブラリは、`JDBCExamples.java` クラスのソース・コードの 1 行目でインポートされているライブラリ (JDBC ライブラリ) のみです。その他のライブラリは外部接続の場合に使用されます。 **java.sql** という名前のこのパッケージには、JDBC クラスが格納されています。

InternalConnect() メソッドは、次のタスクを実行します。

1. 現在の接続を使用して、実行中のデフォルトのデータベースに接続します。
   - 接続文字列 **jdbc:default:connection** を使用して **DriverManager.getConnection** を実行することにより、接続を確立します。
2. 現在の標準出力であるサーバ・ウィンドウに `Hello World` という文字列を出力します。この出力処理は、**System.out.println** によって実行されます。
3. エラーが発生して接続を確立できなかった場合は、エラーが発生した場所を示すエラー・メッセージがサーバ・ウィンドウに表示されます。
   try と **catch** は、エラー処理のフレームワークを使用するための構文です。
4. クラスの実行が終了します。

サーバ側接続サンプル・アプリケーションの実行

この項では、サーバ側接続の例を示すサンプル・アプリケーションの実行方法を説明します。

1. `JDBCExamples.java` ファイルのコンパイルを実行済みでない場合は、コンパイルを行います。 JDK を使用している場合は、Sybase IQ インストール・ディレクトリの下の `C:¥Documents and Settings¥All Users¥SybaseIQ¥samples¥SQLAnywhere¥JDBC` ディレクトリでコマンド・プロンプトから次のコマンドを実行することにより、コンパイルできます。

   ```
   javac JDBCExamples.java
   ```

2. デモ・データベースを指定して、データベース・サーバを起動します。 このようなサーバをローカル・マシン上で起動するには、次のコマンドを使用します (`/ASIQ-12_7/java` サブディレクトリから実行します)。

   UNIX の場合： `start_iq .../iqdemo`

   Windows の場合： `start_iq ...¥iqdemo`

   この例の場合は jConnect を使用しないので、TCP/IP ネットワーク・プロトコルは必要ありません。ただし、データベースで Java クラスを使用するには、8Mb 以上のキャッシュが使用可能であることが必要です。
3. クラスをデモ・データベースにインストールします。 インストールするには、デモ・データベースに接続してから、Interactive SQL において次のコマンドを実行します。

   ```
   INSTALL JAVA NEW
   FROM FILE 'C:¥Documents and Settings¥All Users¥SybaseIQ¥samples
   ¥SQLAnywhere¥JDBC¥JDBCExamples.class'
   ```

   ただし、*path* の部分は実際のインストール・ディレクトリのパスに置き換えてください。

   Sybase Central を使用する方法でクラスをインストールすることもできます。その場合は、デモ・データベースに接続した状態で [Java オブジェクト] フォルダを開き、[Add Class] をダブルクリックします。その後は、ウィザードの指示に従って作業を進めてください。
4. インストールが完了したら、ストアド・プロシージャの場合と同様の方法で、このクラスの **InternalConnect** メソッドを呼び出せます。

   ```
   CALL JDBCExamples>>InternalConnect()
   ```

   セッションで初めて Java クラスを呼び出すときには、内部 Java 仮想マシンをロードする必要があります。この処理は数秒の時間を要する場合があります。
5. サーバの画面上に `Hello World` というメッセージが表示されたことを確認します。

JDBC を使用したデータ・アクセス

一部またはすべてのクラスをデータベース内で保持する Java アプリケーションは、さまざまな点で、従来の SQL ストアド・プロシージャに比べて大幅に優れています。しかし、概要の理解を目的とする段階では、SQL ストアド・プロシージャとの類似性に注目しながら JDBC の機能を示すことも有用です。

この項の例では、`Department` テーブルにローを挿入する Java クラスを作成します。

他のインタフェースと同様、JDBC の SQL 文は「*静的*」の場合と「*動的*」の場合があります。静的 SQL 文は、Java アプリケーション内で構築されてからデータベースに送信されます。データベース・サーバは受信した文を解析し、実行プランを選択して、その文を実行します。解析から実行プラン選択までの処理は、文の「*準備*」と呼ばれます。

類似した文を多数回実行する必要がある場合 (たとえば、1 つのテーブルに対する挿入を多数回実行する場合)、静的 SQL では文の準備の処理を毎回実行する必要があるため、大きなオーバヘッドが発生する可能性があります。

一方、動的 SQL 文ではプレースホルダが使用されます。動的 SQL 文の準備はプレースホルダを使用して 1 回実行すれば済むので、多数回実行してもそれ以上準備のコストが増加することはありません。

この項では、静的 SQL を使用します。動的 SQL については後述します。

JDBC に関するその他の注意

- アクセス・パーミッション—JDBC 文を含んでいるクラスは、データベース内のすべての Java クラスと同じように、すべてのユーザがアクセスできます。プロシージャを実行するパーミッションを付与する GRANT EXECUTE 文に相当するものはなく、クラス名をその所有者名で修飾する必要はありません。
- 実行パーミッション—Java クラスの実行には、その実行に使用される接続のパーミッションが適用されます。 この動作は、所有者のパーミッションで実行されるストアド・プロシージャの場合とは異なります。

JDBCExamples クラスのインストール

この項では、この付録の以後の説明で使用する例を実行するために必要な JDBCExamples.class のインストール方法と準備作業について説明します。

サンプル・コード

この項で示すサンプル・コードは、`C:¥Documents and Settings¥All Users¥SybaseIQ¥samples¥SQLAnywhere¥JDBC¥JDBCExamples.java` クラスの完全なソース・コードの一部を抜粋したものです。このソース・コードは、インストール・ディレクトリの下にあります。

JDBCExamples クラスのインストール

1. クラスのインストールをまだ実行していない場合は、`JDBCExamples.class` ファイルをデモ・データベースにインストールします。
2. SQL からデモ・データベースに接続した状態で、[SQL 文] ウィンドウ枠に次のコマンドを入力してください。

```
INSTALL JAVA NEW
FROM FILE 'C:¥Documents and Settings¥All Users¥SybaseIQ¥samples
¥SQLAnywhere¥JDBC¥JDBCExamples.class'
```

ただし、*path* の部分は実際のインストール・ディレクトリのパスに置き換えてください。

Sybase Central を使用する方法でクラスをインストールすることもできます。その場合は、デモ・データベースに接続した状態で [Java オブジェクト] フォルダを開き、[Add Class or JAR] をダブルクリックします。その後は、ウィザードの指示に従って作業を進めてください。

JDBC を使用した挿入、更新、削除

Statement オブジェクトは、静的 SQL 文を実行します。 INSERT、UPDATE、DELETE など、結果セットを返さない SQL 文を実行するには、**Statement** オブジェクトの **executeUpdate** メソッドを使用します。 CREATE TABLE などのデータ定義文も、**executeUpdate** を使用して実行できます。

次のサンプル・コードは、JDBC による INSERT 文の実行例を示したものです。 この例では、**conn** という名前の Connection オブジェクトで保持されている内部接続を使用しています。JDBC を使用して外部アプリケーションから値を挿入する場合は、別の接続を使用する必要がありますが、それ以外の部分についてはこのサンプル・コードをそのまま使用できます。

```
public static void InsertFixed()  {
    // returns current connection
    conn = DriverManager.getConnection("jdbc:default:connection");
    // Disable autocommit
    conn.setAutoCommit( false );

    Statement stmt = conn.createStatement();

    Integer IRows = new Integer( stmt.executeUpdate
       ("INSERT INTO Department (dept_id, dept_name )"
        + "VALUES (201, 'Eastern Sales')"
         ) );
    // Print the number of rows updated
    System.out.println(IRows.toString() + "row inserted" );
  }
```

注意： この抜粋したコードは、**JDBCExamples** クラスの **InsertFixed** メソッドの一部です。 Windows では、`C:¥Documents and Settings¥All Users¥SybaseIQ¥samples¥SQLAnywhere¥JDBC` にある `build.bat` を使用してこのクラスを作成できます。

- **setAutoCommit** メソッドを使用して自動コミットの動作を無効に設定しているため、変更がコミットされるのは明示的に COMMIT 命令を実行した場合のみです。
- **executeUpdate** メソッドの戻り値は、その操作によって変更されたローの数を示す整数値です。この例では、INSERT が成功した場合、戻り値として 1 が返されます。
- 戻り値の整数は、**Integer** オブジェクトに変換されます。 Integer クラスは **int** 基本データ型に対応するラッパ・クラスであり、**toString()** など、いくつかの有用なメソッドを備えています。
- Integer 型の **IRows** は、文字列に変換されて出力されます。この文字列の出力先はサーバ・ウィンドウです。

JDBC による Insert の例を実行する

非常に単純な JDBC クラスを作成します。

1. Interactive SQL を使用し、ユーザ ID dba を指定してデモ・データベースに接続します。
2. JDBCExamples クラスがインストールされていることを確認します。このクラスは、他の Java サンプル・クラスとともにインストールされます。
3. 次のようにしてメソッドを呼び出します。

   ```
   CALL JDBCExamples>>InsertFixed()
   ```
4. 次の文を実行して、department テーブルに 1 つのローが追加されたことを確認します。

   ```
   SELECT *
   FROM department
   ```

   ID 201 のローはコミットされていません。 **ROLLBACK** 文を実行することにより、このローを削除できます。

Java メソッドの引数の指定

ここでは、**InsertFixed** メソッドを拡張した例を使用して、Java メソッドに引数を渡す方法を説明します。

次のメソッドは、呼び出し時に渡された引数を、テーブルに挿入する値として使用します。

```
public static void InsertArguments(
                  String id, String name)  {
try {
      conn = DriverManager.getConnection(
                     "jdbc:default:connection" );

String sqlStr =  "INSERT INTO Department "
    + " ( dept_id, dept_name )"
```

```
    + " VALUES (" +  id +  ", '"  + name + "')";

      // Execute the statement
      Statement stmt = conn.createStatement();
      Integer IRows = new
Integer( stmt.executeUpdate( sqlStr.toString() ) );

      // Print the number of rows updated
      System.out.println(IRows.toString() + " row inserted" );
    }
    catch ( Exception e ) {
      System.out.println("Error: " + e.getMessage());
      e.printStackTrace();
    }
  }
```

引数を持つ Java メソッドの使用

- このメソッドの 2 つの引数は、部署 ID (整数) と部署名 (文字列) です。 この例では、引数は両方とも文字列としてメソッドに渡されています。これは引数の値が SQL 文の文字列の一部として使用されるためです。
- INSERT は静的 SQL 文であり、SQL 文以外のパラメータを要求しません。
- 指定した引数の数やデータ型が正しくない場合は、`Procedure Not Found` エラーが表示されます。

1. `JDBCExamples.class` ファイルをデモ・データベースにインストールしていない場合は、インストールを実行します。
2. Interactive SQL からデモ・データベースに接続して、次のコマンドを入力します。

   ```
   call JDBCExamples>>InsertArguments( '203', 'Northern Sales' )
   ```
3. 次の文を実行して、Department テーブルに 1 つのローが追加されたことを確認します。

   ```
   SELECT *
   FROM Department
   ```
4. 次のコマンドによってロールバックを実行し、データベースを元の状態に戻します。

   ```
   ROLLBACK
   ```

JDBC を使用したクエリ

Statement オブジェクトは、結果セットを返さない SQL 文だけでなく、静的クエリも実行します。 クエリを実行する場合は、**Statement** オブジェクトの **executeQuery** メソッドを使用します。 このメソッドは、結果セットを **ResultSet** オブジェクトとして返します。

次のサンプル・コードは、JDBC によるクエリの実行例を示したものです。 このサンプル・コードは、1 つの製品の在庫値の合計を **inventory** という名前の変数に

格納します。 製品名を保持する変数は、**String** 型の **prodname** です。 この例は、**JDBCExamples** クラスの **Query** メソッドの部分を抜粋して示したものです。

この例では、すでに内部接続または外部接続を取得して、それを **conn** という名前の Connection オブジェクトとして保持していると仮定しています。また、変数が存在することも仮定しています。

```
public static void Query () {
int max_price = 0;
    try{
      conn = DriverManager.getConnection(
                  "jdbc:default:connection" );

      // Build the query
      String sqlStr =  "SELECT id, unit_price "
    + "FROM product" ;

      // Execute the statement
      Statement stmt = conn.createStatement();
      ResultSet result = stmt.executeQuery( sqlStr );

      while( result.next() ) {
    int price = result.getInt(2);
    System.out.println( "Price is "  + price );
    if( price > max_price ) {
      max_price = price ;
    }
      }
    }
    catch( Exception e ) {
      System.out.println("Error: " + e.getMessage());
      e.printStackTrace();
    }
      return max_price;
  }
```

サンプル・コードの実行
デモ・データベースへの **JDBCExamples** クラスのインストールが完了したら、Interactive SQL で次の文を使用することにより、このメソッドを実行できます。

```
select JDBCExamples>>Query()
```

注意

- このクエリは、**prodname** という名前のすべての製品の数量と単価を取得します。 クエリの結果は、**result** という名前の **ResultSet** オブジェクトに格納されます。
- ループにより、結果セットの各ローに対して反復的に処理が実行されます。このループでは **next** メソッドが使用されています。

- 各ローの処理では、**getInt** メソッドを使用して、各カラムの値が整数変数に格納されます。**ResultSet** では、**getString**、**getDate**、**getBinaryString** など、その他のデータ型に対応したメソッドも使用できます。
  getInt メソッドの引数は、カラムを指定するインデックス番号です。この番号は 1 から始まります。
  SQL から Java へのデータ型変換は、『Sybase IQ リファレンス・マニュアル』の「SQL データ型」の「Java/SQL データ型変換」に記載されている規則に従って実行されます。
- Sybase IQ は、カーソルの双方向スクロールをサポートしています。 しかし、JDBC で提供されているのは、結果セットのカーソルを前に進めることに相当する **next** メソッドのみです。
- このメソッドは呼び出し元の環境に **max_price** の値を返し、Interactive SQL はその値を [結果] ウィンドウ枠に表示します。

参照：
- *分散アプリケーション* (153 ページ)
- *オブジェクトの挿入と検索* (147 ページ)

準備文を使用した効率的なアクセス

Statement インタフェースを使用する場合、データベースに送られた各 SQL 文は、文の解析と実行プランの生成が完了してから実行されます。実際に実行される前のこれらの処理は、文の「*準備*」と呼ばれます。

PreparedStatement を使用すると、パフォーマンスの向上を実現できます。このインタフェースでは、あらかじめプレースホルダを使用して文を準備しておき、文の実行時にプレースホルダに実際の値を割り当てることができます。

特に、多数のローを挿入するときなど、類似した操作を多数回実行する場合には、準備文の使用は非常に有効です。

準備文の詳細については、『リファレンス： 文とオプション』>「SQL 文」>「PREPARE 文 [ESQL]」を参照してください。

例
次の例は、**PreparedStatement** インタフェースの使用方法を示したものです。ただし、実際には、このような単一のローの挿入操作は準備文の使用に適しているとはいえません。

次に示す **JDBCExamples** クラスのメソッドは、準備文を実行します。

```
public static void JInsertPrepared(int id, String name) try {
      conn = DriverManager.getConnection(
                    "jdbc:default:connection");
```

```
      // Build the INSERT statement
      // ? is a placeholder character
      String sqlStr = "INSERT INTO Department "
    + "( dept_id, dept_name ) "
    + "VALUES ( ? , ? )" ;

      // Prepare the statement
      PreparedStatement stmt = conn.prepareStatement( sqlStr );

      stmt.setInt(1, id);
      stmt.setString(2, name );
      Integer IRows = new Integer(
                           stmt.executeUpdate() );

      // Print the number of rows updated
      System.out.println(IRows.toString() + " row inserted" );
    }
    catch ( Exception e ) {
      System.out.println("Error: " + e.getMessage());
      e.printStackTrace();
    }
  }
```

サンプル・コードの実行
デモ・データベースへの **JDBCExamples** クラスのインストールが完了したら、次の文を入力することにより、このサンプル・コードを実行できます。

```
call JDBCExamples>>InsertPrepared(
                             202, 'Eastern Sales' )
```

引数の文字列は一重引用符で囲まれています。これは SQL に適した指定方法です。このメソッドを Java アプリケーションから呼び出す場合は、文字列を二重引用符で囲んでください。

オブジェクトの挿入と検索

JDBC は、リレーショナル・データベースに対するインタフェースとして、従来の SQL データ型を検索、操作できるように設計されています。

Sybase IQ では、Java クラスという形で抽象データ型も提供されています。JDBC を使用して Java クラスにアクセスする方法は、オブジェクトを挿入する場合と検索する場合では異なります。

参照：

- *分散アプリケーション* (153 ページ)
- *JDBC を使用したクエリ* (144 ページ)

オブジェクトの検索

オブジェクトとそのフィールドやメソッドの検索は、次の方法で実行できます。

- メソッドとフィールドへのアクセス—Java メソッドとフィールドは、クエリの select リストで指定できます。 その場合、メソッドまたはフィールドは結果セットにカラムとして表示されるため、**getInt** や **getString** など、標準的な **ResultSet** メソッドのいずれかを使用してアクセスできます。
- オブジェクトの検索—クエリの select リストの中に、Java クラスのデータ型を持つカラムを指定した場合は、**ResultSet** の **getObject** メソッドを使用してオブジェクトを取得し、Java クラスに格納できます。そして、Java クラスとして格納されたそのオブジェクトのメソッドとフィールドにアクセスできます。 Java オブジェクトを格納できる格納先は、カタログ・ストアのみです。

オブジェクトの挿入

サーバ側の Java クラスからは、JDBC **setObject** メソッドを使用して、Java クラスのデータ型を持つカラムにオブジェクトを挿入できます。

オブジェクトの挿入は、準備文を使用して実行できます。たとえば、次のサンプル・コードは、MyJavaClass 型のオブジェクトをテーブル T のカラムに挿入します。

```
java.sql.PreparedStatement ps =
    conn.prepareStatement("insert T values( ? )" );
ps.setObject( 1, new MyJavaClass() );
ps.executeUpdate();
```

別の方法として、SQL 変数を設定してオブジェクトを格納してから、その SQL 変数をテーブルに挿入することもできます。

Sybase jConnect JDBC ドライバ

アプリケーションまたはアプレットから JDBC を使用する場合は、Sybase IQ データベースに接続するために Sybase jConnect が必要です。

Sybase IQ に Sybase jConnect が含まれているかどうかは、使用しているインストール・パッケージの種類によって異なります。クライアント・アプリケーションから JDBC を使用するには jConnect が必要です。サーバ側 JDBC を使用する場合は、jConnect は不要です。

jConnect の詳細情報、および Sybase IQ における jConnect の使用方法の詳細については、jConnect に関するオンラインの資料、または jConnect Web サイトの情報を参照してください。

注意： アプリケーションで jConnect を使用する前に、次の文を実行して、ドライバをロードする必要があります。

```
Class.forName("com.sybase.jdbc.SybDriver").newInstance();
```

newInstance メソッドを使用しているのは、一部のブラウザで発生する問題を回避するためです。

参照：

- *サーバ側 JDBC クラスからの接続の確立* (137 ページ)
- *JDBC クライアント・アプリケーションからの jConnect による接続* (133 ページ)
- *外部接続サンプル・アプリケーションの実行* (136 ページ)

Sybase IQ で提供されている jConnect のバージョン

Sybase IQ では、次の 2 種類のバージョンの Sybase jConnect JDBC ドライバが提供されています。

- フル・バージョン—jConnect をインストールすることを選択した場合、Sybase IQ インストール・ディレクトリに jConnect サブディレクトリが追加されます。このサブディレクトリには、すべての jConnect ファイルを含むディレクトリ・ツリーが格納されます。
- Zip ファイル—リモート・データ・アクセス機能と Java デバッガは、どちらもデータベースへの接続に jConnect を使用します。 そのため、開発用のフル・バージョンのドライバをインストールしない場合でも jConnect を使用できるように、基本的な jConnect クラスは zip ファイルとして提供されます。

jConnect ドライバのファイル

Sybase jConnect ドライバのインストール先は、Sybase IQ インストール・ディレクトリ内の `jConnect` サブディレクトリです。ドライバのファイルは、このサブディレクトリの下の複数のディレクトリに分類されて格納されています。
jConnect をインストールしていない場合は、Java サブディレクトリにインストールされている `jdbcdrv.zip` ファイルを使用できます。

jConnect の CLASSPATH 設定

アプリケーションで jConnect を使用するには、コンパイル時と実行時に、jConnect クラスのパスが CLASSPATH 環境変数の値として指定されている必要があります。これは Java コンパイラと Java ランタイムが必要なファイルを見つけることを可能にするための設定です。

たとえば、次のコマンドは、jConnect ドライバのクラス・パスを既存の CLASSPATH 環境変数の設定に追加します。ただし、*path* の部分は実際の Sybase IQ インストール・ディレクトリのパスに置き換えてください。

```
set classpath=%classpath%;path¥jConnect¥classes
```

CLASSPATH に `jdbcdrv.zip` ファイルを追加する場合は、次のコマンドを使用します。

```
set classpath=%classpath%;path¥java¥jdbcdrv.zip
```

jConnect クラスのインポート

jConnect のクラスはすべて **com.sybase** パッケージに含まれています。クライアント・アプリケーションは、**com.sybase.jdbc** 内のクラスにアクセスする必要があります。そのため、アプリケーションで jConnect を使用するには、各ソース・ファイルの最初の部分に次のように記述して、これらのクラスをインポートする必要があります。

```
import com.sybase.jdbc.*
```

参照：

- *JDBC の概要* (127 ページ)

データベースへの jConnect システム・オブジェクトのインストール

jConnect を使用してシステム・テーブル情報 (データベース・メタデータ) にアクセスする場合は、jConnect システム・オブジェクトをデータベースに追加する必要があります。

デフォルトでは、バージョン 12.7 を使用して作成されるすべてのデータベースと、バージョン 12.7 にアップグレードされるすべてのデータベースに jConnect システム・オブジェクトが追加されます。

データベースへの jConnect オブジェクトの追加を、データベースの作成時またはアップグレード時に実行するか、またはその後の任意の時点で実行するかは、必要に応じて選択できます。

jConnect システム・オブジェクトのインストールは、Interactive SQL から実行できます。

Sybase Central を使用して、バージョン 12.7 のデータベースに jConnect システム・オブジェクトを追加する

1. Sybase Central を使用して、DBA 権限のあるユーザとしてデータベースに接続します。
2. Sybase Central ビューアの左側のウィンドウ枠で、データベースのアイコンを右クリックし、ポップアップ・メニューから [jConnect メタデータ・サポートの再インストール] を選択します。

Interactive SQL を使用して、バージョン 12.7 のデータベースに jConnect システム・オブジェクトを追加する

Interactive SQL を使用して、DBA 権限を持つユーザとしてデータベースに接続し、[SQL Statements] ウィンドウ枠に次のコマンドを入力します。

```
read path¥scripts¥jcatalog.sql
```

path の部分は実際の Sybase IQ インストール・ディレクトリのパスに置き換えてください。

注意： コマンド・プロンプトを使用して、バージョン 12.7 のデータベースに jConnect システム・オブジェクトを追加することもできます。その場合は、コマンド・プロンプトで次のように入力します。

```
dbisql -c "uid=user;pwd=pwd" path¥scripts¥jcatalog.sql
```

user と *pwd* には、DBA 権限を持つ実際のユーザが使用している値を指定し、*path* の部分は実際の Sybase IQ インストール・ディレクトリのパスに置き換えてください。

サーバを示す URL の指定

jConnect を使用してデータベースに接続するには、データベースを示す URL (Universal Resource Locator) を指定する必要があります。

前述した例では、次の部分で URL を指定しています。

```
StringBuffer temp = new StringBuffer();
// Use the Sybase jConnect driver...
temp.append("jdbc:sybase:Tds:");
// to connect to the supplied machine name...
temp.append(_coninfo);
// on the default port number for ASA...
temp.append(":2638");
// and connect.
System.out.println(temp.toString());
conn = DriverManager.getConnection(temp.toString() , _props );
```

文字列を連結して、全体としては次のような URL が構築されます。

```
jdbc:sybase:Tds:machine-name:port-number
```

URL は、次の各部から構成されています。

- jdbc:sybase:Tds—Sybase jConnect JDBC ドライバと TDS アプリケーション・プロトコルを使用することを示します。

- machine-name—サーバが実行されているマシンの IP アドレスまたはマシン名です。 同じマシン上から接続を確立する場合は、現在のマシンを示す `localhost` という名前を使用できます。
- port number—データベース・サーバが受信に使用しているポート番号です。 Sybase IQ に割り当てられているポート番号は 2638 です。ポート番号を変更する理由が特にない場合は、この番号をそのまま使用してください。

接続文字列の長さは 253 文字未満である必要があります。

サーバ上のデータベースの指定

Sybase IQ の各サーバでは、同時に複数のデータベースがロードされている場合があります。前項で説明した URL はサーバを指定しますが、データベースは指定しません。データベースが指定されていない場合は、サーバ上のデフォルトのデータベースに対して接続が試行されます。

特定のデータベースを指定するには、次に示すどちらかの方法で拡張形式の URL を指定します。

ServiceName パラメータの使用

```
jdbc:sybase:Tds:machine-name:port-number?ServiceName=DBN
```

疑問符に続けてパラメータとその値の組を記述することは、URL で引数を指定する標準的な方法です。 **ServiceName** の大文字と小文字は区別されません。= 記号の両側にはスペースを入れないでください。 *DBN* パラメータは、データベース名です。

RemotePWD パラメータの使用

より汎用的な指定方法を使用すると、**RemotePWD** フィールドを使用して、データベース名やデータベース・ファイルなどの接続パラメータを追加指定できます。 **RemotePWD** を Properties のフィールドとして設定するには、**setRemotePassword()** メソッドを使用します。

次のサンプル・コードは、このフィールドの使用例を示したものです。

```
sybDrvr = (SybDriver)Class.forName(
     "com.sybase.jdbc2.jdbc.SybDriver" ).newInstance();
props = new Properties();
props.put( "User", "DBA" );
props.put( "Password", "SQL" );
sybDrvr.setRemotePassword(
    null, "dbf=asiqdemo.db", props );
Connection con = DriverManager.getConnection(
    "jdbc:sybase:Tds:localhost", props );
```

データベース・ファイル・パラメータ **DBF** を使用すると、jConnect を使用して、サーバ上のデータベースを起動できます。デフォルトでは、データベースは

autostop=YES の設定で起動されます。 **utility_db** の DBF または DBN を指定すると、ユーティリティ・データベースが自動的に起動されます。

ユーティリティ・データベースについては、『システム管理ガイド： 第 1 巻』の「Sybase IQ システム管理の概要」と「ユーザ ID とパーミッションの管理」を参照してください。

RemotePWD に、TDS クライアントからの IQ 固有の接続パラメータを指定してください。

次の例は、IQ 固有の接続パラメータの指定方法を示しています。

```
p.put(“RemotePWD“,“,,CON=myconnection”);
```

myconnection が IQ 接続名になります。

分散アプリケーション

「*分散アプリケーション*」では、アプリケーション・ロジックがいくつかの部分に分けられ、その各部が複数のマシン上で別々に実行されます。 Sybase IQ では、分散 Java アプリケーションを作成できます。この分散アプリケーションでは、ロジックの一部がデータベース・サーバ上で実行され、残りのロジックはクライアント・マシン上で実行されます。

Sybase IQ は、外部の Java クライアントとの間で Java オブジェクトを交換する機能を備えています。

クライアント・アプリケーションからデータベース内のオブジェクトを検索できるようにすることは、分散アプリケーションの鍵となる重要なタスクです。この項では、このタスクを実現する方法を説明します。

分散アプリケーションの機能

`JDBCExamples.java` では、分散コンピューティングを使用するメソッドが他に 2 つ定義されています。

- serializeVariable—このメソッドは、データベース・サーバ上の SQL 変数によって参照されるネイティブ Java オブジェクトを作成し、そのオブジェクトをクライアント・アプリケーションに返します。
- serializeColumnCastClass—このメソッドは **serializeColumn** メソッドに類似していますが、こちらは、サブクラスを再構築する処理の例を示すために提供されています。 クエリされたカラム (**product** テーブルからの **JProd**) は、**asademo.Product** データ型です。 一部のローには、Product クラスのサブクラスである **asademo.Hat** のオブジェクトが格納されています。 クライアント側では、元と同じ適切な型のクラスが再構築されます。

分散アプリケーション構築に必要な作業
分散アプリケーションを構築するには、2 つのタスクを実行することが必要です。

- サーバで実行されるすべてのクラスは、Serializable インタフェースを実装する必要があります。これは非常に簡単です。
- クライアント側アプリケーションは、そのクラスをインポートする必要があります。これは、クライアント側でオブジェクトを再構築できるようにするためです。

次の項では、これらのタスクについて説明します。

参照：
- *オブジェクトの挿入と検索* (147 ページ)
- *JDBC を使用したクエリ* (144 ページ)

Serializable インタフェース

オブジェクトがサーバからクライアント・アプリケーションに渡されるときは、「*直列化*」された形で渡されます。そのため、クライアント・アプリケーションに送信されるオブジェクトは Serializable インタフェースを実装している必要があります。幸い、これは非常に簡単なタスクです。

Serializable インタフェースでは、メソッドや変数は何も定義されていません。オブジェクトを直列化すると、そのオブジェクトはバイト・ストリームに変換され、ディスクに保存することや、別の Java アプリケーションに送信してそこでオブジェクトを再構築することが可能になります。この再構築の処理は「*非直列化*」とも呼ばれます。

データベース・サーバで直列化されてクライアント・アプリケーションに送信され、そこで非直列化された Java オブジェクトは、あらゆる点において元のオブジェクトとまったく同一です。ただし、オブジェクトの変数の一部には、直列化する必要がないものや、セキュリティ上の理由から直列化すべきでないものがあります。 そのような変数は、次の変数宣言の例のように、**transient** キーワードを使用して宣言します。

```
transient String password;
```

この変数を持つオブジェクトが非直列化された場合、この変数の値は常に、デフォルト値である null になります。

独自の直列化を定義する必要がある場合は、クラスに **writeObject()** メソッドと **readObject()** メソッドを追加する方法で実現できます。

直列化の詳細については、Sun Microsystems の Java Development Kit (JDK) を参照してください。

Serializable インタフェースの実装

Serializable インタフェースを実装することは、そのクラスが直列化可能であると単に宣言することを意味します。
クラス定義に、**implements java.io.Serializable** という記述を追加します。

たとえば、`$SADIR/samples/asa/java/asademo` (UNIX) または `%SADIR%¥samples¥asa¥java¥asademo` (Windows) サブディレクトリに格納されている Product クラスでは、次の宣言によって Serializable インタフェースが実装されています。

```
public class Product implements java.io.Serializable
```

クライアント側でのクラスのインポート

クライアント側からオブジェクトを検索するすべてのクラスは、そのオブジェクトを使用するための適切なクラス定義にアクセスできる必要があります。

たとえば、**asademo** パッケージに含まれている **Product** クラスを使用するには、アプリケーションのソース・コードに次の行を記述する必要があります。

```
import asademo.*
```

また、`asademo.jar` ファイルを CLASSPATH 環境変数の値として指定して、アプリケーションがこのパッケージを見つけられるようにすることも必要です。

分散アプリケーションの例

`JDBCExamples.java` クラスには、分散 Java コンピューティングの例を示すためのメソッドが3つ含まれています。 これらのメソッドは、すべて **main** メソッドから呼び出されます。

次に、JDBCExamples クラスの **getObjectColumn** メソッドを示します。

```
private static void getObjectColumn() throws Exception {
// Return a result set from a column containing
// Java objects
    asademo.ContactInfo ci;
    String name;
    String sComment ;

    if ( conn != null ) {
      Statement stmt = conn.createStatement();
      ResultSet rs = stmt.executeQuery(
           "SELECT JContactInfo FROM jdba.contact"
    );
    while ( rs.next() ) {
    ci = ( asademo.ContactInfo )rs.getObject(1);
  System.out.println( "¥n¥tStreet: " + ci.street +
     "City: " + ci.city +
```

```
      "¥n¥tState: " + ci.state +
      "Phone: " + ci.phone +
      "¥n" );
      }
    }
  }
```

この例の getObject メソッドは、内部 Java の場合と同様の方法で使用されています。

参照：

- *JDBC クライアント・アプリケーションからの jConnect による接続*(133 ページ)

データベースでのロジックのデバッグ

この付録では、SQL のストアド・プロシージャとイベント・ハンドラ、および Java のストアド・プロシージャの開発効率の向上に役立つ Sybase デバッガの使用法について説明します。

データベースでのデバッグの概要

開発過程ではデバッガを使用できます。

次のオブジェクトを使用できます。

- SQL ストアド・プロシージャ、イベント・ハンドラ、ユーザ定義の関数
- データベース内の Java ストアド・プロシージャ

デバッガの機能

SQL ストアド・プロシージャ、トリガ、イベント・ハンドラ、ユーザ定義の関数の開発過程でデバッガを使用できます。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - SQL の使用法』>「ストアド・プロシージャとトリガ」>「プロシージャ、関数、トリガ、イベントのデバッグ」>「SQL Anywhere のデバッガの概要」を参照してください。

デバッガを使用するための要件

デバッガを使用するために必要なことを以下に示します。

- パーミッション—DBA 権限を持っているか、SA_DEBUG グループでパーミッションを与えられている必要があります。このグループは、データベースの作成時に自動的にすべてのデータベースに追加されます。
- Java クラスのソース・コード—アプリケーションのソース・コードをデバッガから参照できる必要があります。Java クラスの場合、ソース・コードはハード・ディスク上のディレクトリに格納されています。ストアド・プロシージャの場合、ソース・コードはデータベース内に格納されています。
- コンパイル・オプション—Java クラスをデバッグするには、クラスのコンパイル時に、クラス内にデバッグ情報を含めるオプションを指定する必要があります。 たとえば、Sun Microsystems の JDK コンパイラ `javac.exe` を使用する場合は、コマンド・ライン・オプションの `-g` を使用してコンパイルする必要があります。

注意： Sybase IQ デモ・データベースは iqdemo.db です。

チュートリアル 1： デバッガの作業の開始

このチュートリアルでは、デバッガを起動してデータベースに接続する手順と、Java クラスをデバッグする方法について説明します。

レッスン 1： デバッガの起動とデータベースへの接続

このチュートリアルでは、デバッガを起動してデータベースに接続し、デバッグ用の接続に接続する手順を示します。 Sybase IQ デモ・データベースを使用します。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - SQL の使用法』>「ストアド・プロシージャとトリガ」>「プロシージャ、関数、トリガ、イベントのデバッグ」>「チュートリアル： デバッガの使用開始」>「レッスン 1： データベースへの接続とデバッガの起動」を参照してください。

チュートリアル 2： ストアド・プロシージャのデバッグ

このチュートリアルでは、ストアド・プロシージャをデバッグする場合の作業例を示します。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - SQL の使用法』>「ストアド・プロシージャとトリガ」>「プロシージャ、関数、トリガ、イベントのデバッグ」>「チュートリアル： デバッガの作業の開始」>「レッスン 2： ストアド・プロシージャのデバッグ」を参照してください。

チュートリアル 3： Java クラスのデバッグ

このチュートリアルでは、Interactive SQL (dbisql) から **JDBCExamples.Query()** を呼び出して、デバッガでその実行を中断し、このメソッドのソース・コードをトレースします。

JDBCExamples.Query() メソッドは、デモ・データベースに対して次のクエリを実行します。

```
SELECT ID, UnitPrice
FROM Products
```

そして、ループ処理によってクエリの結果セットのローの値を順次取得し、単価が最も高いローの値を返します。

クラスをデバッグするには、javac の -g オプションを指定してクラスをコンパイルする必要があります。サンプル・クラスは、デバッグが可能なようにコンパイルされています。

注意： Java の例を実行する場合は、Java サンプル・クラスをデモ・データベースにインストールする必要があります。

デモ・データベースの Java サンプル・クラス

Java の例を実行する場合は、Java サンプル・クラスをデモ・データベースにインストールする必要があります。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - プログラミング』>「SQL Anywhere データ・アクセス API」>「SQL Anywhere JDBC ドライバ」>「JDBC を使用したデータ・アクセス」>「例を実行するための準備作業」を参照してください。

デバッガでの Java ソース・コードの表示

デバッガはソース・コード・ファイル (拡張子が .java のファイル) が格納されている場所でファイルを調べます。

[Java ソース・コード・パス] ウィンドウには、デバッガが Java ソース・コードを検索するディレクトリのリストが表示されます。ソース・コードの検索には、パッケージを検索する場合の Java の規則が適用されます。デバッガは、現在 CLASSPATH に指定されているディレクトリでもソース・コードを検索します。

1. 左側のフォルダ・ビューで [Sybase IQ 15] を選択します。
2. Sybase Central で、[モード] - [デバッグ] を選択します。
3. デバッグするユーザを選択する画面が表示されたら、「*」つまり、すべてのユーザを指定して [OK] をクリックします。
4. デバッガのインタフェースで、[デバッグ] - [Java ソース・コード・パスの設定] を選択します。
5. インストール・ディレクトリの下にある java サブディレクトリのパスを入力します。 たとえば、Sybase IQ を %IQDIR15% にインストールした場合は、次のように入力します。

   ```
   %IQDIR15%¥java
   ```
6. [Browse Folder] をクリックし、デバッガが Java ソース・コードを検索するフォルダまたは個々のファイルのリストから選択します。
7. [Browse File] をクリックし、リストに追加するファイルを検索します。
8. [OK] をクリックして、ウィンドウを閉じます。

ブレークポイントの設定

Query() メソッドの最初の行にブレークポイントを設定できます。メソッドが呼び出されると、このブレークポイントで実行が停止します。

1. [ソース・コード] ウィンドウで、**Query()** メソッドの始めが見えるところまでスクロールします。これはクラスの終わりの近くにあり、次のような行で始まっています。

   ```
   public static int Query() {
   ```

2. メソッドの最初の行の先頭に表示されている緑色のインジケータをクリックします。クリックすると、インジケータの色が赤に変わります。メソッドの最初の行は、次のようになっています。

   ```
   int max_price = 0;
   ```

   クリックを繰り返すと、インジケータの状態が交互に切り替わります。ブレークポイントの設定の後には、Java クラスをコンパイルし直す必要はありません。

メソッドの実行

Interactive SQL (dbisql) から **Query()** メソッドを呼び出して、その実行がブレークポイントで中断されることを実際に確認できます。

1. Interactive SQL を起動します。 ユーザ ID として `DBA`、パスワードとして `sql` を指定して、デモ・データベースに接続します。

   この操作で確立された接続が、デバッガの [接続] ウィンドウのリスト内に表示されます。

2. メソッドを呼び出すには、次のコマンドを Interactive SQL で入力します。

   ```
   SELECT JDBCExamples.Query()
   ```

   このクエリは完了しません。実行は、デバッガ内のブレークポイントで停止します。Interactive SQL では、[Stop] ボタンがアクティブになります。デバッガの [Source] ウィンドウに表示される赤い矢印は、現在の行を示しています。

ここからは、デバッガでソース・コードを 1 ステップずつ実行しながら、デバッグ作業を進めることができます。

ソース・コードのステップ実行

説明に従って前項までの作業を行った場合、デバッガによる実行は **JDBCExamples.Query()** メソッドの最初の文で停止しているはずです。

1. [デバッグ] - [ステップ] を選択するか、または [F10] キーを押して、現在のメソッドの次の行に進みます。この手順を 2 ～ 3 回実行してみてください。
2. 次に示した行の末尾をマウスでクリックしてから、[デバッグ] - [Run To Cursor] を選択するか、または [CTRL + F10] キーを押します。これにより、実行が選択行まで進み、そこで停止します。
3. 次に示した行 (292 行目) を選択してから [F9] キーを押して、その行にブレークポイントを設定します。

   ```
   return max_price;
   ```

   ブレークポイントが設定されていることを示すアスタリスクが、左側のカラムに表示されます。 [F5] キーを押して、そのブレークポイントまで実行します。
4. いろいろな方法でソース・コードを 1 ステップずつ実行してみてください。 実行を完了するには、[F5] キーを押します。

   実行が完了すると、Interactive SQL のデータ・ウィンドウに、24 という値が表示されます。
5. 次のブレークポイントまで移動するには、[F5] キーを押します。

   実行が完了すると、Interactive SQL のデータ・ウィンドウに、24 という値が表示されます。

[**Run**] メニューには、ソース・コードをステップ実行するための各種オプションがすべて表示されます。詳細については、デバッガのオンライン・ヘルプを参照してください。

変数の検査と修正

メソッド内で宣言されるローカル変数とデバッガ内のクラス静的変数の、両方の値を点検できます。

クラス・レベルの変数 (静的変数) を [デバッガ] ウィンドウに表示して、その値を点検できます。詳細については、デバッガのオンライン・ヘルプを参照してください。

コードを 1 ステップずつ実行しながらメソッド内のローカル変数の値を点検できるので、各ステップがどのように実行されているかがよくわかります。

注意： Java の例を実行する場合は、Java サンプル・クラスをデモ・データベースにインストールする必要があります。

1. **JDBCExamples.Query** メソッドの最初の行にブレークポイントを設定します。この行は次のようになっています。

   ```
   int max_price = 0
   ```

2. Interactive SQL でこのメソッドを再度実行します。

   ```
   SELECT JDBCExamples.Query()
   ```

   クエリはブレークポイントまで実行されされます。
3. [F7] キーを押して、次の行に進みます。 これで **max_price** 変数が宣言され、0 に初期化されます。
4. [ローカル] ウィンドウが表示されていない場合は、[ウィンドウ] - [ローカル] を選択して、そのウィンドウを表示します。

   [ローカル] ウィンドウに複数のローカル変数が表示されますが、**max_price** の値は 0 になっています。 その他の変数については、すべて `variable not in scope` と表示されます。これは、それらの変数がまだ初期化されていないことを意味します。
5. [ローカル] ウィンドウの [値] カラムで **max_price** に対するエントリをダブルクリックして、**max_price** の値を 45 に変更します。

   45 という値は、他のどの製品よりも高い価格です。したがって、クエリが最高価格として返す値は 24 ではなく、45 になるはずです。
6. [ソース] ウィンドウで [F7] キーを繰り返し押して、コードの実行を進めます。このとき、各変数の値が [ローカル] ウィンドウに表示されます。 **stmt** と **result** 変数に値が表示されるまで、ステップ実行を続けてください。
7. **result** オブジェクトの横のアイコンをクリックして、そのオブジェクトの表示を拡張します。そのオブジェクトの行にカーソルを置いて [Enter] キーを押す方法でも表示を拡張できます。これで、そのオブジェクトの各フィールドの値が表示されます。
8. 変数を検査、修正する手順を十分にテストしたら、[F5] キーを押してクエリの実行を完了させ、チュートリアルを終了します。

ブレークポイント

ブレークポイントはデバッガがソース・コードの実行をいつ停止するかを制御します。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - SQL の使用法』>「ストアド・プロシージャとトリガ」>「プロシージャ、関数、トリガ、イベントのデバッグ」>「ブレークポイントの活用」を参照してください。

変数の動作の表示と編集

デバッガでは、コードのステップ実行時に変数の動作を表示したり編集したりできます。

デバッガには [デバッガの詳細] ペインがあり、そこにストアド・プロシージャ内で使用されている各種の変数が表示されます。[デバッガの詳細] ペインは Sybase Central がデバッグ・モードで実行されているとき、画面下部に表示されます。

SQL Anywhere 11.0.1 の『SQL Anywhere サーバ - SQL の使用法』>「ストアド・プロシージャとトリガ」>「プロシージャ、関数、トリガ、イベントのデバッグ」>「変数の活用」を参照してください。

デバッガ・スクリプトの作成

デバッガでは、Java プログラミング言語を使用してスクリプトを作成できます。スクリプトは **sybase.asa.procdebug.DebugScript** クラスを拡張した Java クラスです。

デバッガでスクリプトを実行すると、そのクラスがロードされ、その **run** メソッドが呼び出されます。**run** メソッドの第 1 パラメータは、そのクラスのインスタンスを示すポインタです。このインタフェースを使用すると、デバッガに対する対話的操作と制御を実行できます。

スクリプトをコンパイルするには、次のようなコマンドを実行します。

```
javac -classpath %asany%/procdebug/ProcDebug.jar;%classpath%
myScript.Java.
```

sybase.asa.procdebug.DebugScript クラス

DebugScript クラスの定義は、次のとおりです。

```
// All debug scripts must inherit from this class

package sybase.asa.procdebug;

abstract public class DebugScript
{
    abstract public void run( IDebugAPI db, String args[] );
    /*
        The run method is called by the debugger
        - args will contain command line arguments
    */

    public void OnEvent( int event ) throws DebugError {}
```

```
    /*
    - Override the following methods to process debug events
    - NOTE: this method will not be called unless you call
        DebugAPI.AddEventHandler( this );
    */

}
```

sybase.asa.procdebug.IDebugAPI インタフェース

IDebugAPI インタフェースの定義は、次のとおりです。

```
package sybase.asa.procdebug;
import java.util.*;
public interface IDebugAPI

{
    // Simulate Menu Items

    IDebugWindow MenuOpenSourceWindow() throws DebugError;
    IDebugWindow MenuOpenCallsWindow() throws DebugError;
    IDebugWindow MenuOpenClassesWindow() throws DebugError;
    IDebugWindow MenuOpenClassListWindow() throws DebugError;
    IDebugWindow MenuOpenMethodsWindow() throws DebugError;
    IDebugWindow MenuOpenStaticsWindow() throws DebugError;
    IDebugWindow MenuOpenCatchWindow() throws DebugError;
    IDebugWindow MenuOpenProcWindow() throws DebugError;
    IDebugWindow MenuOpenOutputWindow() throws DebugError;
    IDebugWindow MenuOpenBreakWindow() throws DebugError;
    IDebugWindow MenuOpenLocalsWindow() throws DebugError;
    IDebugWindow MenuOpenInspectWindow() throws DebugError;
    IDebugWindow MenuOpenRowVarWindow() throws DebugError;
    IDebugWindow MenuOpenQueryWindow() throws DebugError;
    IDebugWindow MenuOpenEvaluateWindow() throws DebugError;
    IDebugWindow MenuOpenGlobalsWindow() throws DebugError;
    IDebugWindow MenuOpenConnectionWindow() throws DebugError;
    IDebugWindow MenuOpenThreadsWindow() throws DebugError;
    IDebugWindow GetWindow( String name ) throws DebugError;

    void MenuRunRestart() throws DebugError;
    void MenuRunHome() throws DebugError;
    void MenuRunGo() throws DebugError;
    void MenuRunToCursor() throws DebugError;
    void MenuRunInterrupt() throws DebugError;
    void MenuRunOver() throws DebugError;
    void MenuRunInto() throws DebugError;
    void MenuRunIntoSpecial() throws DebugError;
    void MenuRunOut() throws DebugError;
    void MenuStackUp() throws DebugError;
    void MenuStackDown() throws DebugError;
    void MenuStackBottom() throws DebugError;
    void MenuFileExit() throws DebugError;
    void MenuFileOpen( String name ) throws DebugError;
    void MenuFileAddSourcePath( String what ) throws DebugError;
    void MenuSettingsLoadState( String file ) throws DebugError;
```

```
    void MenuSettingsSaveState( String file ) throws DebugError;
    void MenuWindowTile() throws DebugError;
    void MenuWindowCascade() throws DebugError;
    void MenuWindowRefresh() throws DebugError;
    void MenuHelpWindow() throws DebugError;
    void MenuHelpContents() throws DebugError;
    void MenuHelpIndex() throws DebugError;
    void MenuHelpAbout() throws DebugError;
    void MenuBreakAtCursor() throws DebugError;
    void MenuBreakClearAll() throws DebugError;
    void MenuBreakEnableAll() throws DebugError;
    void MenuBreakDisableAll() throws DebugError;
    void MenuSearchFind( IDebugWindow w, String what ) throws
DebugError;
    void MenuSearchNext( IDebugWindow w ) throws DebugError;
    void MenuSearchPrev( IDebugWindow w ) throws DebugError;
    void MenuConnectionLogin() throws DebugError;
    void MenuConnectionReleaseSelected() throws DebugError;

    // output window
    void OutputClear();
    void OutputLine( String line );
    void OutputLineNoUpdate( String line );
    void OutputUpdate();

    // Java source search path

    void SetSourcePath( String path ) throws DebugError;
    String GetSourcePath() throws DebugError;

    // Catch java exceptions
    Vector GetCatching();
    void Catch( boolean on, String name ) throws DebugError;

    // Database connections
    int  ConnectionCount();
    void ConnectionRelease( int index );
    void ConnectionAttach( int index );
    String ConnectionName( int index );
    void ConnectionSelect( int index );

    // Login to database
    boolean LoggedIn();
    void Login( String url, String userId, String password, String
userToDebug ) throws DebugError;
    void Logout();

    // Simulate keyboard/mouse actions
    void DeleteItemAt( IDebugWindow w, int row ) throws DebugError;
    void DoubleClickOn( IDebugWindow w, int row ) throws DebugError;

    // Breakpoints
    Object BreakSet( String where ) throws DebugError;
    void  BreakClear( Object b ) throws DebugError;
    void BreakEnable( Object b, boolean enabled ) throws DebugError;
    void BreakSetCount( Object b, int count ) throws DebugError;
```

```
    int  BreakGetCount( Object b ) throws DebugError;
    void BreakSetCondition( Object b, String condition ) throws
DebugError;
    String BreakGetCondition( Object b ) throws DebugError;
    Vector GetBreaks() throws DebugError;

    // Scripting
    void RunScript( String args[] ) throws DebugError;
    void AddEventHandler( DebugScript s );
    void RemoveEventHandler( DebugScript s );

    // Miscellaneous
    void EvalRun( String expr ) throws DebugError;
    void QueryRun( String query ) throws DebugError;
    void QueryMoreRows() throws DebugError;
    Vector GetClassNames();
    Vector GetProcedureNames();
    Vector WindowContents( IDebugWindow window ) throws DebugError;
    boolean AtBreak();
    boolean IsRunning();
    boolean AtStackTop();
    boolean AtStackBottom();
    void SetStatusText( String msg );
    String GetStatusText();
    void WaitCursor();
    void OldCursor();
    void Error( Exception x );
    void Error( String msg );
    void Warning( String msg );
    String Ask( String title );
    boolean MenuIsChecked( String cmd );
    void MenuSetChecked( String cmd, boolean on );
    void AddInspectItem( String s ) throws DebugError;

    // Constants for DebugScript.OnEvent parameter
    public static final int EventBreak = 0;
    public static final int EventTerminate = 1;
    public static final int EventStep = 2;
    public static final int EventInterrupt = 3;
    public static final int EventException = 4;
    public static final int EventConnect = 5;
};
```

sybase.asa.procdebug.IDebugWindow インタフェース

IDebugWindow インタフェースの定義は、次のとおりです。

```
// this interface represents a debugger window
package sybase.asa.procdebug;
public interface IDebugWindow
{
    public int GetSelected();
    /*
        get the currently selected row, or -1 if no selection
    */
```

```
    public boolean SetSelected( int i );
    /*
       set the currently selected row.  Ignored if i < 0 or i > #rows
    */

    public String StringAt( int row );
    /*
       get the String representation of the Nth row of the window.
Returns null if row > # rows
    */

    public java.awt.Rectangle GetPosition();
    public void SetPosition( java.awt.Rectangle r );
    /*
        get/set the windows position within the frame
    */

    public void Close();
    /*
        Close (destroy) the window
    */
}
```

索引

A

B

C

D

E

F

G

I

J

P

Q

R

S

T

U

V

W

あ

い

う

え

お

か

き

く

こ

さ

れ

ろ