SYBASE®
An SAP® Company

ユーザ定義関数

Sybase IQ 15.4

ドキュメント ID： DC01139-01-1540-01
改訂： 2011 年 11 月

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

目次

対象読者

このマニュアル『ユーザ定義関数ガイド』は、Sybase® IQ の機能を拡張する必要のある SQL アナリスト、C 開発者、C++ 開発者、および Java 開発者を対象としています。

開発者の方は、タスク、概念、および API リファレンスの説明を使用して、SQL 以外の外部ユーザ定義関数をプログラミングしてください。

SQL アナリストの方は、このマニュアルを使用して、SQL 以外の外部ユーザ定義関数を参照する SQL クエリを開発してください。

ユーザ定義関数について

Sybase IQ でのユーザ定義関数の使用について説明します。

Sybase IQ では、データベース・コンテナ内で実行されるユーザ定義関数 (UDF: User-Defined Function) を使用できます。UDF の実行機能は、Sybase IQ 内または RAP - The Trading Edition Enterprise® の RAPStore コンポーネント内で使用するオプション・コンポーネントとして提供されます。

これらの外部 C/C++ UDF インタフェースを使用するには、正規のライセンスを取得している必要があります。

これらの外部 C/C++ UDF は、以前のバージョンの Sybase IQ で使用可能な Interactive SQL UDF とは異なります。Interactive SQL UDF は、変更は加えられておらず、使用に際して特別なライセンスも必要ありません。

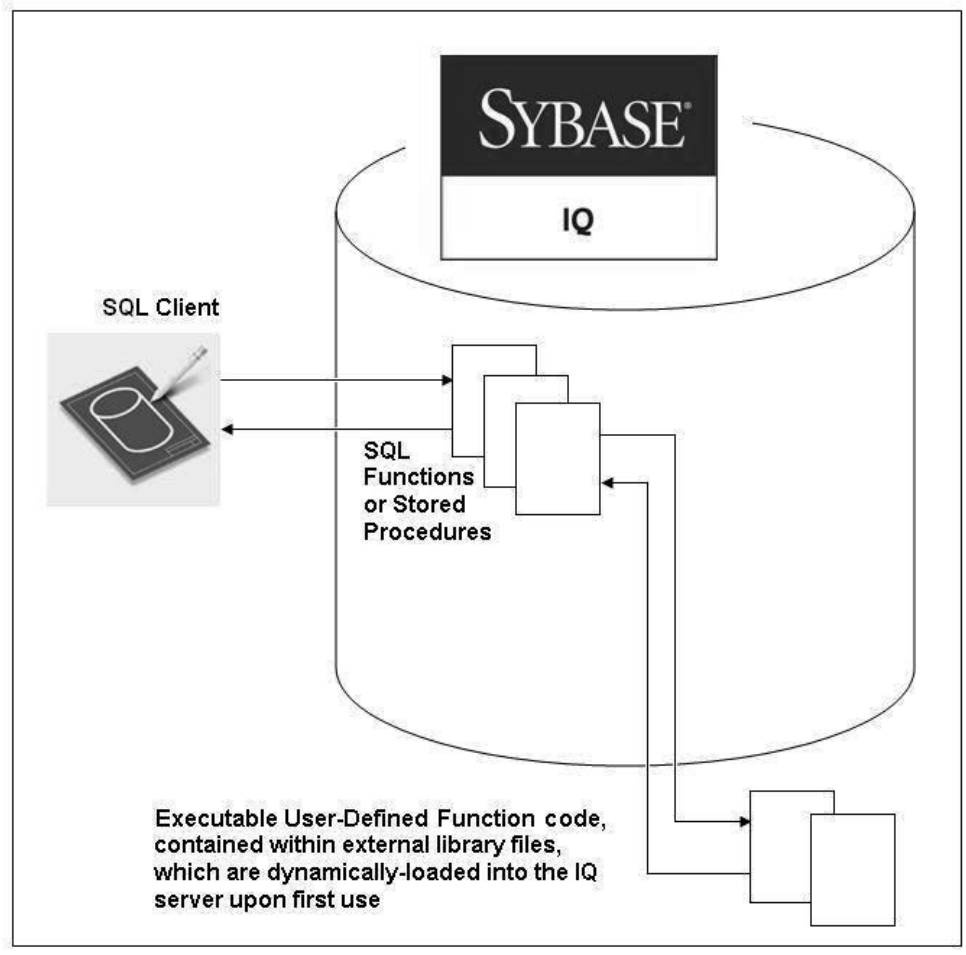
Sybase IQ 内で実行される UDF は、データの分析とプログラミング・ソリューションの柔軟性をユーザに提供しながら、サーバの高度なパフォーマンスを利用します。 ユーザ定義関数は、次の 2 つのコンポーネントで構成されます。

- UDF 宣言
- UDF 実行可能コード

UDF は、パラメータを記述し、外部ライブラリへの参照を提供する SQL 関数またはストアド・プロシージャを使用して SQL 環境内で宣言します。

UDF の実際の実行可能部分は、UDF 宣言関数またはストアド・プロシージャを最初に呼び出すとサーバによって自動的にロードされる外部 (共有オブジェクトまたは動的ロード) ライブラリ・ファイルに含まれています。ライブラリは、いったんロードされるとサーバに常駐するため、そのライブラリを参照する SQL 関数またはストアド・プロシージャの次回以降の呼び出し時にアクセスが高速化されます。

次の図は、Sybase IQ のユーザ定義関数アーキテクチャを示します。



Sybase IQ は、高パフォーマンスのインプロセス外部 C/C++ ユーザ定義関数をサポートしています。このような UDF は、このマニュアルで説明しているインタフェースで C または C++ コードを使用して記述された関数をサポートしています。

UDF の C/C++ ソース・コードは、1 つ以上の外部ライブラリにコンパイルされます。その後、このライブラリは、必要に応じてサーバのプロセス領域にロードされます。UDF を呼び出すメカニズムは、SQL 関数を介してサーバに対して定義されます。この SQL 関数が SQL クエリから呼び出されると、サーバはそのライブラリをロードします (対応するライブラリがロードされていない場合)。

UDF のインストール環境を容易に管理できるように、UDF 開発者は 1 つのライブラリに多数の UDF 関数をパッケージすることをおすすめします。

Sybase IQ には、UDF の構築を容易にする C ベースの API が用意されています。この API は、UDF の定義済みエントリ・ポイントのセット、適切に定義されたコンテキスト・データ構造、および UDF からサーバへの通信メカニズムを提供する一連の SQL コールバック関数で構成されます。Sybase IQ UDF API により、ソフトウェア・ベンダや熟練したエンド・ユーザは独自の UDF の開発、パッケージ、および販売を行うことができます。

学習ロードマップ： UDF の種類

次の表は、Sybase IQ で使用できるユーザ定義関数 (UDF) の種類を示します。

UDF の種類	説明	必要なライセンス	参照先
UDF (SQL)	SQL で作成したユーザ定義関数	なし	『システム管理ガイド：第 2 巻』の「プロシージャとバッチの使用」>「ユーザ定義関数の概要」
C または C++ のスカラ UDF	単一の値に対して演算を行う、v3 の C または C++ の外部プロシージャ	IQ_UDF	学習ロードマップ：外部 C と C++ UDF の種類 (6 ページ)
C または C++ のスカラ UDF	単一の値に対して演算を行う、v4 の C または C++ の外部プロシージャ	IQ_IDA	学習ロードマップ：外部 C と C++ UDF の種類 (6 ページ)
C または C++ の集合 UDF	複数の値に対して演算を行う、v3 の C または C++ の外部プロシージャ。集合 UDF は、UDA または UDAF とも呼ばれます。集合 UDF をコーディングするためのコンテキスト構造体は、スカラ UDF をコーディングするためのコンテキスト構造体とは少し異なります。	IQ_UDF	学習ロードマップ：外部 C と C++ UDF の種類 (6 ページ)
C または C++ の集合 UDF	複数の値に対して演算を行う、v4 の C または C++ の外部プロシージャ。集合 UDF は、UDA または UDAF とも呼ばれます。集合 UDF をコーディングするためのコンテキスト構造体は、スカラ UDF をコーディングするためのコンテキスト構造体とは少し異なります。	IQ_IDA	学習ロードマップ：外部 C と C++ UDF の種類 (6 ページ)

UDF の種類	説明	必要なライセンス	参照先
テーブル UDF	ローのセットを生成し、SQL 文の **FROM** 句でテーブル式として使用できる、C または C++ の外部プロシージャ	IQ_IDA	学習ロードマップ：外部 C と C++ UDF の種類 (6 ページ)
テーブル・パラメータ関数	スカラ・パラメータに加えてテーブル (非スカラ) パラメータを受け取ることができ、ロー・セットの複数パーティションに対して並列に実行できるテーブル UDF。テーブル・パラメータ・ユーザ定義関数ともいいます。	IQ_IDA	学習ロードマップ：外部 C と C++ UDF の種類 (6 ページ)
Java のスカラ UDF	Java コードで実装された、プロセス外 (外部環境) のスカラ・ユーザ定義関数	なし	Java 外部環境 (373 ページ)
Java のテーブル UDF	Java コードで実装された、プロセス外 (外部環境) のテーブル UDF	なし	Java 外部環境 (373 ページ)

学習ロードマップ： 外部 C と C++ UDF の種類

次の表は、IQ_IDA ライセンスで使用できる、高パフォーマンスのインプロセス外部 C/C++ ユーザ定義関数を示します。

v3 の API には IQ_UDF または IQ_IDA ライセンスが必要です。v4 の API には IQ_IDA ライセンスが必要です。

UDF の種類	入力パラメータ	戻り値	API	参照先
スカラ UDF	スカラ	単一のスカラ値	v3、v4	スカラ UDF の宣言と定義 (39 ページ)
集合 UDF	スカラ	単一のスカラ値	v3、v4	集合 UDF の宣言と定義 (54 ページ)
テーブル UDF	スカラ	テーブル	v4	テーブル UDF と TPF (109 ページ)
テーブル・パラメータ関数 (TPF: Table Parameterized Function)	スカラおよびテーブル	テーブル	v4	テーブル・パラメータ関数 (152 ページ)

これらの UDF は、決定的か非決定的のどちらかです。関数の結果は、入力パラメータとデータによって決定されるか (決定的)、または何らかのランダムな動作

によって決定されます (非決定的)。通常、非決定的 UDF のパラメータには、入力パラメータの 1 つとしてランダム・シードが必要になります。

Sybase IQ データベースに準拠したユーザ定義関数

Sybase IQ データベースで使用できるユーザ定義関数を開発できます。

シームレスな実行

UDF は、データベース・コンテナ内でシームレスに実行される必要があります。Sybase IQ は多数のファイルで構成される複雑な製品ですが、主なユーザ対話は、業界標準の SQL (Structured Query Language) を使用してサーバ・プロセス (iqsrv15) 経由で行われます。UDF は、すべて SQL コマンドを使用して実行されます。UDF を使用するために、ユーザが基盤となる実装メソッドを理解する必要はありません。

`EXTFN_V3_API` および `EXTFN_V4_API` は、UDF からメッセージ・ファイル (`.iqmsg`) への書き込みを実現するコールバック関数を提供しています。

UDF は、`EXTFN_V3_API` および `EXTFN_V4_API` で定義されたとおりに、メモリおよび一時的な結果を管理します。

Sybase IQ はマルチユーザ・アプリケーションです。多くのユーザが同じ UDF を同時に実行できます。特定の OLAP クエリによって、UDF が同じクエリ内で何度も実行されたり、ときには並列で実行されたりします。UDF が並列実行されるように設定する方法の詳細については、「集合 UDF パターン呼び出し (96 ページ)」を参照してください。

国際化

Sybase IQ はグローバルな使用向けに国際化されています。エラー・メッセージは外部ファイルに配置されているため、エラー・メッセージを新しい言語にローカライズする場合でも、コードに大幅な変更を加える必要はありません。

複数言語をサポートするためには、UDF も国際化する必要があります。一般に、ほとんどの UDF は数値データに対して実行されます。しかし、UDF は、文字列キーワードを 1 つ以上のパラメータとして受け付ける可能性があります。UDF が使用する例外テキストおよびログ・メッセージに加えて、これらのキーワードを外部ファイルに配置してください。

Sybase IQ は、英語以外のいくつかの言語にもローカライズされています。Sybase IQ が対応している言語へのローカライゼーションをサポートするには、UDF を国際化して、後で別の組織がローカライズできるようにすることをおすすめします。

Sybase IQ の国際言語サポートの詳細については、『Sybase IQ システム管理ガイド』の「各国語と文字セット」を参照してください。

詳細については、www.Sybase.com で『Debugging Using Cross-Character-Set Maps』を参照してください。この資料では、マルチバイト・データのデバッグ方法を、入力キーワード、例外メッセージ、ログ・エントリとの対比で説明しています。

プラットフォームでの相違

UDF は、Sybase IQ でサポートされているさまざまなプラットフォーム上で実行されるように開発できます。Sybase IQ 15.x サーバは 64 ビット・アーキテクチャ上で実行され、MS Windows (64 ビット) オペレーティング・システム・ファミリの複数のプラットフォームでサポートされます。Sybase IQ は、Solaris、HP-UX、AIX、Linux を含む各種 UNIX (64 ビット) でもサポートされます。

回避する必要がある方法

ユーザ定義関数の適切な作成方法について説明します。

- SQL 登録スクリプト内でライブラリ・パスをハードコードしないでください。このような方法を使用すると、ユーザは Sybase IQ と同じディレクトリに UDF をインストールできなくなり、柔軟性が損なわれます。
- UDF の呼び出しをキャンセルするメカニズムをユーザに提供しない場合は、曖昧なコードや、予期せず永久にループする可能性のある構造を記述しないでください (**'get_is_cancelled()'** 関数を参照)。
- 呼び出しごとに繰り返される複雑な演算、またはメモリを集中的に使用する演算を実行しないでください。何千ものローを含むテーブルに対して UDF 呼び出しを実行する場合、効率的に実行することが最も重要になります。ロー単位でなく、一度に 1000 ～数千のローにメモリのブロックを割り付けることをおすすめします。
- UDF 内からデータベース接続を開いたり、データベース操作を実行したりしないでください。UDF の実行に必要なパラメータおよびデータは、すべてパラメータとして UDF に渡す必要があります。
- UDF に名前を付ける際に、予約語を使用しないでください。

ユーザ定義関数の命名規則

UDF 名は、Sybase IQ 内の他の識別子と同じ制限に従っている必要があります。

Sybase IQ の識別子の最大長は 128 バイトです。容易に使用できるように、UDF 名はアルファベット文字で始めます。Sybase IQ で定義されているアルファベット文字は、アルファベット、アンダースコア (_)、アット・マーク (@)、シャープ記号

(#)、ドル記号 ($) です。UDF 名は、これらのアルファベット文字および数字 (0 ～ 9) だけで構成されている必要があります。UDF 名は、SQL 予約語と競合しないようにします。SQL 予約語のリストについては、Sybase IQ の『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」にある「予約語」を参照してください。

他の識別子と同様に、予約語、スペース、上記のリストに記載されていない文字を UDF 名に含めたり、UDF 名を英字以外の文字で始めたりすることも可能ですが、この方法は推奨されません。UDF 名にこれらの文字を含める場合は、引用符または角カッコで囲みます。ただし、使い勝手は悪くなります。

UDF は、他の SQL 関数およびストアド・プロシージャと同じネーム・スペースに存在します。既存のストアド・プロシージャおよび関数との競合を回避するには、プレフィクスとして、ユニークな短い (2 ～ 5 文字) 頭字語とアンダースコアを UDF 名の前に付けます。ローカル環境内ですでに定義されている他の SQL 関数およびストアド・プロシージャと競合しない UDF 名を選択してください。

次のプレフィックスは、すでに使用されています。

- **debugger_tutorial** – Sybase IQ のネイティブ・インストール環境とともに提供されるストアド・プロシージャ。
- **ManageContacts** – Sybase IQ のデモ・データベースとともに提供されるストアド・プロシージャ。
- **Show** – Sybase IQ のデモ・データベースのデータを表示するために使用するストアド・プロシージャ。
- **sp_Detect_MPX_DDL_conflicts** – Sybase IQ のネイティブ・インストール環境とともに提供されるストアド・プロシージャ。
- **sp_iqevbegintxn** – Sybase IQ のネイティブ・インストール環境とともに提供されるストアド・プロシージャ。
- **sp_iqmpx** – マルチプレックス管理を支援するために Sybase IQ によって提供される関数とストアド・プロシージャ。
- **ts_** – オプションの金融時系列関数と予測関数。

SQL データ型

UDF 宣言は、特定の SQL データ型のみをサポートします。

UDF 宣言では、UDF の引数のデータ型として、または戻り値のデータ型として、次の SQL データ型を使用できます。

SQL データ型	C/C++ データ型識別子	C/C++ typedef	説明
UNSIGNED BIGINT	DT_UNSBIGINT	a_sql_uint64	記憶領域を 8 バイト必要とする符号なし 64 ビット整数です。
BIGINT	DT_BIGINT	a_sql_int64	記憶領域を 8 バイト必要とする符号付き 64 ビット整数です。
UNSIGNED INT	DT_UNSINT	a_sql_uint32	記憶領域を 4 バイト必要とする符号なし 32 ビット整数です。
INT	DT_INT	a_sql_int32	記憶領域を 4 バイト必要とする符号付き 32 ビット整数です。
SMALLINT	DT_SMALLINT	short	記憶領域を 2 バイト必要とする符号付き 16 ビット整数です。
TINYINT	DT_TINYINT	unsigned char	記憶領域を 1 バイト必要とする符号なし 8 ビット整数です。
DOUBLE	DT_DOUBLE	double	記憶領域を 8 バイト必要とする符号付き 64 ビット倍精度浮動小数点数です。
REAL	DT_FLOAT	float	記憶領域を 4 バイト必要とする符号付き 32 ビット浮動小数点数です。

SQL データ型	C/C++ データ型識別子	C/C++ typedef	説明
FLOAT	DT_FLOAT	float	SQL では、関連する精度に応じて、FLOAT は、記憶領域を 4 バイト必要とする符号付き 32 ビット浮動小数点数か、記憶領域を 8 バイト必要とする符号付き 64 ビット倍精度浮動小数点数のどちらかになります。 FLOAT データ型のオプションの精度が指定されていない場合、UDF 宣言でのみ、SQL データ型 FLOAT を使用できます。精度が指定されていない場合、 FLOAT は REAL と同義になります。
CHAR(<n>)	DT_FIXCHAR	char	データベースのデフォルトの文字セット内の、ブランクを埋め込まれた固定長の文字列です。 最大長 "<n>" は 32767 です。データは NULL バイトで終了できません。
VARCHAR(<n>)	DT_VARCHAR	char	データベースのデフォルトの文字セット内の、可変長の文字列です。 最大長 "<n>" は 32767 です。データは NULL バイトで終了できません。 UDF 入力引数の値が NULL でない場合、実際の長さは、an_extfn_value 構造体の *total_len* フィールドから取得する必要があります。 同様に、この種類の UDF 結果について、実際の長さを *total_len* フィールドに設定する必要があります。

SQL データ型	C/C++ データ型識別子	C/C++ typedef	説明
LONG VARCHAR(<n>) または CLOB	DT_VARCHAR	char	データベースのデフォルトの文字セット内の、可変長の文字列です。 LONG VARCHAR データ型は入力引数でのみ使用します。戻り値のデータ型では使用しません。最大長 "<n>" は、v3 UDF では 4GB (ギガバイト) です。 データは NULL バイトで終了できません。 LONG VARCHAR データ型は WD インデックスまたは TEXT インデックスを持つことができます。UDF 入力引数の値が NULL でない場合、実際の長さは、an_extfn_value 構造体の *total_len* フィールドから取得する必要があります。 既存のスカラ UDF または集合 UDF に `get_value()` メソッドと `get_piece()` メソッドを介して値の各部分を読み取るループが含まれている場合は、LOB データ型を入力パラメータとして使用するために、その関数を再作成する必要も再コンパイルする必要もありません。 ループは、*remain_len* が 0 より大きくなるまで、または v3 UDF では 4GB に達するまで続行します (v4 では 4GB の上限はありません)。 テーブル UDF と TPF では、データの処理と取得に `get_piece()` メソッドを使用しません。テーブル

SQL データ型	C/C++ データ型識別子	C/C++ typedef	説明
			UDF と TPF では、代わりに `Blob (a_v4_extfn_blob)` API を使用します。入力パラメータの長さの判別には `blob_length` を使用します。 ラージ・オブジェクト・データのサポートには、別途ライセンスが必要な Sybase IQ オプションが必要です。
`BINARY`(<n>)	DT_BINARY	unsigned char	NULL バイトが埋め込まれた固定長のバイナリです。値の最大バイナリ長 "<n>" は 32767 です。データは NULL バイトで終了できません。
`VARBINARY`(<n>)	DT_BINARY	unsigned char	可変長のバイナリ値です。値の最大長 "<n>" は 32767 です。データは NULL バイトで終了できません。 UDF 入力引数の値が NULL でない場合、実際の長さは、an_extfn_value 構造体の *total_len* フィールドから取得する必要があります。同様に、この型の UDF 結果については、実際の長さを *total_len* フィールドに設定する必要があります。データは NULL バイトで終了できません。

SQL データ型	C/C++ データ型識別子	C/C++ typedef	説明
LONG BINARY(<n>) または BLOB	DT_BINARY	unsigned char	NULL バイトが埋め込まれた固定長のバイナリです。値の最大バイナリ長 "<n>" は、v3 UDF では 4GB (ギガバイト) です。 LONG BINARY データ型は入力引数でのみ使用します。戻り値のデータ型では使用しません。 既存のスカラ UDF または集合 UDF に get_value() メソッドと get_piece() メソッドを介して値の各部分を読み取るループが含まれている場合は、LOB データ型を入力パラメータとして使用するために、その関数を再作成する必要も再コンパイルする必要もありません。 ループは、*remain_len* が 0 より大きくなるまで、または v3 UDF では 4GB に達するまで続行します (v4 では 4GB の上限はありません)。 テーブル UDF と TPF では、データの処理と取得に get_piece() メソッドを使用しません。テーブル UDF と TPF では、代わりに Blob (a_v4_extfn_blob) API を使用します。入力パラメータの長さの判別には blob_length を使用します。 ラージ・オブジェクト・データのサポートには、別途ライセンスが必要な

SQL データ型	C/C++ データ型識別子	C/C++ typedef	説明
			Sybase IQ オプションが必要です。

SQL データ型	C/C++ データ型識別子	C/C++ typedef	説明
DATE	DT_TIMESTAMP_STRUCT	unsigned integer	符号なし整数として UDF との間で受け渡しされる暦日値です。UDF に渡された値は、比較／ソート操作で使用できることが保証されます。大きい値は後の日付を示します。実際の日付部分が必要な場合、DT_TIMESTAMP_STRUCT 型に変換するために、UDF は `convert_value` を呼び出す必要があります。この日付型は、次のような構造で日付と時刻を表します。 `typedef struct sqldatetime { unsigned short year; /* e.g. 1992*/ unsigned char month; /* 0-11 */ unsigned char day_of_week; /* 0-6 0=Sunday, 1=Monday, ... */ unsigned short day_of_year; /* 0-365 */ unsigned char day; /* 1-31 */ unsigned char hour; /* 0-23 */ unsigned char minute; /* 0-59 */ unsigned char second; /* 0-59 */`

SQL データ型	C/C++ データ型識別子	C/C++ typedef	説明
			`a_sql_uint32 microsecond; /* 0-999999 */ } SQLDATETIME;`
	DT_TIMESTAMP_STRUCT	unsigned bigint	指定された日付内のある時刻を正確に表す値です。UDF に渡された値は、比較／ソート操作で使用できることが保証されます。大きい値は後の時刻を示します。実際の時刻部分が必要な場合、`DT_TIMESTAMP_STRUCT` 型に変換するために、UDF は `convert_value` を呼び出す必要があります。
`DATETIME`、`SMALLDATETIME`、または `TIMESTAMP` のいずれか	DT_TIMESTAMP_STRUCT	unsigned bigint	暦日と時刻の値です。UDF に渡された値は、比較／ソート操作で使用できることが保証されます。大きい値は後の日時を示します。実際の時刻部分が必要な場合、`DT_TIMESTAMP_STRUCT` 型に変換するために、UDF は `convert_value` を呼び出す必要があります。
`TABLE`	DT_EXTFN_TABLE	a_v4_extfn_table	入力のテーブル・パラメータの結果セットを表します。このデータ型は TPF でのみ使用できます。

サポートしてないデータ型
UDF 宣言では、UDF の引数のデータ型として、または戻り値のデータ型として、次の SQL データ型を使用できません。

- `BIT` – 通常、 `TINYINT` データ型 として UDF 宣言で処理され、`BIT` からの暗黙のデータ型変換によって自動的に値変換が処理されます。
- `DECIMAL`(<precision>, <scale>) または `NUMERIC`(<precision>, <scale>) – 使用方法により、`DECIMAL` は、通常は `DOUBLE` データ型として処理されます。ただし、

INT または BIGINT データ型を使用できるようにするには、さまざまな規則が適用されます。

- LONG VARCHAR (CLOB) – 入力引数でのみサポートされています。戻り値のデータ型ではサポートされていません。パススルー TPF には例外があり、LONG VARCHAR が戻り値のデータ型としてサポートされています。
- LONG BINARY (BLOB) – 入力引数でのみサポートされています。戻り値のデータ型ではサポートされていません。パススルー TPF には例外があり、LONG BINARY が戻り値のデータ型としてサポートされています。
- TEXT – 現在サポートされていません。

参照：

- BLOB (a_v4_extfn_blob) (219 ページ)
- BLOB 入力ストリーム (a_v4_extfn_blob_istream) (222 ページ)
- convert_value (326 ページ)
- テーブル (a_v4_extfn_table) (337 ページ)

UDF の作成

UDF の設計、作成、およびテストについて説明します。

ユーザ定義関数の設計の基本

UDF を開発するときには、いくつかの基本的な注意事項に留意する必要があります。

このマニュアルは、適切なプログラムの設計と開発、独立テストなど、ソフトウェア開発の基本に UDF の開発者が精通していることを前提としています。

標準のソフトウェア開発方法に加えて、Java 以外の UDF の開発者は、Sybase IQ データベース・コンテナ内で実行されるコードを開発中であることを常に意識し、データベース・コンテナによって課される制限事項を理解する必要があります。

また、集合 UDF の開発者は、OLAP クエリと、OLAP クエリが UDF パターン呼び出しに変換されるしくみを理解しておく必要もあります。

UDF は複数のスレッドによって同時に呼び出される可能性があるため、スレッドセーフであるように作成してください。

サンプル・コード

この製品には、UDF のサンプル ソース コードが用意されています。最新バージョンの Sybase IQ には、常に最新バージョンのソース・コードが付属します。

UNIX プラットフォームの場合、UDF のサンプル・コードは `$SYBASE/IQ-15_4/samples/udf` にあります ($SYBASE はインストール・ルート)。

Windows プラットフォームの場合、UDF のサンプル・コードは次の場所にあります。`C:¥Documents and Settings¥All Users¥SybaseIQ¥samples¥udf`

この『ユーザ定義関数ガイド』に記載されている UDF のサンプル コードは、Sybase IQ 製品に付属する最新バージョンのものと異なる可能性があります。 UDF のサンプル・ソース・コードに対する最新の変更については、対応するオペレーティング・システム・プラットフォームの Sybase IQ リリース・ノートに記載されています。

ダイナミック・ライブラリ・インタフェースの設定

ダイナミック・リンク・ライブラリで使用するインタフェース・スタイルを指定します。

動的にロードされたライブラリにはそれぞれ、次の定義のコピーが 1 つ含まれている必要があります。

```
extern "C" a_sql_uint32 extfn_use_new_api(void )
{
     return EXTFN_V4_API;
}
```

この定義は、使用しているインタフェース・スタイルと、このダイナミック・リンク・ライブラリで定義される UDF へのアクセス方法をサーバに示しています。高パフォーマンスの UDF については、新しいインタフェース・スタイル `EXTFN_V3_API` と `EXTFN_V4_API` のみがサポートされています。

v4 API へのアップグレード

15.4 が装備している v4 API にアップグレードすることをおすすめします。

前提条件
Sybase IQ サーバのバージョン 15.4 をインストールします。

手順

Sybase IQ サーバのバージョン 15.1、15.2、または 15.3 向けに開発した既存のスカラ UDF または集合 UDF がある場合、それらの UDF は v3 API のインタフェース・スタイルを使用しており、`extfnapiv3.h` ヘッダ・ファイルを参照しています。`extfnapiv4.h` ヘッダ・ファイルを参照するように、従来の C または C++ 外部ライブラリ・ファイルを修正します。

既存の v3 スカラ関数および集合関数は、設計どおりに引き続き動作します。ただし、PlexQ でスカラおよび集合の分散を利用するためには、ヘッダ・ファイルとライブラリ・バージョンを v4 にアップグレードする必要があります。スカラ関数または集合関数の typedef の名前を変更する必要はありません。

1. スカラ UDF または集合 UDF を定義している C または C++ 外部ライブラリ・ファイルを開きます。

2. `#include 'extfnapiv3.h'` と記述している箇所を検索し、すべて `#include 'extfnapiv4.h'` に変更します。
3. ダイナミック・ライブラリ・インタフェースを `EXTFN_V4_API` に設定します。
4. 再構築します。

次のステップ
Sybase パートナーは、ライブラリで `extfn_get_license_info` をエントリ・ポイントとしてエクスポートする必要があります。

参照：
- 外部関数のプロトタイプ (105 ページ)
- ライセンス情報 (a_v4_extfn_license_info) (331 ページ)
- 集合 UDF の定義 (63 ページ)
- Defining a Scalar UDF (46 ページ)
- テーブル UDF の開発 (115 ページ)
- TPF の開発 (153 ページ)

ライブラリ・バージョン (extfn_get_library_version)

`extfn_get_library_version` メソッドを使用して、現在のマルチプレックス・ノードからライブラリ・バージョンを取得します。サーバがマルチプレックス・ノード間でクエリの分割を検討するのは、インストールされているライブラリが他のノードと互換性がある場合のみです。

実装
v4 ライブラリは、このオプションのエントリ・ポイントを定義できます。

```
size_t extfn_get_library_version( uint8 *buff, size_t len );
```

説明
ライブラリ・バージョンのメソッドはライブラリ・レベルにあり、メソッド名に `a_v4` プレフィクスは付きません。

v4 ライブラリがこのオプションのエントリ・ポイントを定義している場合、サーバは他のノードにクエリを分散できます。このエントリ・ポイントは、指定のバッファにライブラリ・バージョン文字列 (ASCII 文字のみを含み、末尾に **¥0** が付いた C のスタイルの文字列) を設定し、設定したバージョン文字列の実際のサイズを戻り値で返します。この文字列の上限は 256 バイトです。

エントリ・ポイントが定義されていない場合、サーバはマルチプレックスの他のノードに UDF を分散しません。

参照：
- ライブラリ・バージョンの互換性 (extfn_check_version_compatibility) (22 ページ)
- ダイナミック・ライブラリ・インタフェースの設定 (20 ページ)

ライブラリ・バージョンの互換性 (extfn_check_version_compatibility)

`extfn_check_version_compatibility` メソッドを使用して、マルチプレックスのノード間のライブラリ・バージョンの互換性の基準を定義します。

実装
v4 ライブラリは、このオプションのエントリ・ポイントを定義できます。

```
a_bool extfn_check_version_compatibility( uint8 *buff, size_t
len );
```

説明
ライブラリ・バージョンのメソッドはライブラリ・レベルにあり、メソッド名に `a_v4` プレフィクスは付きません。

このオプションのエントリ・ポイントは、バージョン文字列が格納されたバッファとバージョン文字列の長さを受け取り、 ターゲット・ノードのライブラリ・バージョンとバージョン文字列パラメータに互換性があるかどうかを返します。互換性の基準はライブラリ開発者が定義します。

`extfn_get_library_version` の使用
リーダ・ノードは、バージョンの互換性をチェックする前に `extfn_get_library_version` を呼び出します。リーダ・ノードに `extfn_get_library_version` が実装されていない場合、分散は行われません。リーダ・ノードに `extfn_get_library_version` が実装されている場合、UDF または TPF には分散の対象としての資格があります。分散の対象としての資格があっても、分散クエリ処理が必ず行われるとは限りません。

`extfn_get_library_version` メソッドは長さ 0 の文字列を返す場合があります。しかしこれは `extfn_get_library_version` が実装されていないという意味にはなりません。

注意： `extfn_get_library_version` が長さ 0 の文字列を返した場合でも、TPF または UDF には分散の対象としての資格があります。

`extfn_get_library_version` が長さ 0 の文字列を返した場合、ワーカ・ノードが分散処理を受け入れるかどうかは、ワーカ・ノードでの

extfn_check_version_compatibility の実装によります。分散処理を実行するためには、互換性のあるライブラリがワーカ・ノードに必要です。

参照：
- ライブラリ・バージョン (extfn_get_library_version) (21 ページ)
- ダイナミック・ライブラリ・インタフェースの設定 (20 ページ)

ライセンス情報 (extfn_get_license_info)

Sybase デザイン・パートナーは、extfn_get_license_info ライブラリレベル関数を実装して、サーバが v4 UDF からライセンス情報を取得できるようにします。

データ型
an_extfn_license_info

実装
```
(_entry an_extfn_get_license_info) ( an_extfn_license_info
**license_info );
```

パラメータ
license_info は、ライブラリから受け取ったライセンス情報を返す出力パラメータです。 ライセンス情報は a_v4_extfn_license_info 構造体で定義します。

説明
Sybase パートナーは、Sybase が提供したライセンス・キーを a_v4_extfn_license_info 構造体で指定し、ライブラリで extfn_get_license_info をエントリ・ポイントとしてエクスポートする必要があります。

extfn_get_license_info メソッドの追加

Sybase デザイン・パートナーは、a_v4_extfn_license_info に文字列を設定し、extfn_get_license_info を v4 エントリ・ポイントとして定義します。

1. a_v4_extfn_license_info 構造体で、会社名を指定します。最大長は 255 文字です。
2. a_v4_extfn_license_info 構造体で、ライブラリ・バージョンやビルド番号などの追加的なライブラリ情報を指定します。最大長は 255 文字です。

3. `a_v4_extfn_license_info` 構造体で、Sybase から提供されたライセンス・キーを入力します。
4. ライブラリで `extfn_get_license_info` をエントリ・ポイントとしてエクスポートします。

```
a_v4_extfn_license_info my_info = {
    1,
    "Company Name",
    "Library Info String",
    (void *)"KEY_STRING"
};

void SQL_CALLBACK extfn_get_license_info( an_extfn_license_info
**license_info )
/
**********************************************************************
**********************/
{
    *license_info = (an_extfn_license_info *)& my_info;
}
```

ダイナミック・リンク・ライブラリを構築するためのソース・コードのコンパイルとリンク

任意のユーザ定義関数のダイナミック・リンク・ライブラリを構築する場合、コンパイルおよびリンク・スイッチを使用します。

1. UDF ダイナミック・リンク・ライブラリには、関数 "**extfn_use_new_api()**" の実装が必要です。この関数のソース・コードは、「ダイナミック・ライブラリ・インタフェースの設定」 (20 ページ)」に記載されています。この関数は、ライブラリ内のすべての関数が準拠している API スタイルのサーバを通知します。サンプル・ソース・ファイル "`my_main.cxx`" にはこの関数が含まれており、修正せずに使用できます。
2. UDF ダイナミック・リンク・ライブラリには、少なくとも 1 つの UDF 関数のオブジェクト・コードも必要です。UDF ダイナミック・リンク・ライブラリは、オプションで複数の UDF を含めることができます。
3. 各 UDF のオブジェクト・コードと **extfn_use_new_api()** をリンクして、1 つのライブラリを構築します。
   たとえば、ライブラリ "libudfex" を構築するための手順は、次のとおりです。
   - 各ソース・ファイルをコンパイルし、オブジェクト・ファイルを生成します。

     ```
     my_main.cxx
     my_bit_or.cxx
     my_bit_xor.cxx
     my_interpolate.cxx
     ```

```
my_plus.cxx
my_plus_counter.cxx
my_sum.cxx
my_byte_length.cxx
my_md5.cxx
my_toupper.cxx
tpf_agg.cxx
tpf_blob.cxx
tpf_dt.cxx
tpf_filt.cxx
tpf_oby.cxx
tpf_pby.cxx
tpf_rg_1.cxx
tpf_rg_2.cxx
udf_blob.cxx
udf_main.cxx
udf_rg_1.cxx
udf_rg_2.cxx
udf_rg_3.cxx
udf_utils.cxx
```

- 生成された各オブジェクトをリンクし、1 つのライブラリを構築します。

ダイナミック・リンク・ライブラリのコンパイルおよびリンクが完了した後、次のタスクのいずれかを完了します。

- (推奨) UDF ライブラリの明示的なパス名が含まれるように、**CREATE FUNCTION ... EXTERNAL NAME** または **CREATE PROCEDURE ... EXTERNAL NAME** を更新します。
- すべての Sybase IQ ライブラリが格納されるディレクトリに、UDF ライブラリ・ファイルを配置します。
- UDF ライブラリの場所を含むライブラリ・ロード・パスで、サーバを起動します。

  UNIX では、`start_iq startup` スクリプト内の LD_LIBRARY_PATH を変更します。 すべての UNIX 系 OS で LD_LIBRARY_PATH が一般的に使用されますが、HP では SHLIB_PATH が、AIX では LIB_PATH が優先的に使用されます。

  UNIX プラットフォームでは、LD_LIBRARY_PATH が使用されない場合に、外部名の指定に完全修飾名を含めることができます。 Windows プラットフォームでは、完全修飾名は使用できず、ライブラリ検索パスは PATH 環境変数で定義されます。

4. Windows では `iqdir15/samples/udf/build.bat` を実行します。UNIX では `iqdir15/samples/udf/build.sh` を実行します。

Windows でのサンプル UDF のコンパイルとリンク

build.bat スクリプトを実行して、samples¥udf ディレクトリに格納されているサンプルのスカラ UDF、集合 UDF、テーブル UDF、TPF をコンパイルおよびリンクします。

1. %ALLUSERSPROFILE%¥samples¥udf に移動します。
2. build.bat を実行します。

パラメータ	説明
-clean	オブジェクトとビルド・ディレクトリを削除します。
-v3	サンプルのスカラ UDF と集合 UDF を v3 API でビルドします。
-v4	(デフォルト) サンプルのテーブル UDF と TPF を v4 API でビルドします。

UNIX でのサンプル UDF のコンパイルとリンク

build.sh スクリプトを実行して、samples/udf ディレクトリに格納されているサンプルのスカラ UDF、集合 UDF、テーブル UDF、TPF をコンパイルおよびリンクします。

1. $IQDIR15/samples/udf に移動します。
2. build.sh を実行します。

パラメータ	説明
-clean	オブジェクトとビルド・ディレクトリを削除します。
-v3	サンプルのスカラ UDF と集合 UDF を v3 API でビルドします。
-v4	(デフォルト) サンプルのテーブル UDF と TPF を v4 API でビルドします。

AIX のスイッチ

AIX で共有ライブラリを構築する場合、次のコンパイルおよびリンク・スイッチを使用します。

xlC 10.0 (PowerPC)

重要： 各 UDF ライブラリに extfn_use_new_api() のコードを含めてください。

注意： AIX 6.1 システム上でコンパイルする場合、xlC コンパイラの最小レベルは 10.0 です。

[コンパイル・スイッチ]

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

[リンク・スイッチ]

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

HP-UX のスイッチ

HP-UX で共有ライブラリを構築する場合、次のコンパイルおよびリンク・スイッチを使用します。

aCC 6.24 (Itanium)

重要： 各 UDF ライブラリに extfn_use_new_api() のコードを含めてください。

[コンパイル・スイッチ]

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

[リンク・スイッチ]

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

Linux のスイッチ

Linux で共有ライブラリを構築する場合、次のコンパイルおよびリンク・スイッチを使用します。

g++ 4.1.1 (x86)

重要： 各 UDF ライブラリに extfn_use_new_api() のコードを含めてください。

[コンパイル・スイッチ]

```
-fPIC -fsigned-char -fno-exceptions -pthread -fno-omit-frame-
pointer
-Wno-deprecated -Wno-ctor-dtor-privacy -O2 -Wall
```

注意： C++ アプリケーションをコンパイルして Linux 上に共有ライブラリを構築するときは、**-O2** スイッチと **-Wall** スイッチを UDF コンパイル・スイッチのリストに追加すると、計算時間が短くなります。

[リンク・スイッチ]

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

注意： gcc は、Linux でも使用できます。**gcc** とリンクすると同時に、`-lstdc++` をリンク・スイッチに追加し、C++ ランタイム・ライブラリでリンクします。

[例]

- *例 1*

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

- *例 2*

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

- *例 3*

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

xlC 10.0 (PowerPC)

[コンパイル・スイッチ]

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

[リンク・スイッチ]

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

Solaris のスイッチ

Solaris で共有ライブラリを構築する場合、次のコンパイルおよびリンク・スイッチを使用します。

Sun Studio 12 (SPARC)

重要： 各 UDF ライブラリに extfn_use_new_api() のコードを含めてください。

[コンパイル・スイッチ]

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

[リンク・スイッチ]

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

Sun Studio 12 (x86)

[コンパイル・スイッチ]

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

[リンク・スイッチ]

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

Windows のスイッチ

Windows で共有ライブラリを構築する場合、次のコンパイルおよびリンク・スイッチを使用します。

Visual Studio 2008 (x86)

重要： 各 UDF ライブラリに extfn_use_new_api() のコードを含めてください。

[コンパイルおよびリンク・スイッチ]

次に、my_plus 関数を含む DLL の例を示します。DLL に含まれる UDF ごとに、記述子関数の EXPORT スイッチを含める必要があります。

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

[例]

環境のセットアップ

```
set VCBASE=c:¥dev¥vc9
      set MSSDK=C:¥dev¥mssdk6.0a
      set IQINSTALLDIR=C:¥Sybase¥IQ
      set OBJ_DIR=%IQINSTALLDIR%¥IQ-15_4¥samples¥udf¥objs
      set SRC_DIR=%IQINSTALLDIR%¥IQ-15_4¥samples¥udf¥src
      call %VCBASE%¥VC¥bin¥vcvars32.bat
```

- *例 1*

  ```
  %VCBASE%¥VC¥bin¥amd64¥cl -c -nologo -DNDEBUG -DWINNT -D_USRDLL
        -D_WINDLL -D_WIN64 -DWIN64 -
  D_WIN32_WINNT=_WIN32_WINNT_WINXP
       -DWINVER=_WIN32_WINNT_WINXP -D_MBCS -GS -W3 -Zi -favor:AMD64
  ```

```
      -DSYB_LP64 -D_LARGEFILE_SOURCE -D_FILE_OFFSET_BITS=64 -
DHMSWNT
      -D_CRT_SECURE_NO_DEPRECATE -D_CRT_NONSTDC_NO_DEPRECATE
      -DPOINTERS_ARE_64BITS -DLONG_IS_64BITS -
D_RWSTD_NO_EXCEPTIONS
      -I"%VCBASE%¥VC¥include" -I"%MSSDK%¥include "-I"%MSSDK%¥Lib
¥AMD64"
      -I"%VCBASE%¥VC¥lib¥amd64" -DMSDCXX -DINT64_WORKAROUND
      -DSUPPORTS_UDAF -Od -Zi -MD -I"%IQINSTALLDIR%¥IQ-15_4¥sdk
¥include"
     -Fo"%OBJ_DIR%¥my_interpolate.o" %SRC_DIR%¥my_interpolate.cxx
```

- *例 2*

```
%VCBASE%¥VC¥bin¥amd64¥cl -c -nologo -DNDEBUG -DWINNT -D_USRDLL
      -D_WINDLL -D_WIN64 -DWIN64 -
D_WIN32_WINNT=_WIN32_WINNT_WINXP
     -DWINVER=_WIN32_WINNT_WINXP -D_MBCS -GS -W3 -Zi -favor:AMD64
      -DSYB_LP64 -D_LARGEFILE_SOURCE -D_FILE_OFFSET_BITS=64 -
DHMSWNT
      -D_CRT_SECURE_NO_DEPRECATE -D_CRT_NONSTDC_NO_DEPRECATE
      -DPOINTERS_ARE_64BITS -DLONG_IS_64BITS -
D_RWSTD_NO_EXCEPTIONS
      -I"%VCBASE%¥VC¥include" -I"%MSSDK%¥include "-I"%MSSDK%¥Lib
¥AMD64"
      -I"%VCBASE%¥VC¥lib¥amd64" -DMSDCXX -DINT64_WORKAROUND
      -DSUPPORTS_UDAF -Od -Zi -MD -I"%IQINSTALLDIR%¥IQ-15_4¥sdk
¥include"
      -Fo"%OBJ_DIR%¥my_main.o" %SRC_DIR%¥my_main.cxx
```

- *例 3*

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

- *例 4*

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

ユーザ定義関数のテスト

UDF の外部コードのコーディング、コンパイル、リンク、および対応する SQL 関数とストアド・プロシージャの定義が完了したら、UDF をテストできます。

データベースには、非常に高い信頼性が要求されます。 データベース環境内で実行される UDF では、このような高いレベルの信頼性を維持する必要があります。UDF API の最初の実装では、UDF は Sybase IQ サーバ内で実行されます。UDF が途中で (または予期せず) アボートすると、Sybase IQ サーバがアボートする場合があります。開発環境またはテスト環境における徹底的なテストにより、どのような状況であっても、UDF が途中で終了したり、予期せずアボートしたりしないことを確認してください。

ユーザ定義関数の有効化および無効化

`inmemory_external_procedure` セキュリティ機能を使用して、サーバによる高パフォーマンスのインプロセス UDF の使用を有効または無効にできます。

データベースでは、データの整合性が維持される必要があります。いかなる事情があっても、データの消失、変更、増補、または破損は許されません。UDF は Sybase IQ サーバ内で実行されるため、データが破損する危険性があります。メモリ管理およびその他のポインタの使用に注意してください。読み込み専用のマルチプレックス・ノード内で UDF をインストールして実行することを強くおすすめします。保護を強化するために、各サーバでセキュア機能 **(-sf)** 起動オプションを使用して UDF の実行を有効または無効にしてください。

注意： デフォルトでは、マルチプレックス・ライタおよびコーディネータ・ノード上での UDF の実行は無効になっています。その他のすべてのノードは、デフォルトで有効になっています。

管理者は、サーバの起動コマンドまたは設定ファイル内で次のように指定すると、v3 UDF と v4 UDF を有効にできます。

```
-sf -inmemory_external_procedure
```

管理者は、サーバの起動コマンドまたは設定ファイル内で次のように指定すると、v3 UDF と v4 UDF を無効にできます。

```
-sf inmemory_external_procedure
```

-sf フラグに関する追加の情報は、『SQL Anywhere サーバ - データベース管理』に記載されています。SQL Anywhere のマニュアルに記載されている値は Sybase IQ には該当しないため、使用しないでください。

ユーザ定義関数の初回の実行

できる限り安全な環境を確保するために、マルチプレックス・インストール環境では、読み込み専用サーバ・ノードから UDF のインストールおよび呼び出しを行うことをおすすめします。

Sybase IQ サーバは、UDF が最初に呼び出されるまで、UDF コードが含まれるライブラリをロードしません。ロードされていないライブラリに存在する UDF の初回実行速度は、通常より遅くなる可能性があります。ライブラリがロードされると、同じ UDF または同じライブラリに含まれる別の UDF の以降の呼び出しは、予期されたパフォーマンスで実行されます。

- **ストアド・プロシージャ SA_EXTERNAL_LIBRARY_UNLOAD を使用するライブラリ** – これらのライブラリは、Sybase IQ サーバが停止して再開するときに、再ロードされません。

業務時間後のメンテナンス操作で IQ サーバの停止および再起動が必要になる環境では、サーバの再起動後にテスト・クエリを実行します。これにより、適切なライブラリがメモリにロードされ、業務時間帯に最適なクエリ・パフォーマンスが得られます。

外部ライブラリの管理

各外部ライブラリは、ライブラリを必要とする UDF が最初に呼び出されたときにロードされます。ロードされたライブラリは、サーバの使用期間中、ロードされたままです。**CREATE FUNCTION** または **CREATE PROCEDURE** の呼び出しが実行されてもロードされず、**DROP FUNCTION** または **DROP PROCEDURE** の呼び出しが実行されても自動的にアンロードされません。

ライブラリ・バージョンの更新が必要な場合、サーバを再起動しなくても、**dbo.sa_external_library_unload** プロシージャにより、ライブラリが強制的にアンロードされます。外部ライブラリをアンロードするための呼び出しは、対象のライブラリが現在使用されていない場合にのみ正常に完了します。プロシージャは、アンロードするライブラリの名前を指定する、1 つのオプションのパラメータ long varchar を取ります。パラメータが指定されていない場合、使用されていないすべての外部ライブラリがアンロードされます。

注意： ダイナミック・リンク・ライブラリを置き換える前に、実行中の Sybase IQ サーバから既存のライブラリをアンロードしてください。 ライブラリをアンロードしないと、サーバで問題が発生することがあります。ダイナミック・リンク・ライブラリを置き換える前に、Sybase IQ サーバを停止するか、**sa_external_library_unload** 関数を使用してライブラリをアンロードしてください。

Windows の場合、次のコマンドを使用して外部関数ライブラリをアンロードします。

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

UNIX の場合、次のコマンドを使用して外部関数ライブラリをアンロードします。

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

登録済み関数が /abc/def/library のような完全パスを使用する場合は、最初に関数の登録を解除します。

Windows では、次のように使用します。

```
call sa_external_library_unload('¥abc¥def¥library.dll')
```

UNIX では、次のように使用します。

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

注意： ライブラリ・パスは、ライブラリ・ロード・パスのディレクトリ内にライブラリがまだ存在しない場合にのみ、SQL 関数宣言内に必要です。

エラー・チェックと呼び出しトレーシングの制御

external_UDF_execution_mode オプションは、v3 および v4 の外部ユーザ定義関数を含む文の評価時に実行されるエラー・チェックと呼び出しトレーシングの数を制御します。

UDF の開発中、デバッグを支援するために、**external_UDF_execution_mode** を使用できます。

指定できる値
0、1、2

デフォルト値
0

スコープ
パブリック、一時的、またはユーザとして設定できます。

説明
デフォルトで 0 に設定されている場合、UDF を使用して文のパフォーマンスを最適化する方法で、外部 UDF が評価されます。

1 に設定されている場合、各 UDF とやり取りする情報を検証するために、外部 UDF が評価されます。 この設定はスカラ UDF と集合 UDF 向けです。

2 に設定されている場合、各 UDF とやり取りする情報を検証するためだけでなく、UDF により提供される関数のすべての呼び出しと、それらの関数からサーバへのすべてのコールバックのログを iqmsg ファイルに記録するために、外部 UDF が評価されます。 この設定は C または C++ のすべての外部 UDF 向けです。テーブル UDF と TPF に対してはメモリ・トレースが有効になっています。

Sybase IQ ログ・ファイルの表示

Sybase IQ には、包括的ロギング機能とトレース機能が用意されています。UDF のコードに問題がある場合には、それと同じかそれ以上のレベルの詳細なロギングを UDF が提供する必要があります。

データベースのログ・ファイルは、通常、データベース・ファイルおよび設定ファイルと同じ場所にあります。UNIX プラットフォームの場合は、データベース・インスタンスに基づく名前の 2 つのファイルがあり、1 つは拡張子 `.stderr`、もう 1 つは拡張子 `.stdout` を持ちます。Windows の場合は、`stderr` ファイルはデフォルトでは生成されません。

Windows で、`stdout` メッセージに加えて `stderr` メッセージを取得するには、次のように `stdout` と `stderr` をリダイレクトします。

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

Windows の出力メッセージは、UNIX プラットフォームで生成される出力メッセージとは少し異なります。

ユーザ定義関数に対する Microsoft Visual Studio デバッガの使用

Microsoft Visual Studio 2008 を使用する開発者は、Microsoft Visual Studio のデバッガでユーザ定義関数のコードを 1 ステップずつ実行できます。

1. 実行中のサーバにデバッガをアタッチします。

   ```
   devenv /debugexe "%IQDIR15%¥Bin64¥iqsrv15.exe"
   ```

2. [Debug] - [Attach to Process] に移動します。
3. サーバとデバッガを同時に起動するには、次のコマンドを使用します。

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

   プラットフォームごとにデバッガが用意されており、コマンド・ライン構文はそれぞれ独自です。Sybase IQ のソース・コードは必要ありません。Microsoft Visual Studio のデバッガは、いつユーザ定義関数のソースが実行され、設定したブレークポイントでブレークするかを認識します。 ユーザ定義関数からサーバに制御が返されると、マシン・コードだけが表示されます。

実行時の UDF の修正

Sybase IQ インストール環境の多くは、顧客が非常に高いレベルの可用性を要求するミッション・クリティカルな環境です。システム管理者は、Sybase IQ サーバに影響をほとんど (またはまったく) 与えないように UDF をインストールおよびアップグレードできる必要があります。

関連するライブラリ・ファイルが移動、上書き、または削除されている間は、アプリケーションが外部ライブラリにアクセスしないようにします。ライブラリは関連付けられた SQL 関数が呼び出されるたびに自動的にロードされるため、既存の UDF ライブラリに対してどのような種類の管理作業を行う場合も、次の手順に従うことが重要です。

1. UDF を呼び出すすべてのユーザが保留中のクエリを持たないことを確認します。
2. ユーザの実行パーミッションを取り消し、外部 UDF コード・モジュールを参照する SQL 関数およびストアド・プロシージャを削除します。
3. **call sa_external_library_unload** コマンドを使用して、ライブラリを Sybase IQ サーバからアンロードします (IQ サーバを停止しても、ライブラリが自動的にアンロードされます)。
4. 外部ライブラリ ファイルに対して、目的の管理作業 (コピー、移動、更新、削除) を行います。
5. ライブラリを移動した場合は、登録スクリプトの SQL 関数およびストアド・プロシージャ定義を編集して外部ライブラリの場所を反映させます。
6. 実行パーミッションをユーザに付与し、登録スクリプトを実行して、外部 UDF コード・モジュールを参照する SQL 関数およびストアド・プロシージャを再作成します。
7. 外部 UDF コードを参照する SQL 関数またはストアド・プロシージャを呼び出して、Sybase IQ サーバが外部ライブラリを動的にロードできることを確認します。

パーミッションの付与と取り消し

ユーザ定義関数のデフォルトの所有者は、その関数を作成したユーザですが、作成時にそのユーザが別の所有者を指定することもできます。

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
```

また、次の文を発行してパーミッションを取り消すことができます。

```
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。

ユーザ定義関数の削除

作成したユーザ定義関数は、明示的に削除しないかぎり、データベースに保持されます。 関数またはプロシージャの所有者か、DBA 権限のあるユーザだけが、関数またはプロシージャをデータベースから削除できます。

たとえば、スカラ関数または集合関数 *fullname* をデータベースから削除するには、次のように入力します。

```
DROP FUNCTION fullname
```

テーブル UDF または TPF *fullname* をデータベースから削除するには、次のように入力します。

```
DROP PROCEDURE fullname
```

スカラ UDF と集合 UDF

スカラ UDF と集合 UDF は、呼び出し元の環境に単一の値を返します。

注意： スカラ UDF と集合 UDF は、ライセンスが必要なオプションであり、使用するには IQ_UDF または IQ_IDA ライセンスが必要です。ライセンスをインストールすると、ユーザ定義関数を使用できるようになります。

Sybase IQ は、さまざまな設定でインストールできます。UDF は、この環境内に簡単にインストールできる必要があります。また、サポートされているすべての設定で実行できる必要があります。Sybase IQ インストーラにはデフォルトのインストール・ディレクトリが用意されていますが、別のインストール・ディレクトリをユーザが選択することもできます。UDF の開発者は、UDF ライブラリおよび関連する SQL 関数定義スクリプトをインストールするときに、同様の柔軟性を提供することを考慮する必要があります。

スカラ UDF と集合 UDF の制限

外部 C/C++ スカラ UDF と集合 UDF には、次の制限があります。

- さまざまなコンテキスト関数を受け取りながら、複数のユーザが同時に呼び出すことができるように、すべての UDF を記述する必要があります。
- UDF がグローバルまたは共有のデータ構造にアクセスする場合、UDF 定義によって、データへのアクセスに関する適切なロックを実装する必要があります。これには、すべての通常のコード・パスやすべてのエラー処理状況におけるロックの解放などが含まれます。
- C++ で UDF を実装すると、そのクラスの "new" 演算子が過負荷になる可能性がありますが、グローバルな "new" 演算子を過負荷にしないようにしてください。一部のプラットフォームでは、過負荷による影響が特定のライブラリ内の定義されたコードに限定されません。
- すべての集合 UDF とすべての決定的なスカラ UDF は、同じ入力値を受け取ると必ず同じ出力値を生成するように記述します。これに該当しないスカラ関数については、NONDETERMINISTIC と宣言して、正しくない応答の生成を回避する必要があります。
- ユーザは、DBA 権限を持っていなくても標準 SQL 関数を作成できます。ただし、外部ライブラリを呼び出す関数は、DBA パーミッションを持っていないと作成できません。作成しようとすると、`"You do not have permission to use the create function statement."` というエラー・メッセージが表示されます。

スカラ UDF または集合 UDF の作成

外部 C/C++ スカラ UDF と集合 UDF の作成と設定について説明します。

Interactive SQL を使用した UDF の作成方法については、『システム管理ガイド：第 2 巻』の「プロシージャとバッチの使用」を参照してください。

1. **CREATE FUNCTION** 文または **CREATE AGGREGATE FUNCTION** 文を使用して、サーバに対して UDF を宣言します。これらの文を記述してコマンドとして実行するか、Sybase Central の新機能ウィザードを使用して、適切な CREATE 文を作成します。

   CREATE FUNCTION 文の外部 C/C++ 形式には DBA または RESOURCE 権限が必要であるため、通常のユーザにはこの種類の UDF を宣言する権限がありません。
2. UDF ライブラリ識別関数を記述します。 (20 ページ)
3. UDF を、C または C++ 関数のセットとして定義します。「スカラ UDF の定義」 (46 ページ) または「集計 UDF の定義」 (63 ページ) を参照してください。
4. C/C++ で関数のエントリ・ポイントを実装します。
5. UDF 関数およびライブラリ識別関数をコンパイルします。 (24 ページ)
6. コンパイル済みファイルをダイナミック・リンク・ライブラリにリンクします。

最初に、SQL 文の UDF への参照により、必要に応じてダイナミック・リンク・ライブラリがリンクされます。次に、パターン呼び出し (94 ページ) が呼び出されます。

これらの高パフォーマンスの外部 C/C++ ユーザ定義関数は、サーバのプロセス領域への非サーバ・ライブラリ・コードのロードも行うため、関数の記述が不完全な場合や意図的に不正な場合、データの整合性やセキュリティ、およびサーバの堅牢性に関してリスクが発生する可能性があります。 これらのリスクを管理するために、Sybase IQ サーバごとに明示的にこの機能を有効または無効にできます (31 ページ)。

SQL Anywhere のダイアレクトを使用した UDF の Sybase Central での作成

Watcom-SQL および Transact-SQL は SQL Anywhere がサポートする SQL ダイアレクトであり、ユーザ定義関数の作成時に使用できます。

1. Sybase Central で、DBA または RESOURCE 権限のあるユーザとしてデータベースに接続します。
2. [表示] > [フォルダ] を選択します。
3. 左側のウィンドウ枠で、[プロシージャと関数] を右クリックして、[新規] > [関数] を選択します。
4. 関数の名前を入力し、関数を所有するユーザを選択します。
5. 関数の SQL 構文または言語を選択します。[次へ] をクリックします。
6. 関数の戻り値の種類を選択し、値のサイズ、単位、および位取りを指定します。
7. 戻り値の名前を入力して、[次へ] をクリックします。
8. 新しい関数の目的を説明するコメントを追加します。[完了] をクリックします。
9. 右側のウィンドウ枠で、[SQL] タブをクリックし、プロシージャ・コードを完了します。

スカラ UDF の宣言と定義

Sybase IQ では、SQRT 関数を使用できる場所で使用できる単純なスカラ・ユーザ定義関数 (UDF) をサポートしています。

これらのスカラ UDF には、決定的スカラ関数、つまり特定の引数のセットに対して常に同じ結果値を返す関数、および非決定的スカラ関数、つまり同じ引数が異なる結果を返す場合のある関数のいずれを使用することもできます。

注意： この章で説明するスカラ UDF の例は IQ サーバとともにインストールされ、`$IQDIR15/samples/udf` にある .cxx ファイルに記載されています。`$IQDIR15/lib64/libudfex` ダイナミック・リンク・ライブラリからも参照できます。

スカラ UDF の宣言

DBA、または DBA 権限を持つユーザのみがインプロセス外部 UDF を宣言できます。管理者がこのようなユーザ定義関数を有効または無効にできる、サーバ起動オプションもあります。

UDF コードを作成してコンパイルしたら、適切なライブラリ ファイルから UDF を呼び出す SQL 関数を作成して、入力データを UDF に送信します。

注意： Sybase Central でユーザ定義関数宣言を作成 (45 ページ) することもできます。

デフォルトでは、すべてのユーザ定義関数が、UDF の所有者のアクセス・パーミッションを使用します。

注意： UDF 関数を宣言するには、DBA 権限が必要です。

スカラ UDF を作成するための構文は、次のとおりです。

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

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

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

上記の構文の特性のデフォルトは、次のとおりです。

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

潜在的なセキュリティの問題を最小限に抑えるため、EXTERNAL NAME 句のライブラリ名部分には安全なディレクトリの完全修飾パス名を使用することをおすすめします。

SQL セキュリティ

INVOKER (関数を呼び出しているユーザ) または DEFINER (関数を所有しているユーザ) のどちらとして関数が実行されるかを定義します。デフォルトは DEFINER です。

SQL SECURITY INVOKER を指定すると、プロシージャを呼び出すユーザごとに注釈が必要になるため、より多くのメモリが使用されます。また、ユーザ名と INVOKER の両方で名前の決定が行われます。適切な所有者名で、すべてのオブジェクト名 (テーブル、プロシージャなど) を修飾します。

外部名

EXTERNAL NAME 句を使用する関数は、外部ライブラリにある関数への呼び出しのラッパです。**EXTERNAL NAME** を使用する関数は、**RETURNS** 句の後に他の句を持つことができません。ライブラリ名にはファイル拡張子が付く場合があります。

この拡張子は通常、Windows では .dll、UNIX では .so です。拡張子が付いていなければ、ソフトウェアがプラットフォーム固有のデフォルトのファイル拡張子をライブラリに付加します。

EXTERNAL NAME 句はテンポラリ関数ではサポートされていません。『SQL Anywhere サーバー - プログラミング』の「SQL Anywhere 外部呼び出しインターフェイス」を参照してください。

注意： 参照先は SQL Anywhere のマニュアルです。

サーバは、UDF ライブラリの場所を含むライブラリ・ロード・パスで起動できます。 UNIX 系 OS では、start_iq startup スクリプトの LD_LIBRARY_PATH を変更します。 すべての UNIX 系 OS で LD_LIBRARY_PATH が一般的に使用されますが、HP では SHLIB_PATH が、AIX では LIB_PATH が優先的に使用されます。

UNIX プラットフォームでは、LD_LIBRARY_PATH が使用されない場合に、外部名の指定に完全修飾名を含めることができます。 Windows プラットフォームでは、完全修飾名は使用できず、ライブラリ検索パスは PATH 環境変数で定義されます。

注意： スカラ・ユーザ定義関数とユーザ定義の集合関数は、更新可能なカーソルではサポートされません。

参照：

- Defining a Scalar UDF (46 ページ)

UDF Example: my_plus Declaration

"my_plus" の例は、2つの整数引数値を加算した結果を返す単純なスカラ関数です。

my_plus 宣言

ダイナミック・リンク・ライブラリ my_shared_lib 内に my_plus がある場合、この例の宣言は、次のようになります。

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

この宣言は、my_plus が describe_my_plus という名前の記述子ルーチンを持つ my_shared_lib にある単純なスカラ UDF であることを示します。UDF を実装するため、その動作には実際の C/C++ エントリ・ポイントが複数必要な場合があるので、このエントリ・ポイント・セットは CREATE FUNCTION 構文に直接的には含まれません。代わりに、CREATE FUNCTION 文の EXTERNAL NAME 句がこの UDF の記述子関数を指定します。記述子関数は呼び出されると、次の項で詳しく定義さ

れている記述子構造を返します。この記述子構造には、この UDF の実装を具体的に表す必須またはオプションの関数ポインタが含まれています。

この宣言は、my_plus が 2 つの INT 引数を受け入れ、1 つの INT 結果値を返すことを示します。関数が INT 以外の引数で呼び出され、引数を暗黙的に INT に変換できる場合は、関数が呼び出される前に変換が行われます。関数が暗黙的に INT に変換できない引数で呼び出された場合は、変換エラーが生成されます。

さらに、この宣言は、関数が決定的であることを示しています。一般的に、決定的関数は、同じ入力値が指定された場合、同じ結果値を返します。これは、結果が、指定された引数値以外の外部情報や以前の呼び出しに関連する動作に依存しないことを意味します。デフォルトでは、関数は決定的であると見なされ、この特性が CREATE 文から省略された場合も結果は同じになります。

上記の宣言の最後の部分は、IGNORE NULL VALUES 特性を示します。ほぼすべての組み込みスカラ関数は、入力引数のいずれかが NULL である場合、NULL 結果値を返します。IGNORE NULL VALUES は、my_plus 関数がこの規則に従うことを示します。このため、入力値のいずれかが NULL である場合、この UDF ルーチンは実際には呼び出されません。この関数のデフォルトは RESPECT NULL VALUES なので、パフォーマンスを高めるには、この特性をこの UDF の宣言で指定する必要があります。NULL 入力値を指定された場合に NULL 以外の結果を返す可能性のあるすべての関数は、デフォルトの RESPECT NULL VALUES 特性を使用する必要があります。

次のクエリの例では、SELECT リストに my_plus が同等の算術式とともに記述されています。

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

次の例では、my_plus が同じクエリ内のいくつかの場所で異なる方法で使用されています。

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

UDF Example: my_plus_counter Declaration

`my_plus_counter` は、1 つの整数引数を取り、内部整数使用カウンタに引数値を追加した結果を返す、単純な非決定的スカラ UDF の例です。 入力引数値が NULL である場合、結果は、使用カウンタの現在の値になります。

my_plus_counter 宣言

my_plus_counter もダイナミック・リンク・ライブラリ my_shared_lib 内にあると仮定すると、この例の宣言は次のようになります。

```
CREATE FUNCTION my_plus_counter (IN arg1 INT DEFAULT 0)
     RETURNS INT
     NOT DETERMINISTIC
     RESPECT NULL VALUES
     EXTERNAL NAME 'describe_my_plus_counter@my_shared_lib'
```

RESPECT NULL VALUES 特性は、入力引数値が NULL である場合にもこの関数が呼び出されることを意味します。この特性は、my_plus_counter のセマンティックに次のものが含まれるため必須です。

- 引数が NULL である場合も増加する使用カウントの内部保持。
- NULL 引数を渡した場合の NULL 以外の値の結果。

RESPECT NULL VALUES はデフォルトであるため、この句が宣言で省略されている場合でも、結果は同じです。

Sybase IQ では、すべての非決定的関数の使用を制限しています。非決定的関数は、最上位レベルのクエリ・ブロックの SELECT リスト、または UPDATE 文の SET 句でのみ使用できます。 サブクエリ内、WHERE 句、ON 句、GROUP BY 句、HAVING 句内では使用できません。 この制限は、GETUID、NUMBER などの非決定的な組み込み関数と同様に、非決定的な UDF に適用されます。

上記の宣言の最後の部分は、入力パラメータの DEFAULT 修飾子です。この修飾子はサーバに対して、この関数を引数なしで呼び出せることと、その場合は欠落した引数に対してサーバが自動的にゼロを指定することを通知します。DEFAULT 値が指定された場合、その引数のデータ型に暗黙的に変換する必要があります。

次の例では、最初の SELECT リスト項目によって、各ローの t.x の値に実行中のカウンタが追加されます。2 番目および 3 番目の SELECT リスト項目は、各ローの同じ値を NUMBER 関数として返します。

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

UDF の例：my_byte_length 宣言

my_byte_length は、カラムのサイズをバイト単位で返す単純なスカラ UDF です。

my_byte_length 宣言

ダイナミック・リンク・ライブラリ my_shared_lib 内に **my_byte_length** がある場合、この例の宣言は次のようになります。

```
CREATE FUNCTION my_byte_length(IN arg1 LONG BINARY)
//             RETURNS UNSIGNED INT
//             DETERMINISTIC
//           IGNORE NULL VALUES
//             EXTERNAL NAME 'my_byte_length@libudfex'
```

この宣言は、**my_byte_length** が describe_my_byte_length という名前の記述子ルーチンを持つ my_shared_lib にある単純なスカラ UDF であることを示します。 UDF の動作の実装には実際の C/C++ エントリ・ポイントが複数必要な場合があるので、このエントリ・ポイント・セットは **CREATE FUNCTION** 構文に直接的には含まれません。 代わりに、**CREATE FUNCTION** 文の **EXTERNAL NAME** 句でこの UDF の記述子関数を指定します。 記述子関数は、呼び出されると、記述子構造を返します。この記述子構造には、この UDF の実装を具体的に表す必須またはオプションの関数ポインタが含まれています。

この宣言は、**my_byte_length** が 1 つの `LONG BINARY` を受け入れ、`UNSIGNED INT` の結果値を返すことも示します。

注意： ラージ・オブジェクト・データのサポートには、別途ライセンスが必要な Sybase IQ オプションが必要です。

この宣言は、関数が決定的であることを示しています。決定的関数は、同じ入力値が指定された場合、常に同じ結果値を返します。 つまり、関数の結果は、指定された引数値以外の外部情報や以前の呼び出しには影響されません。 デフォルトでは、関数は決定的であると見なされます。したがって、この特性を **CREATE** 文から省略した場合も結果は同じになります。

この宣言の最後の部分は、IGNORE NULL VALUES 特性です。入力引数のいずれかが NULL の場合、ほぼすべての組み込みスカラ関数が NULL の結果値を返します。IGNORE NULL VALUES は、**my_byte_length** 関数がその規則に従うことを指定します。これにより、その入力値のいずれかが NULL のときでも、この UDF ルーチンが実際には呼び出されないようになります。RESPECT NULL VALUES は関数のデフォルト設定であるため、パフォーマンスを向上させるには、この UDF に対する宣言でこの特性を指定する必要があります。NULL の入力値から NULL 以外の結果を返す可能性のある関数の場合は常に、デフォルトの RESPECT NULL VALUES 特性を使用する必要があります。

SELECT リストに **my_byte_length** を指定したクエリの次の例は、`exTable` のローごとに 1 つのローを含むカラムを返します。これには、バイナリ・ファイルのサイズを示す `INT` も含まれます。

```
SELECT my_byte_length(exLOBColumn)
FROM exTable
```

Sybase Central におけるスカラ・ユーザ定義関数の宣言

Sybase IQ では、SQRT 関数を使用できる場所で使用できる単純なスカラ UDF をサポートしています。 これらのスカラ UDF は決定的関数にできます。これは、指定された引数の値のセットに対して、関数が常に同じ結果値を返すことを意味します。また、Sybase IQ は、非決定的なスカラ関数もサポートしています。これは、同じ引数が異なる結果を返すことができることを意味します。

1. Sybase Central で、DBA または RESOURCE 権限のあるユーザとしてデータベースに接続します。
2. 左側のウィンドウ枠で、[プロシージャと関数] を右クリックして、[新規] > [関数] を選択します。
3. [ようこそ] ダイアログで、関数の名前を入力し、関数の所有者になるユーザを選択します。
4. ユーザ定義関数を作成するには、[外部 C/C++] を選択します。[次へ] をクリックします。
5. [外部関数属性] ダイアログで、[スカラ] を選択します。
6. .so または .dll 拡張子を省略して、ダイナミック・リンク・ライブラリ・ファイルの名前を入力します。
7. 記述子関数の名前を入力します。[次へ] をクリックします。
8. 関数の戻り値の種類を選択し、値のサイズ、単位、および位取りを指定します。[次へ] をクリックします。
9. 関数が決定的であるかどうかを選択します。
10. 関数が NULL 値を尊重するか無視するかを指定します。
11. 関数の実行に使用される権限が、定義しているユーザ (定義者) の権限であるか、呼び出し側のユーザ (呼び出し者) の権限であるかを選択します。
12. 新しい関数の目的を説明するコメントを追加します。[完了] をクリックします。
13. 右側のウィンドウ枠で、[SQL] タブをクリックし、プロシージャ・コードを完了します。

Defining a Scalar UDF

スカラ・ユーザ定義関数を定義するための C/C++ コードには、次の 4 つの要素が必要です。

- **extfnapiv3.h** – UDF インタフェース定義ヘッダ・ファイルが含まれます。
- **_evaluate_extfn** – 評価関数。すべての評価関数は、2 つの引数を取ります。
  - スカラ UDF コンテキスト構造のインスタンス。コールバック関数ポインタと、UDF が UDF 固有データを格納するポインタのセットを含む UDF の各使用に対してユニークです。
  - データ構造へのポインタ。指定されたコールバックを介して引数値および結果値にアクセスできます。
- **a_v3_extfn_scalar** – スカラ UDF 記述子構造のインスタンス。評価関数へのポインタを含みます。
- **記述子関数** – スカラ UDF 記述子構造へのポインタを返します。

次の要素は省略可能です。

- **_start_extfn** – 通常、SQL の使用ごとに呼び出される初期化関数。指定した場合、この関数へのポインタも、スカラ UDF 記述子構造内に配置する必要があります。すべての初期化関数は、UDF の使用ごとにユニークなスカラ UDF コンテキスト構造へのポインタを 1 つの引数として取ります。渡されるコンテキスト構造は、評価ルーチンに渡されるものと同じです。
- **_finish_extfn** – 通常、SQL の使用ごとに呼び出されるシャットダウン関数。指定した場合、この関数へのポインタも、スカラ UDF 記述子構造内に配置する必要があります。すべてのシャットダウン関数は、UDF の使用ごとにユニークなスカラ UDF コンテキスト構造へのポインタを 1 つの引数として取ります。渡されるコンテキスト構造は、評価ルーチンに渡されるものと同じです。

参照：

- スカラ UDF の宣言 (39 ページ)

Scalar UDF Descriptor Structure

スカラ UDF 記述子構造 a_v3_extfn_scalar は次のように定義されます。

```
typedef struct a_v3_extfn_scalar {    //
    // Metadata descriptor for a scalar UDF
    // supplied by the UDF library to the server
    // An optional pointer to an initialize function
    void (*_start_extfn)(a_v3_extfn_scalar_context * cntxt);
    //
    // An optional pointer to a shutdown function
    void (*_finish_extfn)(a_v3_extfn_scalar_context *    cntxt);
    //
    // A required pointer to a function that will be
    // called for each invocation of the UDF on a
```

```
    // new set of argument values
    void (*_evaluate_extfn)(a_v3_extfn_scalar_context * cntxt, void
*args_handle);
    // RESERVED FIELDS MUST BE INITIALIZED TO NULL
 void  *reserved1_must_be_null;
 void  *reserved2_must_be_null;
 void  *reserved3_must_be_null;
 void  *reserved4_must_be_null;
 void  *reserved5_must_be_null;
                    …

} a_v3_extfn_scalar;
```

定義済みのスカラ UDF ごとに、**a_v3_extfn_scalar** のインスタンスが 1 つあります。オプションの初期化関数が指定されていない場合、記述子構造内の対応する値は NULL ポインタです。同様に、シャットダウン関数が指定されていない場合、記述子構造内の対応する値は NULL ポインタになります。

初期化関数は評価ルーチンへの呼び出しの前に少なくとも 1 回呼び出されます。シャットダウン関数は最後の評価呼び出しの後に少なくとも 1 回呼び出されます。通常、初期化関数とシャットダウン関数は、使用ごとに 1 回だけ呼び出されます。

Scalar UDF Context Structure

スカラ UDF 記述子構造内で指定された各関数に渡される、スカラ UDF コンテキスト構造 **a_v3_extfn_scalar_context** は、次のように定義されます。

```
typedef struct a_v3_extfn_scalar_context {
//-------- Callbacks available via the context --------
//
  short (SQL_CALLBACK *get_value)(
    void            *arg_handle,
             a_sql_uint32    arg_num,
             an_extfn_value *value
             );
 short (SQL_CALLBACK *get_piece)(
             void *          arg_handle,
             a_sql_uint32    arg_num,
             an_extfn_value *value,
             a_sql_uint32    offset
             );
 short (SQL_CALLBACK *get_value_is_constant)(
             void *          arg_handle,
             a_sql_uint32    arg_num,
             a_sql_uint32 *  value_is_constant
             );
 short (SQL_CALLBACK *set_value)(
             void *          arg_handle,
             an_extfn_value *value,
             short              append
             );
 a_sql_uint32 (SQL_CALLBACK *get_is_cancelled)(
             a_v3_extfn_scalar_context * cntxt
```

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

注意： **get_piece** コールバックは、v3 と v4 のスカラ UDF と集合 UDF で有効です。v4 のテーブル UDF と TPF については、BLOB (`a_v4_extfn_blob`) および BLOB 入力ストリーム (`a_v4_extfn_blob_istream`) の構造体を代わりに使用します。

スカラ UDF コンテキスト構造内の _user_data フィールドには、UDF が要求するデータを格納できます。 通常は、_start_extfn 関数により構造が割り付けられ、_finish_extfn 関数により割り付けが解除されるヒープが格納されます。

スカラ UDF コンテキスト構造の残りの部分には、ユーザの UDF 関数ごとに使用するためにエンジンによって指定される、コールバック関数のセットが格納されます。これらのコールバック関数のほとんどは、簡単な結果値で成功ステータスを返します。戻り値 true は成功を示します。UDF 実装が正しく記述されていれば、失敗ステータスを生じることはありませんが、開発中と、可能であれば指定された UDF ライブラリのすべてのデバッグ・ビルド中は、コールバックから返されるステータス値を確認することをおすすめします。失敗は、UDF が取得するように定義されているよりも多くの引数を要求するなど、UDF 実装のコーディング・エラーに起因する可能性があります。

ほとんどのコールバックで使用される引数の一般的なセットには、次のものが含まれます。

- **arg_handle** – すべての形式の評価メソッドが受け取るポインタ。このポインタにより、UDF に渡される入力引数の値が使用可能になり、UDF の結果値が設定されます。
- **arg_num** – どの入力引数にアクセスしているかを示す整数。入力引数は 1 から始まり、昇順で左から右に番号が付けられます。
- **cntxt** – サーバがすべての UDF エントリ・ポイントに渡す、コンテキスト構造へのポインタ。

- **value** – サーバからの入力引数値の取得または関数の結果値の設定に使用される、an_extfn_value 構造のインスタンスへのポインタ。an_extfn_value 構造の形式は、次のとおりです。

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

表 1 : スカラ外部関数コンテキスト： a_v3_extfn_scalar_context

a_v3_extfn_scalar_context 構造体のメソッド	説明
void set_cannot_be_distributed(a_v3_extfn_scalar_context * cntxt)	ライブラリ・レベルで分散の基準を満たす場合でも、UDF レベルで分散を無効にできます。 デフォルトでは、ライブラリが分散可能であれば UDF も分散可能と見なされます。 分散を無効にするという判断をサーバに通知することは UDF の役割です。

参照：

- BLOB (a_v4_extfn_blob) (219 ページ)
- BLOB 入力ストリーム (a_v4_extfn_blob_istream) (222 ページ)

例： my_plus 定義

スカラ UDF my_plus の定義の例を、次に示します。

my_plus 定義

この UDF には初期化関数やシャットダウン関数が必要ないので、記述子構造内のこれらの値は 0 に設定されます。記述子関数名は、宣言で使用される EXTERNAL NAME と一致します。この評価メソッドでは、引数が INT として宣言されているため、引数のデータ型をチェックしません。

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


//  A simple deterministic scalar UDF that just adds
//  two integer arguments and then returns the result.
//
//  Corresponding SQL declaration:
//
//    CREATE FUNCTION my_plus(IN arg1 INT, IN arg2 INT)
```

```
//                                              RETURNS INT
//                                              DETERMINISTIC
//                                    IGNORE NULL VALUES
//                                              EXTERNAL NAME
'my_plus@libudfex'
//
#if defined __cplusplus
extern "C" {
#endif

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

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

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

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


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


a_v3_extfn_scalar *my_plus()
```

```
{
  return &my_plus_descriptor;
}

#if defined __cplusplus
}
#endif
```

例：my_plus _counter 定義

このスカラ UDF の例では、入力引数値が NULL かどうかを確認するために、引数値ポインタ・データをチェックします。 また、初期化関数とシャットダウン関数も含まれ、それぞれが複数の呼び出しを受け入れます。

my_plus _counter 定義

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

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


typedef struct my_counter {
  a_sql_int32 _counter;
} my_counter;

#if defined __cplusplus
extern "C" {
#endif

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

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


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

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

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

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


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


a_v3_extfn_scalar *my_plus_counter()
{
  return &my_plus_counter_descriptor;
```

```
}

#if defined __cplusplus
}
#endif
```

例：my_byte_length 定義

スカラ UDF **my_byte_length** は、データを部分単位でストリームすることでカラムのサイズを測定し、そのカラムのサイズをバイト単位で返します。

my_byte_length definition

注意： ラージ・オブジェクト・データのサポートには、別途ライセンスが必要な Sybase IQ オプションが必要です。

```
#include "extfnapiv4.h"
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <assert.h>


//  A simple function that returns the size of a cell value in bytes
//
//    CREATE FUNCTION my_byte_length(IN arg1 LONG BINARY)
//            RETURNS UNSIGNED INT
//            DETERMINISTIC
//          IGNORE NULL VALUES
//            EXTERNAL NAME 'my_byte_length@libudfex'

#if defined __cplusplus
extern "C" {
#endif

static void my_byte_length_evaluate(a_v3_extfn_scalar_context
*cntxt,
                     void *arg_handle)
{

    if (cntxt == NULL || arg_handle == NULL)
    {
        return;
    }

    an_extfn_value  arg;
    an_extfn_value  outval;

    a_sql_uint64 total_len;

    //  Get first argument
    a_sql_uint32 fetchedLength = 0;
    (void) cntxt->get_value( arg_handle, 1, &arg );
    if (arg.data == NULL)
    {
```

```
        return;
    }

    fetchedLength += arg.piece_len;

    // saving total length as it loses scope inside get_piece
    total_len = arg.len.total_len;

    while (fetchedLength < total_len)
    {
       (void) cntxt->get_piece( arg_handle, 1, &arg, fetchedLength );
        fetchedLength += arg.piece_len;
    }

   //if this fails, the function did not get the full data from the
cell
    assert(fetchedLength == total_len);

    outval.type = DT_UNSINT;
    outval.piece_len = 4;
    outval.data = &fetchedLength;
    cntxt->set_value(arg_handle, &outval, 0);
}

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

a_v3_extfn_scalar *my_byte_length()
{
  return &my_byte_length_descriptor;
}

#if defined __cplusplus
}
#endif
```

参照：

- 例： my_byte_length 定義 (53 ページ)

集合 UDF の宣言と定義

Sybase IQ は集合 UDF をサポートしています。組み込みの集合関数の例としては SUM 関数があります。 単純な集合関数は、一連の引数値から 1 つの結果値を生成

します。 SUM 集合関数を使用できる場所であればどこでも使用できる集合 UDF を作成できます。

注意： ここで説明する集合 UDF の例はサーバとともにインストールされ、`$IQDIR15/samples/udf` にある .cxx ファイルに記述されています。`$IQDIR15/lib64/libudfex` ダイナミック・リンク・ライブラリからも参照できます。

集合関数は、単一の結果または複数の結果のセットを出力できます。出力結果セットのデータ・ポイントの数は、入力セットのデータ・ポイントの数と一致するとはかぎりません。複数出力の集合 UDF は、テンポラリ出力ファイルを使用して結果を保持します。

集合 UDF の宣言

集合 UDF は、スカラ UDF よりも機能が強力で、作成もより複雑です。

UDF コードを作成してコンパイルしたら、適切なライブラリ ファイルから UDF を呼び出す SQL 関数を作成して、入力データを UDF に送信します。

注意： Sybase Central でユーザ定義関数宣言を作成 (45 ページ) することもできます。

集合 UDF を実装するときには、次のことを決定します。

- RANK のように、データ・セット全体またはパーティションでのみオンライン分析処理 (OLAP) スタイルの集合として動作するかどうか。
- 1 つの集合として動作するか、または SUM のように OLAP スタイルの集合として動作するか。
- グループ全体で 1 つの集合としてのみ動作するかどうか。

集合 UDF の宣言と定義には、使用法における上記の決定が反映されます。

ユーザ定義の集合関数を作成する構文は、次のとおりです。

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

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

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

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

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

restrict:   { [ NOT ] ALLOWED } | REQUIRED

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

戻り値のデータ型、引数、データ型、およびデフォルト値の処理は、スカラ UDF 定義の場合と同じです。

集合 UDF を単純な集合として使用できる場合は、DISTINCT 修飾子とともに使用できる可能性があります。 集合 UDF の宣言の DUPLICATE 句では、次のことを決定します。

- 結果が重複に応じて変わるため、集合 UDF が呼び出される前に重複値が削除対象と見なされるかどうか (組み込みの "COUNT(DISTINCT T.A)" の場合など)。または
- 重複が存在しても結果が変わらないかどうか ("MAX(DISTINCT T.A)" の場合など)。

DUPLICATE INSENSITIVE オプションにより、オプティマイザは、結果に影響を与えずに重複の削除を検討でき、クエリの実行方法を選択できます。 集合 UDF は、重複の発生を前提に記述してください。重複の削除が必要な場合、_next_value_extfn 呼び出しのセットを開始する前に、サーバは削除を実行します。

スカラ UDF 構文の一部ではない、残りの句の大部分では、この関数の使用法を指定できます。 デフォルトでは、集合 UDF は、任意のウィンドウ・フレームで単純な集合としても OLAP スタイルの集合としても使用できるものと見なされます。

集合 UDF を単純な集合関数としてのみ使用するには、次のように宣言します。

```
OVER NOT ALLOWED
```

この集合を OLAP スタイルの集合として使用しようとすると、エラーが生成されます。

集合 UDF で OVER 句を使用できる、または必須である場合、UDF 定義者は、"ORDER" とそれに続く制限タイプを指定することで、OVER 句内の ORDER BY 句の存在に対して制限を指定できます。 ウィンドウ順序付けの制限タイプは、次のとおりです。

- **REQUIRED** – ORDER BY の指定は必須であり、削除できません。
- **SENSITIVE** – ORDER BY を指定するかどうかは選択できますが、指定した場合は削除できません。
- **INSENSITIVE** – ORDER BY を指定するかどうかは選択できますが、サーバは順序付けを削除して効率を高めることができます。
- **NOT ALLOWED** – ORDER BY を指定できません。

組み込みの RANK のように、順序付けされたセット全体またはパーティションに対して、OLAP スタイルの集合としてのみ集合 UDF を宣言するには、次のような構文を使用します。

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

UNBOUNDED PRECEDING のデフォルトのウィンドウ・フレームを CURRENT ROW に対して使用し、OLAP スタイルの集合としてのみ集合 UDF を宣言するには、次のような構文を使用します。

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

すべてのさまざまなオプションおよび制限のセットのデフォルトは、次のとおりです。

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

SQL セキュリティ

INVOKER (関数を呼び出しているユーザ) または DEFINER (関数を所有しているユーザ) のどちらとして関数が実行されるかを定義します。デフォルトは DEFINER です。

SQL SECURITY INVOKER が指定された場合、プロシージャを呼び出すユーザごとに注釈が必要であるため、より多くのメモリが使用されます。また、**SQL SECURITY INVOKER** が指定された場合、ユーザ名と INVOKER の両方で名前の決定

が行われます。適切な所有者名で、すべてのオブジェクト名 (テーブル、プロシージャなど) を修飾します。

外部名

EXTERNAL NAME 句を使用する関数は、外部ライブラリにある関数への呼び出しのラッパです。**EXTERNAL NAME** を使用する関数は、**RETURNS** 句の後に他の句を持つことができません。ライブラリ名にはファイル拡張子が付く場合があります。この拡張子は通常、Windows では `.dll`、UNIX では `.so` です。拡張子が付いていなければ、ソフトウェアがプラットフォーム固有のデフォルトのファイル拡張子をライブラリに付加します。

EXTERNAL NAME 句はテンポラリ関数ではサポートされていません。『SQL Anywhere サーバー - プログラミング』の「SQL Anywhere 外部呼び出しインターフェイス」を参照してください。

注意： 参照先は SQL Anywhere のマニュアルです。

サーバは、UDF ライブラリの場所を含むライブラリ・ロード・パスで起動できます。 UNIX 系 OS では、start_iq 起動スクリプトの内 LD_LIBRARY_PATH を変更することで実行できます。 すべての UNIX 系 OS で LD_LIBRARY_PATH が一般的に使用されますが、HP では SHLIB_PATH が、AIX では LIB_PATH が優先的に使用されます。

UNIX プラットフォームでは、LD_LIBRARY_PATH が使用されない場合に、外部名の指定に完全修飾名を含めることができます。 Windows プラットフォームでは、完全修飾名は使用できず、ライブラリ検索パスは PATH 環境変数で定義されます。

注意： スカラ・ユーザ定義関数とユーザ定義の集合関数は、更新可能なカーソルではサポートされません。

参照：
- 集合 UDF の定義 (63 ページ)
- 集合 UDF のコンテキスト記憶領域 (92 ページ)

例：my_sum 宣言

"my_sum" の例は組み込みの SUM に似ていますが、整数でのみ動作する点が異なります。

my_sum 宣言

SUM と同様に my_sum は任意のコンテキストで使用できるので、その宣言は比較的簡潔です。

```
CREATE AGGREGATE FUNCTION my_sum(IN arg1 INT)
    RETURNS BIGINT
```

```
    ON EMPTY INPUT RETURNS NULL
    EXTERNAL NAME 'describe_my_integer_sum@my_shared_lib'
```

さまざまな使用上の制限はすべてデフォルトで ALLOWED で、これにより、集合関数が許容される SQL 文のどの場所でもこの関数を使用できます。

使用上の制限がない場合は、次に示すように、my_sum は、ローのセット全体で単純な集合として使用できます。

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

使用上の制限がない場合、my_sum は、GROUP BY 句の指定に従って、グループごとに計算される単純な集合としても使用できます。

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

使用上の制限がないため、my_sum は、次の累積合計の例が示すように、OVER 句とともに OLAP スタイルの集合として使用できます。

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

例：my_bit_xor 宣言

"my_bit_xor" の例は SQL Anywhere (SA) の組み込みの BIT_XOR に似ていますが、符号なし整数でのみ動作する点が異なります。

my_bit_xor 宣言

結果の宣言は次のようになります。

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

my_sum の例と同様に、my_bit_xor には関連する使用上の制限がないため、任意のウィンドウで単純な集合としても、OLAP スタイルの集合としても使用できます。

例： my_bit_or 宣言

"my_bit_or" の例は、SA の組み込みの BIT_OR に似ていますが、符号なし整数でのみ動作する点が異なり、単純な集合としてのみ使用できます。

my_bit_or 宣言

結果の宣言は次のようになります。

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

my_bit_xor の例とは異なり、宣言の OVER NOT ALLOWED フレーズは、この関数を単純な集合として使用するように制限しています。この使用上の制限により、my_bit_or は、ローのセット全体で単純な集合として、または、次の例に示すように GROUP BY 句の指定に従ってグループごとに計算される単純な集合としてのみ使用できます。

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

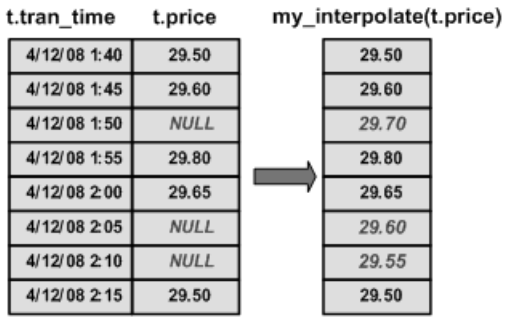
例： my_interpolate 宣言

"my_interpolate" の例は、隣接する NULL 値のセットに対して、各方向で最も近い NULL 以外の値への線形補間を実行することにより、シーケンス内の NULL で示された欠落値に入力しようとする OLAP スタイルの UDAF です。

my_interpolate 宣言

特定のローの入力が NULL でない場合、そのローの結果は入力値と同じです。

図 1 : my_interpolate の結果



相応なコストで動作するには、固定幅のローベースのウィンドウを使用して my_interpolate を実行する必要がありますが、ユーザは、隣接する NULL 値の予想される最大数に基づいて、ウィンドウの幅を設定できます。この関数は、倍精度浮動小数点数値のセットを取得し、double 型の結果セットを生成します。

結果の UDAF 宣言は、次のようになります。

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

OVER REQUIRED は、この関数が単純な集合として使用できないことを意味します (使用された場合、ON EMPTY INPUT は関係ありません)。

WINDOW FRAME の部分は、この関数を使用したときに現在のローから前後に拡張する固定幅のローベースのウィンドウを使用する必要があることを指定します。これらの使用上の制限により、my_interpolate は、次のように、OVER 句とともに OLAP スタイルの集合として使用できます。

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

my_interpolate の OVER 句内では、ローの前後の正確な数はさまざまであり、オプションで PARTITION BY 句を使用できます。それ以外の場合、宣言で使用上の制限が指定されているとき、ローは、上記の例と同様である必要があります。

Sybase Central での集合 UDF の宣言

Sybase IQ は集合 UDF をサポートしています。組み込みの集合関数の例としては SUM 関数があります。単純な集合関数は、一連の引数値を受け取り、その入力値のセットから 1 つの結果値を生成します。 SUM 集合関数を使用できる場所であればどこでも使用できる集合 UDF を作成できます。

1. Sybase Central で、DBA または RESOURCE 権限のあるユーザとしてデータベースに接続します。

2. 左側のウィンドウ枠で、[プロシージャと関数] を右クリックして、[新規] > [関数] を選択します。
3. [ようこそ] ダイアログで、関数の名前を入力し、関数の所有者になるユーザを選択します。
4. ユーザ定義関数を作成するには、[外部 C/C++] を選択します。[次へ] をクリックします。
5. [集計] を選択します。
6. .so または .dll 拡張子を省略して、ダイナミック・リンク・ライブラリ・ファイルの名前を入力します。
7. 記述子関数の名前を入力します。[次へ] をクリックします。
8. 関数の戻り値の種類を選択し、値のサイズ、単位、および位取りを指定します。[次へ] をクリックします。
9. 関数の実行に使用される権限が、定義しているユーザ (定義者) の権限であるか、呼び出し側のユーザ (呼び出し者) の権限であるかを選択します。
10. 関数を **OVER** 句で使用できるようにするか、使用する必要があるか、または使用できないようにするかを指定します。[次へ] をクリックします。

    関数を **OVER** 句で使用できないようにする場合は、手順 14 に進みます。
11. ウィンドウの定義に使用する場合に関数が **ORDER BY** 句のユーザを必要とするかどうかを指定します。[次へ] をクリックします。
12. 関数を **WINDOW FRAME** 句で使用できるようにするか、**WINDOW FRAME** 句で使用する必要があるか、または **WINDOW FRAME** 句で使用できないようにするかを指定します。[次へ] をクリックします。

    関数を **WINDOW FRAME** 句で使用できないようにする場合は、手順 14 に進みます。
13. **WINDOW FRAME** 句に対する制限を指定します。[次へ] をクリックします。
14. 関数を呼び出す前に、重複する入力値をデータベース・サーバでフィルタする必要があるかどうかを指定します。
15. データを持たない関数が呼び出されたときに、関数の戻り値を **NULL** にするか、固定値にするかを指定します。[次へ] をクリックします。
16. 新しい関数の目的を説明するコメントを追加します。[完了] をクリックします。
17. 右側のウィンドウ枠で、[SQL] タブをクリックし、プロシージャ・コードを完了します。

    [プロシージャと関数] に新しい関数が表示されます。

集合 UDF の定義

ユーザ定義の集合関数を定義するための C/C++ コードには、8 つの要素が必要です。

- **extfnapiv3.h** – UDF インタフェース定義ヘッダ・ファイル。 v4 API 用のファイルは `extfnapiv4.h` です。
- **_start_extfn** – SQL の使用ごとに呼び出される初期化関数。 すべての初期化関数は 1 つの引数を取ります。 集合 UDF の使用ごとにユニークな、集合 UDF コンテキスト構造へのポインタです。 渡されるコンテキスト構造は、その使用で指定されるすべての関数に渡されるものと同じです。
- **_finish_extfn** – SQL の使用ごとに呼び出されるシャットダウン関数。 すべてのシャットダウン関数は 1 つの引数を取ります。 集合 UDF の使用ごとにユニークな、集合 UDF コンテキスト構造へのポインタです。
- **_reset_extfn** – 新しいグループ、新しいパーティションの開始ごとに呼び出されるリセット関数。必要に応じて、各ウィンドウ動作の開始時にも呼び出されます。 すべてのリセット関数は 1 つの引数を取ります。 集合 UDF の使用ごとにユニークな、集合 UDF コンテキスト構造へのポインタです。
- **_next_value_extfn** – 新しい入力引数セットごとに呼び出される関数。
  _next_value_extfn は次の 2 つの引数を取ります。
  - 集合 UDF コンテキストへのポインタ。
  - args_handle。

  スカラ UDF の場合と同様に、arg_handle は、実際の引数値にアクセスするために指定されたコールバック関数ポインタとともに使用されます。
- **_evaluate_extfn** – スカラ UDF 評価関数と同様の評価関数。すべての評価関数は、2 つの引数を取ります。
  - 集合 UDF コンテキスト構造へのポインタ。
  - args_handle。
- **a_v3_extfn_aggregate** – 集合 UDF 記述子構造のインスタンス。この UDF について指定されたすべての関数へのポインタを含みます。
- **記述子関数** – 集合 UDF 記述子構造へのポインタを返す記述子関数。

必須要素に加え、特定の使用状況のために最適化されたアクセスを可能にする、いくつかのオプションの要素があります。

- **_drop_value_extfn** – オプションの関数ポインタ。移動ウィンドウ・フレームからあふれた引数値の入力セットごとに呼び出されます。この関数は、集合の結果を設定しません。入力引数値にアクセスするには、get_value コールバック関数を使用します。また、必要に応じて、get_piece コールバック関数を繰り返し呼び出します。
  次の場合は、関数ポインタを NULL ポインタに設定します。

- この集合をウィンドウ・フレームで使用できない場合。
- 方法によっては集合に可逆性がない場合。
- パフォーマンスの最適化が重要でない場合。

_drop_value_extfn が指定されておらず、ユーザが移動ウィンドウを指定した場合、ウィンドウ・フレームが移動するたびにリセット関数が呼び出され、next_value 関数の呼び出しによりウィンドウ内の各ローが挿入され、最後に評価関数が呼び出されます。

_drop_value_extfn が指定された場合、ウィンドウ・フレームが移動するたびに、ウィンドウ・フレームからあふれたローごとにこの drop_value 関数が呼び出され、ウィンドウ・フレームに追加されたローごとに next_value 関数が呼び出され、最後に、評価関数が呼び出されて集合の結果が生成されます。

- **_evaluate_cumulative_extfn** – 引数値の新しい入力セットごとに呼び出すことのできる、オプションの関数ポインタ。この関数が指定され、UNBOUNDED PRECEDING から CURRENT ROW にまたがるローベースのウィンドウ・フレームで使用される場合、next_value 関数の呼び出しの代わりにこの関数が呼び出され、直後に評価関数が呼び出されます。

  _evalutate_cumulative_extfn は、set_value コールバックを介して、集合の結果を設定します。入力引数値のセットへのアクセスは、通常の get_value コールバック関数を介して行われます。次の場合は、この関数ポインタを NULL ポインタに設定します。
  - この集合がこの方法で使用されない場合。
  - パフォーマンスの最適化が重要でない場合。
- **_next_subaggregate_extfn** – 並列実行によりこの集合の一部の使用を最適化するために、_evaluate_superaggregate_extfn とともに使用する、オプションのコールバック関数ポインタ。

  一部の集合は、単純な集合として使用される場合 (すなわち、OVER 句を伴う OLAP スタイルの集合として使用されない場合)、最初の中間集合結果セットの生成により、パーティションに分割できます。この場合、中間結果はそれぞれ、入力ローの分離サブセットから計算されます。

  このような分割可能な集合の例を次に示します。
  - SUM。最終 SUM は、入力ローの各分離サブセットの SUM を実行し、サブ SUM に対して SUM を実行することにより計算されます。
  - COUNT(*)。最終 COUNT は、入力ローの各分離サブセットの COUNT を実行し、各パーティションの COUNT に対して SUM を実行することにより計算されます。

  集合が上記の条件を満たす場合、サーバは、集合の並列計算を実行できます。集合 UDF については、_next_subaggregate_extfn 関数ポインタと _evaluate_superaggregate_extfn ポインタの両方が指定されている場合にのみ、この並列最適化を適用できます。

_reset_extfn 関数は集合の最終結果を設定しません。また、この関数は、定義により、集合 UDF の定義済みの戻り値と同じデータ型である入力引数値を 1 つだけ持ちます。
サブ集合入力値へのアクセスは、通常の get_value コールバック関数を介して行われます。サブ集合とスーパー集合との間の直接的なやり取りはできません。このようなやり取りはすべてサーバが処理します。サブ集合とスーパー集合は、コンテキスト構造を共有しません。代わりに、個々のサブ集合は、非分割集合と同様に扱われます。独立したスーパー集合は、次のようなパターン呼び出しを認識します。

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

次のようなパターンも認識します。

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

_evaluate_superaggregate_extfn および _next_subaggregate_extfn のどちらも指定されていない場合、集合 UDF は制限され、GROUP BY CUBE または GROUP BY ROLLUP を含むクエリ・ブロック内の単純な集合としては許容されません。

- **_evaluate_superaggregate_extfn** – 並列化により単純な集合の一部の使用を最適化するために、_next_subaggregate_extfn とともに使用する、オプションのコールバック関数ポインタ。_evaluate_superaggregate_extfn は、分割集合の結果を返すために呼び出されます。結果値は、通常の set_value コールバック関数を使用して、a_v3_extfn_aggregate_context 構造からサーバに送信されます。

参照：

- 集合 UDF の宣言 (55 ページ)
- 集合 UDF のコンテキスト記憶領域 (92 ページ)
- BLOB (a_v4_extfn_blob) (219 ページ)
- BLOB 入力ストリーム (a_v4_extfn_blob_istream) (222 ページ)

集合 UDF 記述子構造

集合 UDF 記述子構造は、次の要素で構成されています。

- **typedef struct a_v3_extfn_aggregate** – ライブラリによって指定される集合 UDF 関数のメタデータ記述子。
- **_start_extfn** – 引数のみが a_v3_extfn_aggregate_context へのポインタである、初期化関数への必須ポインタ。通常は、一部の構造の割り付けと、a_v3_extfn_aggregate_context 内の _user_data フィールドへのアドレスの格納に使用されます。_start_extfn は、a_v3_extfn_aggregate_context ごとに 1 回だけ呼び出されます。

  ```
  void (*_start_extfn)(a_v3_extfn_aggregate_context *);
  ```
- **_finish_extfn** – 引数のみが a_v3_extfn_aggregate_context へのポインタである、シャットダウン関数への必須ポインタ。通常は、a_v3_extfn_aggregate_context 内の _user_data フィールドにアドレスが格納された、一部の構造の割り付け解除に使用されます。_finish_extfn は、a_v3_extfn_aggregate_context ごとに 1 回だけ呼び出されます。

  ```
  void (*_finish_extfn)(a_v3_extfn_aggregate_context *);
  ```
- **_reset_extfn** – 引数のみが a_v3_extfn_aggregate_context へのポインタである、start-of-new-group 関数への必須ポインタ。通常は、a_v3_extfn_aggregate_context 内の _user_data フィールドにアドレスが格納された構造の一部の値をリセットするために使用されます。_reset_extfn は繰り返し呼び出されます。

  ```
  void (*_reset_extfn)(a_v3_extfn_aggregate_context *);
  ```
- **_next_value_extfn** – 引数値の新しい入力セットごとに呼び出される、必須の関数ポインタ。この関数は、集合の結果を設定しません。入力引数値にアクセスするには、get_value コールバック関数を使用します。また、必要に応じて、get_piece コールバック関数を繰り返し呼び出します。これは、piece_len が total_len より小さい場合に必要です。

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

  注意： get_piece コールバックは、v3 と v4 のスカラ UDF と集合 UDF で有効です。v4 のテーブル UDF と TPF については、BLOB (`a_v4_extfn_blob`) および BLOB 入力ストリーム (`a_v4_extfn_blob_istream`) の構造体を代わりに使用します。
- **_evaluate_extfn** – 結果の集合結果値を返すために呼び出される、必須の関数ポインタ。_evaluate_extfn は、set_value コールバック関数を使用してサーバに送信されます。

  ```
  void (*_evaluate_extfn)(a_v3_extfn_aggregate_context *cntxt, void
  *args_handle);
  ```
- **_drop_value_extfn**– 移動ウィンドウ・フレームからあふれた引数値の入力セットごとに呼び出される、オプションの関数ポインタ。この関数を使用して集合

の結果を設定しないでください。入力引数値にアクセスするには、get_value コールバック関数を使用します。また、必要に応じて、get_piece コールバック関数を繰り返し呼び出します。ただし、アクセスは、piece_len が total_len より小さい場合に必要です。次の場合は、_drop_value_extfn を NULL ポインタに設定します。

- 集合をウィンドウ・フレームで使用できない場合。
- 方法によっては集合に可逆性がない場合。
- パフォーマンスの最適化が重要でない場合。

注意： get_piece コールバックは、v3 と v4 のスカラ UDF と集合 UDF で有効です。v4 のテーブル UDF と TPF については、BLOB (`a_v4_extfn_blob`) および BLOB 入力ストリーム (`a_v4_extfn_blob_istream`) の構造体を代わりに使用します。

この関数が指定されておらず、ユーザが移動ウィンドウを指定した場合、ウィンドウ・フレームが移動するたびにリセット関数が呼び出され、next_value 関数の呼び出しによりウィンドウ内の各ローが挿入されます。最後に評価関数が呼び出されます。

一方、この関数が指定された場合、ウィンドウ・フレームが移動するたびに、ウィンドウ・フレームからあふれたローごとにこの drop_value 関数が呼び出され、ウィンドウ・フレームに追加されたローごとに next_value 関数が呼び出されます。最後に、評価関数が呼び出されて集合結果が生成されます。

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

- **_evaluate_cumulative_extfn** – 引数値の新しい入力セットごとに呼び出される、オプションの関数ポインタ。この関数が指定され、UNBOUNDED PRECEDING から CURRENT ROW にまたがるローベースのウィンドウ・フレームで使用される場合、next_value の代わりにこの関数が呼び出され、直後に評価関数が呼び出されます。_evaluate_cumulative_extfn は、set_value コールバックを介して、集合の結果を設定します。入力引数値にアクセスするには、get_value コールバック関数を使用します。また、必要に応じて、get_piece コールバック関数を繰り返し呼び出します。これは、piece_len が total_len より小さい場合に必要です。

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

注意： get_piece コールバックは、v3 と v4 のスカラ UDF と集合 UDF で有効です。v4 のテーブル UDF と TPF については、BLOB (`a_v4_extfn_blob`) および BLOB 入力ストリーム (`a_v4_extfn_blob_istream`) の構造体を代わりに使用します。

- **_next_subaggregate_extfn** – 並列および部分的な結果集合により集合の一部の使用を最適化するために、_evaluate_superaggregate_extfn 関数とともに使用する

(使用法によっては、_drop_subaggregate_extfn 関数も使用)、オプションのコールバック関数ポインタ。
一部の集合は、単純な集合として使用された場合 (すなわち、OVER 句を伴う OLAP スタイルの集合として使用されない場合)、最初の中間集合結果のセットの生成により、パーティションに分割できます。この場合、中間結果はそれぞれ、入力ローの分離サブセットから計算されます。このような分割可能な集合の例を次に示します。

- SUM。最終 SUM は、入力ローの各分離サブセットの SUM を実行し、サブ SUM に対して SUM を実行することにより計算されます。
- COUNT(*)。最終 COUNT は、入力ローの各分離サブセットの COUNT を実行し、各パーティションの COUNT に対して SUM を実行することにより計算されます。

集合が上記の条件を満たす場合、サーバは、集合の並列計算を実行できます。集合 UDF については、_next_subaggregate_extfn コールバックと _evaluate_superaggregate_extfn コールバックの両方が指定されている場合にのみ、この最適化を適用できます。この使用パターンには、_drop_subaggregate_extfn は不要です。
同様に、RANGE ベースの OVER 句とともに集合を使用できるとき、_next_subaggregate_extfn、_drop_subaggregate_extfn、および _evaluate_superaggregate_extfn 関数がすべて集合 UDF 実装で指定されている場合にのみ、最適化を適用できます。
_next_subaggregate_extfn は集合の最終結果を設定しません。また、この関数は、定義により、集合 UDF の戻り値と同じデータ型である入力引数値を 1 つだけ持ちます。サブ集合入力値にアクセスするには、get_value コールバック関数を使用します。また、必要に応じて、get_piece コールバック関数を繰り返し呼び出します。これは、piece_len が total_len より小さい場合に必要です。

注意： get_piece コールバックは、v3 と v4 のスカラ UDF と集合 UDF で有効です。v4 のテーブル UDF と TPF については、BLOB (`a_v4_extfn_blob`) および BLOB 入力ストリーム (`a_v4_extfn_blob_istream`) の構造体を代わりに使用します。

サブ集合とスーパー集合との間の直接的なやり取りはできません。このようなやり取りはすべてサーバが処理します。サブ集合とスーパー集合は、コンテキスト構造を共有しません。個々のサブ集合は、非分割集合と同様に扱われます。独立したスーパー集合は、次のようなパターン呼び出しを認識します。

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

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

- **_drop_subaggregate_extfn** – 部分的な集合により、RANGE ベースの OVER 句を含む一部の使用を最適化するために、_next_subaggregate_extfn や _evaluate_superaggregate_extfn とともに使用する、オプションのコールバック関数ポインタ。_drop_subaggregate_extfn は、共通の順序付けキー値を共有するロー・セットが移動ウィンドウからまとめてあふれたときに必ず呼び出されます。この最適化は、3 つの関数がすべて UDF で指定されている場合にのみ適用されます。

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

- **_evaluate_superaggregate_extfn** – 並列実行により一部の使用を最適化するために、_next_subaggregate_extfn とともに使用する (場合によっては、_drop_subaggregate_extfn も使用)、オプションのコールバック関数ポインタ。前述したように、_evaluate_superaggregate_extfn は、分割集合の結果を返すときに呼び出されます。結果値は、set_value コールバック関数を使用して、a_v3_extfn_aggregate_contex 構造からサーバに送信されます。

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

- **NULL フィールド** – 次のフィールドを NULL に初期化します。

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

- **ステータス・インジケータ・ビット・フィールド** – 集合の処理に使用するアルゴリズムをエンジンが最適化するためのインジケータを含む、1 ビットのフィールド。

```
a_sql_uint32  indicators;
```

- **_calculation_context_size** – サーバが UDF の計算コンテキストごとに割り付けるバイト数。サーバは、クエリ処理時に複数の計算コンテキストを割り付けることができます。現在アクティブなグループ・コンテキストは、a_v3_extfn_aggregate_context_user_calculation_context で使用できます。

```
short _calculation_context_size;
```

- **_calculation_context_alignment** – ユーザの計算コンテキストの配置要件を指定します。有効な値は、1、2、4、または 8 です。

```
short _calculation_context_alignment;
```

- **必要な外部領域** – 次のフィールドにより、オプティマイザは、外部割り付けメモリのコストを検討できます。これらの値を使用し、オプティマイザは、複数の同時計算をどの程度実行できるかを検討できます。これらのカウンタの見積もりは、通常のローまたはグループに基づいて行い、最大値にならないようにします。UDF によりメモリが割り付けられていない場合は、これらのフィールドをゼロに設定します。

- external_bytes_per_group – 各集合の開始時に 1 つのグループに割り付けられるメモリの容量。通常、reset() の呼び出し時に割り付けられるメモリです。
- external_bytes_per_row – グループの各ローについて、UDF により割り付けられるメモリの容量。通常、これは next_value() 時に割り付けられるメモリの容量です。

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

- **将来の使用のための予約フィールド** – 次のフィールドを初期化します。

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

- **終わりの構文** – 次の構文により、記述子を完了します。

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

参照：

- BLOB (a_v4_extfn_blob) (219 ページ)
- BLOB 入力ストリーム (a_v4_extfn_blob_istream) (222 ページ)

計算コンテキスト

_user_calculation_context フィールドにより、サーバは、複数のデータ・グループについての計算を同時に実行できます。

集合 UDF は、ローを処理する際、計算のために中間カウンタを保持する必要があります。 これらのカウンタを管理するための単純なモデルでは、API 関数の開始時にメモリを確保し、集合コンテキストの _user_data フィールドにそのポインタを格納し、集合が API を完了するときにメモリを解放します。
_user_calculation_context フィールドに基づく別のメソッドを使用すると、サーバは、複数のデータ・グループについての計算を同時に実行できます。

_user_calculation_context フィールドは、同時処理グループごとにサーバが作成する、サーバ割り付けメモリ・ポインタです。サーバは、_user_calculation_context が現在処理中のロー・グループの正しい計算コンテキストを常に示すようにします。各 UDF API 呼び出しの間は、データに応じて、サーバは新しい _user_calculation_context 値を割り付けることができます。クエリ処理時、サーバは、計算コンテキスト領域をディスクに保存したりリストアしたりすることができます。

UDF は、すべての中間計算値をこのフィールドに格納します。通常の使用法を次に示します。

```
struct my_average_context
{
```

```
        int    sum;
        int    count;
};

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

    next_value(a_v3_aggregate_context *context, void *args_handle)
{
    mycontext  = (my_average_context *) context-
>_user_calculation_context;
    mycontext->count++;
    ..
}
```

このモデルでは、_user_data フィールドを使用できますが、中間結果計算に関する値を格納することはできません。開始および終了エントリ・ポイントの両方で、_user_calculation_context は NULL です。

_user_calculation_context を使用して同時処理を有効にするには、UDF は、計算コンテキスト用のサイズおよび配置要件を指定し、値を保持して a_v3_extfn_aggregate と _calculation_context_size を構造の sizeof() に設定するための構造を定義する必要があります。

また、UDF は、_calculation_context_alignment を介して _user_calculation_context のデータ配置要件も指定する必要があります。user_calculation_context メモリが文字バイト配列のみを含んでいる場合、特別な配置は不要であり、配置 1 を指定できます。同様に倍精度浮動小数点値には 8 バイト配置が必要な場合があります。配置要件は、プラットフォームとデータ型により異なります。必要なサイズより大きい配置を指定できますが、最小の配置を使用することでメモリの使用効率が向上します。

集合 UDF コンテキスト構造

集合 UDF コンテキスト構造 a_v3_extfn_aggregate_context には、スカラ UDF コンテキスト構造と同じコールバック関数ポインタ・セットがあります。

また、スカラ UDF コンテキストとよく似た読み込み／書き込み _user_data ポインタと、現在の使用および場所を示す、読み込み専用のデータ・フィールド・セットもあります。文中の UDF のユニークな各インスタンスには、呼び出されたときに、集合 UDF 記述子構造で指定された各関数に渡される、1 つの集合 UDF コンテキスト・インスタンスがあります。集合コンテキスト構造は、次のように定義されます。

- **typedef struct a_v3_extfn_aggregate_context** – クエリ内で参照される外部関数のインスタンスごとに作成されます。クエリ内の並列サブツリーで使用する場合は、並列サブツリーのコンテキストが個別にあります。
- **コンテキストから利用できるコールバック** – コールバック・ルーチンの一般的な引数は、次のとおりです。
  - **arg_handle** – サーバが提供する、関数インスタンスおよび引数へのハンドル。
  - **arg_num** – 引数値。戻り値は、0..N です。
  - **data** – 引数データへのポインタ。

  コンテキストは get_piece の前に get_value を呼び出す必要がありますが、piece_len が total_len より小さい場合のみ、get_piece を呼び出す必要があります。

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

- **引数が定数かどうかの確認** – UDF は、指定された引数が定数かどうかを示す情報を要求できます。これは、たとえば、_next_value 関数を呼び出すたびに行うのではなく、_next_value 関数の最初の呼び出しで 1 回行うだけで済む場合に便利です。

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

- **NULL 値の応答** – NULL 値を返すには、an_extfn_value で、"data" を NULL に設定します。set_value の呼び出し時、total_len フィールドは無視されます。append が FALSE である場合は、指定されたデータが引数値になります。それ以外の場合は、引数の現在値にデータが追加されます。set_value は、引数の append が TRUE の呼び出しの前に、同じ引数の append が FALSE の呼び出しにより呼び出されると予期されています。固定長データ型 (すなわち、すべての数値データ型) の場合、append フィールドは無視されます。

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

- **文を中断したかどうかの確認** – UDF エントリ・ポイントが長期間 (かなりの秒数) にわたって動作を実行する場合、可能であれば、ユーザが現在の文を中断したかどうかを確認するために、1 または 2 秒ごとに get_is_cancelled コールバックを呼び出します。文が中断されている場合、ゼロ以外の値が返され、UDF エントリ・ポイントはただちに動作を実行します。最終的には、必要なクリーンアップを実行するために _finish_extfn 関数が呼び出されますが、他の UDF エントリ・ポイントが続いて呼び出されることはありません。

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

- **エラー・メッセージの送信** – エラー・メッセージがユーザに送信され、現在の文が停止されるような何らかのエラーが UDF エントリ・ポイントに発生した場合、set_error コールバック・ルーチンが呼び出されます。set_error により、現在の文がロールバックされ、"`Error from external UDF: <error_desc_string>`" と表示されます。SQLCODE は、<error_number> の否定形式です。set_error への呼び出しの後、UDF エントリ・ポイントはただちにリターンを実行します。最終的には、必要なクリーンアップを実行するために _finish_extfn が呼び出されますが、他の UDF エントリ・ポイントが続いて呼び出されることはありません。

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

- **メッセージ・ログの書き込み** – 255 バイトより長いメッセージをトランケートできます。

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

- **ほかのデータ・タイプへの変換**– 入力用です。
  - **an_extfn_value.data** – 入力データ・ポインタ。
  - **an_extfn_value.total_len** – 入力データの長さ。
  - **an_extfn_value.type** – 入力の DT_ datatype。

  出力用です。
  - **an_extfn_value.data** – UDF が提供する出力データ・ポインタ。
  - **an_extfn_value.piece_len** – 出力データの最大長。
  - **an_extfn_value.total_len** – 変換済み出力のサーバ設定長。
  - **an_extfn_value.type** – 対象の出力の DT_ datatype。

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

```
        an_extfn_value   *output
        );
```

- **将来使用するために予約されるフィールド** – 将来使用するために予約されています。

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

- **コンテキストから利用できるデータ** – このデータ・ポイントは、外部ルーチンが要求するコンテキスト・データの使用により、入力できます。UDF はこのメモリを割り付けたり、割り付けを解除したりします。文ごとに、_user_data の 1 つのインスタンスが有効です。中間結果値には、このメモリを使用しないでください。

```
void  * _user_data;
```

- **現在アクティブな計算コンテキスト** – UDF はこのメモリの場所を使用して、集合を計算する中間値を格納します。このメモリは、a_v3_extfn_aggregate で要求されるサイズに基づき、サーバにより割り付けられます。エンジンが、複数のグループについて同時計算を実行することがあるため、中間計算はこのメモリに格納される必要があります。各 UDF エントリ・ポイントの前に、サーバは、正しいコンテキスト・データがアクティブであることを確認します。

```
void  * _user_calculation_context;
```

- **ほかに利用可能な集約情報** – start_extfn を含む、すべての外部関数のエントリ・ポイントで使用できます。ゼロは、不明な値または不適切な値を示します。パーティションまたはグループごとに見積もられた、ローの平均数。
  - **a_sql_uint64 _max_rows_in_frame;** – ウィンドウ・フレームで定義されたローの最大数を計算します。範囲ベースのウィンドウの場合、ユニークな値を示します。ゼロは、不明な値または不適切な値を示します。
  - **a_sql_uint64 _estimated_rows_per_partition;** – パーティションまたはグループごとに見積もられた、ローの平均数を表示します。0 は、不明な値または不適切な値を示します。
  - **a_sql_uint32 _is_used_as_a_superaggregate;** – このインスタンスが通常の集合であるか、スーパー集合であるかを指定します。インスタンスが通常の集合である場合、結果として 0 を返します。
- **ウィンドウ指定の確認** – ウィンドウがクエリに存在する場合の、ウィンドウ指定。
  - **a_sql_uint32 _is_window_used;** – 文がウィンドウ化されるかどうかを指定します。
  - **a_sql_uint32 _window_has_unbounded_preceding;** – 戻り値 0 は、ウィンドウにバインドを解除した先行ローがないことを示します。
  - **a_sql_uint32 _window_contains_current_row;** – 戻り値 0 は、ウィンドウに現在のローが含まれないことを示します。

  - **a_sql_uint32 _window_is_range_based;** – リターン・コードが 1 である場合、ウィンドウは範囲ベースです。リターン・コードが 0 である場合、ウィンドウはローベースです。
- **reset_extfn() セルでの使用** – 現在のパーティションのローの実際の数を返します。ウィンドウ化されない集合の場合は、0 を返します。

```
a_sql_uint64 _num_rows_in_partition;
```

- **ウィンドウ用 evaluate_extfn() セルの使用** – パーティション内の現在評価されているロー番号 (1 で始まる)。これは、無制限ウィンドウの評価フェーズにおいて便利です。

```
a_sql_uint64 _result_row_from_start_of_partition;
```

- **終わりの構文** – 次の構文を使用し、コンテキストを完了します。

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

集合外部関数コンテキスト： a_v3_extfn_aggregate_context

a_v3_extfn_aggregate_context 構造体のメソッド	説明
void set_cannot_be_distributed(a_v3_extfn_aggregate_context * cntxt)	ライブラリ・レベルで分散の基準を満たす場合でも、UDF レベルで分散を無効にできます。 デフォルトでは、ライブラリが分散可能であれば UDF も分散可能と見なされます。 分散を無効にするという判断をサーバに通知することは UDF の役割です。

参照：

- BLOB (a_v4_extfn_blob) (219 ページ)
- BLOB 入力ストリーム (a_v4_extfn_blob_istream) (222 ページ)

例： my_sum 定義

集合 UDF の **my_sum** の例は、整数についてのみ動作します。

my_sum 定義

SUM と同様に **my_sum** は任意のコンテキストで使用できるため、最適化されたすべてのオプションのエントリ・ポイントが提供されています。 次の例では、normal _evaluate_extfn 関数も _evaluate_superaggregate_extfn 関数として使用できます。

```
#include "extfnapiv4.h"
#include <stdlib.h>
#include <assert.h>
```

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


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


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


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


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


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

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

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


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

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


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

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


extern "C"
void my_integer_sum_cum_evaluate(
                        a_v3_extfn_aggregate_context *cntxt,
                        void *arg_handle)
```

```
{
  an_extfn_value  outval;
  an_extfn_value  arg;
  int arg1;
  my_total *cptr = (my_total *)cntxt->_user_calculation_context;

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

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


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

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

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


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

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

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



a_v3_extfn_aggregate my_integer_sum_descriptor =
{
        &my_integer_sum_start,
        &my_integer_sum_finish,
        &my_integer_sum_reset,
        &my_integer_sum_next_value,
        &my_integer_sum_evaluate,
        &my_integer_sum_drop_value,
        &my_integer_sum_cum_evaluate,
        &my_integer_sum_next_subagg_value,
        &my_integer_sum_drop_subagg_value,
        &my_integer_sum_evaluate,
        NULL, // reserved1_must_be_null
        NULL, // reserved2_must_be_null
        NULL, // reserved3_must_be_null
        NULL, // reserved4_must_be_null
        NULL, // reserved5_must_be_null
        0, // indicators
        ( short )sizeof( my_total ), // context size
        8, // context alignment
        0.0, //external_bytes_per_group
        0.0, // external bytes per row
        0, // reserved6_must_be_null
        0, // reserved7_must_be_null
        0, // reserved8_must_be_null
        0, // reserved9_must_be_null
        0, // reserved10_must_be_null
        NULL // _for_server_internal_use
};

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

例： my_bit_xor 定義

集合 UDF の **my_bit_xor** の例は、SA の組み込みの BIT_XOR に似ていますが、**my_bit_xor** は符号なし整数でのみ動作する点が異なります。

my_bit_xor 定義

入力および出力のデータ型が同じであるため、サブ集合値を累計し、スーパー集合結果を生成するには、通常の _next_value_extfn および _evaluate_extfn 関数を使用します。

```
#include "extfnapiv4.h"
#include <stdlib.h>
#include <assert.h>


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


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

#if defined __cplusplus
extern "C" {
#endif

static void my_xor_start(a_v3_extfn_aggregate_context *cntxt)
{
}


static void my_xor_finish(a_v3_extfn_aggregate_context *cntxt)
{
}
```

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


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

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

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


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

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


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

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

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


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

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

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


static a_v3_extfn_aggregate my_xor_descriptor =
    {
        &my_xor_start,
        &my_xor_finish,
        &my_xor_reset,
        &my_xor_next_value,
        &my_xor_evaluate,
        &my_xor_drop_value,
        &my_xor_cum_evaluate,
        &my_xor_next_value,
        &my_xor_drop_value,
        &my_xor_evaluate,
        NULL, // reserved1_must_be_null
        NULL, // reserved2_must_be_null
        NULL, // reserved3_must_be_null
        NULL, // reserved4_must_be_null
        NULL, // reserved5_must_be_null
        0,    // indicators
        ( short )sizeof( my_xor_result ), // context size
```

```
        8,    // context alignment
        0.0,  // external_bytes_per_group
        0.0,  // external bytes per row
        0,    // reserved6_must_be_null
        0,    // reserved7_must_be_null
        0,    // reserved8_must_be_null
        0,    // reserved9_must_be_null
        0,    // reserved10_must_be_null
        NULL  // _for_server_internal_use
    };

a_v3_extfn_aggregate *my_bit_xor()
{
  return &my_xor_descriptor;
}

#if defined __cplusplus
}
#endif
```

例：my_bit_or 定義

集合 UDF の **my_bit_or** の例は、SA の組み込みの BIT_OR に似ていますが、**my_bit_or** は符号なし整数でのみ動作する点が異なり、単純な集合としてのみ使用できます。

my_bit_or 定義

my_bit_or の定義は、**my_bit_xor** の例よりも少し簡単です。

```
#include "extfnapiv4.h"
#include <stdlib.h>
#include <assert.h>


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


typedef struct my_or_result {
```

```
  a_sql_uint32 _or_result;
  a_sql_uint32 _non_null_seen;
} my_or_result;



#if defined __cplusplus
extern "C" {
#endif

static void my_or_start(a_v3_extfn_aggregate_context *cntxt)
{
}

static void my_or_finish(a_v3_extfn_aggregate_context *cntxt)
{
}

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


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

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

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


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

  outval.type = DT_UNSINT;
  outval.piece_len = sizeof(a_sql_uint32);
  if (cptr->_non_null_seen)
```

```
  {
      outval.data = &cptr->_or_result;
  }
  else
  {
      // Return null if no values seen
      outval.data = 0;
  }
  cntxt->set_value( arg_handle, &outval, 0 );
}



static a_v3_extfn_aggregate my_or_descriptor =
{
    &my_or_start,
    &my_or_finish,
    &my_or_reset,
    &my_or_next_value,
    &my_or_evaluate,
    NULL, // drop_val_extfn
    NULL, // cume_eval,
    NULL, // next_subaggregate_extfn
    NULL, // drop_subaggregate_extfn
    NULL, // evaluate_superaggregate_extfn
    NULL, // reserved1_must_be_null
    NULL, // reserved2_must_be_null
    NULL, // reserved3_must_be_null
    NULL, // reserved4_must_be_null
    NULL, // reserved5_must_be_null
    0, // indicators
    ( short )sizeof( my_or_result ), // context size
    8, // context alignment
    0.0, //external_bytes_per_group
    0.0, // external bytes per row
    0, // reserved6_must_be_null
    0, // reserved7_must_be_null
    0, // reserved8_must_be_null
    0, // reserved9_must_be_null
    0, // reserved10_must_be_null
    NULL // _for_server_internal_use
};


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


#if defined __cplusplus
}
#endif
```

例： my_interpolate 定義

集合 UDF の **my_interpolate** の例は、隣接する NULL 値のセットに対して、各方向で最も近い NULL 以外の値への線形補間を実行することにより、シーケンス内の NULL 値に入力しようとする OLAP スタイルの集合 UDF です。

my_interpolate 定義

相応なコストで動作するには、固定幅のローベースのウィンドウを使用して **my_interpolate** を実行する必要がありますが、ユーザは、隣接する NULL 値の予測される最大数に基づいて、ウィンドウの幅を設定できます。 特定のローの入力が NULL でない場合、そのローの結果は入力値と同じです。 この関数は、倍精度浮動小数点数値のセットを取得し、double 型の結果セットを生成します。

```
#include "extfnapiv4.h"
#include <stdlib.h>
#include <assert.h>

//  MY_INTERPOLATE
//
//  OLAP-style aggregate UDF that accepts a double precision
//  floating point argument.  If the current argument value is
//  not NULL, then the result value is the same as the
//  argument value.  On the other hand, if the current row's
//  argument value is NULL, then the result, where possible,
//  will be the arithmetic interpolation across the nearest
//  preceding and nearest following values that are not NULL.
//  In all cases the result is also a double precision value.
//
//  The start function creates a structure for maintaining the
//  argument values within the window including their NULLness.
//  The finish function then deallocates this structure.
//
//  Since there are some strict aggregate usage restrictions
//  for this aggregate (must be used with a row-based window
//  frame that includes the current row), the corresponding
//  SQL declaration will look like:
//
//      CREATE AGGREGATE FUNCTION my_interpolate(IN arg1 DOUBLE)
//                       RETURNS DOUBLE
//                       OVER REQUIRED
//                       WINDOW FRAME REQUIRED
//                              RANGE NOT ALLOWED
//                              PRECEDING REQUIRED
//                              UNBOUNDED PRECEDING NOT ALLOWED
//                              FOLLOWING REQUIRED
//                              UNBOUNDED FOLLOWING NOT ALLOWED
//                       EXTERNAL NAME  'my_interpolate@libudfex'


typedef struct my_window {
  int      _allocated_elem;
```

```
  int      _first_used;
  int      _next_insert_loc;
  int     *_is_null;
  double  *_dbl_val;
  int      _num_rows_in_frame;
} my_window;



#if defined __cplusplus
extern "C" {
#endif


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

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


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

  // Make sure function was defined correctly
  if (!cntxt->_is_window_used)
  {
      cntxt->set_error(cntxt, 20001, "Function requires window");
      return;
  }
  if (cntxt->_window_has_unbounded_preceding ||
      cntxt->_window_has_unbounded_following)
  {
      cntxt->set_error(cntxt, 20002, "Window cannot be unbounded");
      return;
  }
  if (cntxt->_window_is_range_based)
  {
      cntxt->set_error(cntxt, 20003, "Window must be row based");
      return;
  }

  if (!cptr) {
    //
    cptr = (my_window *)malloc(sizeof(my_window));
    if (cptr) {
```

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


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


static void my_interpolate_next_value(a_v3_extfn_aggregate_context
*cntxt,
                                      void *arg_handle)
{
  an_extfn_value  arg;
  double arg1;
  my_window *cptr = (my_window *)cntxt->_user_data;

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

```
  } else {
    cptr->_is_null[curr_cell_num] = 1;
  }

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


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

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


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

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

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

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

```
  //  Compute the result value
  if (cptr->_is_null[curr_cell_num] == 0) {
    //
    //  If the current rows input value is not NULL, then there is
    //  no need to interpolate, just use that input value.
    //
    result = cptr->_dbl_val[curr_cell_num];
    result_is_null = 0;
    //
  } else {
    //
    //  If the current rows input value is NULL, then we do
    //  need to interpolate to find the correct result value.
    //  First, find the nearest following non-NULL argument
    //  value after the current row.
    //
    int rows_following = cptr->_num_rows_in_frame -
                 result_row_offset_from_start_of_frame - 1;
    for (j=0; j<rows_following; j++) {
      tmp_cell_num = ((curr_cell_num + j + 1) % cptr-
>_allocated_elem);
      if (cptr->_is_null[tmp_cell_num] == 0) {
          following_value = cptr->_dbl_val[tmp_cell_num];
          following_value_is_null = 0;
          following_distance = j + 1;
          break;
      }
    }
    //  Second, find the nearest preceding non-NULL
    //  argument value before the current row.
    //
    int rows_before = result_row_offset_from_start_of_frame;
    for (j=0; j<rows_before; j++) {
     tmp_cell_num = ((curr_cell_num + cptr->_allocated_elem - j - 1)
                     % cptr->_allocated_elem);
      if (cptr->_is_null[tmp_cell_num] == 0) {
          preceding_value = cptr->_dbl_val[tmp_cell_num];
          preceding_value_is_null = 0;
          preceding_distance = j + 1;
          break;
      }
    }
    //  Finally, see what we can come up with for a result value
    //
    if (preceding_value_is_null && !following_value_is_null) {
      //
     //  No choice but to mirror the nearest following non-NULL value
      //  Example:
      //
      //    Inputs:  NULL    Result of my_interpolate:  40.0
      //             NULL                               40.0
      //             40.0                               40.0
      //
      result = following_value;
```

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

  //  And last, pass the result value out
  outval.type = DT_DOUBLE;
  outval.piece_len = sizeof(double);
  if (result_is_null) {
    outval.data = 0;
  } else {
    outval.data = &result;
  }
  cntxt->set_value( arg_handle, &outval, 0 );
}


static a_v3_extfn_aggregate my_interpolate_descriptor =
    {
        &my_interpolate_start,
        &my_interpolate_finish,
        &my_interpolate_reset,
```

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

a_v3_extfn_aggregate *my_interpolate()
  { return &my_interpolate_descriptor; }

#if defined __cplusplus
  }
#endif
```

集合 UDF のコンテキスト記憶領域

コンテキスト領域は、同じクエリ内 (特に、OLAP スタイルのクエリ内) の UDF の複数の呼び出し間でのデータの転送または通信用に使用されます。

コンテキスト変数は、UDF 自体が集合関数の中間結果を管理する (UDF を逐次的に実行するように Sybase IQ サーバを強制する) かどうか、または Sybase IQ サーバがメモリを管理するかどうかを制御します。

_calculation_context_size が 0 に設定されている場合、UDF は、メモリ内のすべての中間結果を管理する必要があります (OLAP クエリの実行中に多数の UDF インスタンスを並列で呼び出さずに、データ上で UDF を順番に呼び出すように Sybase IQ サーバを強制します)。

_calculation_context_size が 0 以外の値に設定されている場合、Sybase IQ サーバは、UDF の呼び出しごとに個別のコンテキスト領域を管理するため、UDF の複数のインスタンスを並列で呼び出すことができます。メモリを最も効率的に使用するには、(必要なコンテキスト記憶領域のサイズに応じて) デフォルトよりも小さな _calculation_context_alignment 値を設定することを検討します。

コンテキスト記憶領域の詳細については、「集合 UDF 記述子構造 (66 ページ)」の _calculation_context_size および _calculation_context_alignment の説明を参照してください。これらの変数は、記述子構造の末尾の近くにあります。

コンテキスト記憶領域の使用の詳細については、「計算コンテキスト (70 ページ)」を参照してください。

重要： メモリ内の中間結果を集計 UDF 内に格納するには、_start_extfn 関数を使用してメモリを初期化し、_finish_extfn 関数を使用してメモリのクリーンアップおよび割り付け解除を行います。

参照：

- 集合 UDF の宣言 (55 ページ)
- 集合 UDF の定義 (63 ページ)

スカラ UDF と集合 UDF 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。

次の Interactive SQL 文を実行すると、姓と名前を含んでいる 2 つのカラムから氏名が返されます。

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

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

次の文を実行すると、指定された姓と名前から氏名が返されます。

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

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

fullname 関数は、この関数に対する実行パーミッションが付与されているどのユーザでも使用できます。

スカラ UDF と集合 UDF のパターン呼び出し

パターン呼び出しは、結果の収集時に関数が実行する手順です。

スカラ UDF と集合 UDF のコールバック関数

エンジンは a_v3_extfn_scalar_context 構造を通じて以下のコールバック関数を提供しており、ユーザの UDF 関数内でこれらが使用されます。

- **get_value –** 各入力引数の値を取得するために、評価メソッドで使用される関数。 引数のデータ型の範囲が狭い場合 (256 バイト未満)、get_value の呼び出しで十分に引数値全体を取得できます。 引数のデータ型の範囲が広い場合、このコールバックに渡される an_extfn_value 構造の piece_len フィールドが total_len フィールドの値より小さい値を返すときは、次の get_piece コールバックを使用して残りの入力値を取得します。
- **get_piece –** 長い引数入力値の後続のフラグメントを取得するために使用される関数。

  注意： get_piece コールバックは、v3 と v4 のスカラ UDF と集合 UDF で有効です。v4 のテーブル UDF と TPF については、BLOB (a_v4_extfn_blob) および BLOB 入力ストリーム (a_v4_extfn_blob_istream) の構造体を代わりに使用します。
- **get_is_constant –** 指定される入力引数値が定数かどうかを決定する関数。 この関数は、たとえば、評価関数を呼び出すたびに行うのではなく、_evaluate_extfn 関数の最初の呼び出しで 1 回行うだけで済む場合など、UDF の最適化に便利です。
- **set_value –** サーバにこの呼び出しの UDF の結果値を通知するために、評価関数内で使用される関数。 結果のデータ型の範囲が狭い場合は、set_value を 1 回呼び出すだけで十分です。一方、結果のデータ値の範囲が広い場合は、値全体を渡すには set_value を複数回呼び出す必要があり、最後の呼び出しを除き、各フラグメントについてコールバックへの append 引数が true である必要があります。 NULL 結果を返すには、UDF は、結果値の an_extfn_value 構造にあるデータ・フィールドを NULL ポインタに設定する必要があります。
- **get_is_cancelled –** 文がキャンセルされているかどうかを決定する関数。 UDF エントリ・ポイントが長期間 (かなりの秒数) にわたって動作を実行する場合、可能であれば、ユーザが現在の文を中断したかどうかを確認するために、1 秒または 2 秒ごとに get_is_cancelled コールバックを呼び出します。文が中断されていない場合、戻り値は 0 です。

  Sybase IQ では、きわめて大きなデータ・セットを処理できるため、一部のクエリが長時間にわたって実行される可能性があります。ときには、クエリの実

行時間が異常に長くなることもあります。長時間経過しても処理が完了しない場合は、SQL クライアントでクエリをキャンセルできます。ネイティブ関数は、ユーザがいつクエリをキャンセルしたかを追跡します。ユーザによってクエリがキャンセルされたかどうかを追跡するように UDF を記述する必要もあります。つまり、UDF は、長時間実行されている呼び出し元のクエリをユーザがキャンセルする機能をサポートする必要があります。

- **set_error –** サーバに (最終的にはユーザに) エラーを返すために使用される関数。 エラー・メッセージがユーザに送信されるようなエラーが UDF エントリ・ポイントに発生した場合、このコールバック・ルーチンを呼び出します。呼び出されると、set_error により現在の文がロールバックされ、ユーザは "`Error from external UDF: error_desc_string`" というメッセージを受け取ります。SQLCODE は、指定された error_number の否定形式です。既存のエラーとの衝突を避けるため、UDF は 17000 から 99999 までの error_number 値を使用します。"error_desc_string" の最大長は 140 文字です。
- **log_message –** サーバのメッセージ・ログにメッセージを送信するために使用される関数。文字列は、255 バイト以内の印刷可能なテキスト文字列です。
- **convert_value –** 異なるデータ型の間のデータ変換を可能にする関数。主に、DT_DATE、DT_TIME、および DT_TIMESTAMP、および DT_TIMESTAMP_STRUCT の間の変換に使用されます。入力および出力の an_extfn_value は、この関数に渡されます。

参照：

- スカラ UDF のパターン呼び出し (95 ページ)
- 集合 UDF のパターン呼び出し (96 ページ)
- BLOB (a_v4_extfn_blob) (219 ページ)
- BLOB 入力ストリーム (a_v4_extfn_blob_istream) (222 ページ)

スカラ UDF のパターン呼び出し

スカラ UDF パターン呼び出しの場合、指定された関数ポインタについて予測されるパターン呼び出しは、次のようになります。

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

参照：

- スカラ UDF と集合 UDF のコールバック関数 (94 ページ)
- 集合 UDF のパターン呼び出し (96 ページ)

集合 UDF のパターン呼び出し

ユーザが指定した集合 UDF 関数のパターン呼び出しは、スカラ関数のパターン呼び出しより、複雑で多様です。

以下の例では、次のテーブル定義を使用します。

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

次の省略形が使用されます。

RR = a_v3_extfn_aggregate_context. _result_row_offset_from_start_of_partition – この値は、値が計算される現在のパーティションにある、現在のロー番号を示します。この値は、ウィンドウ集合の実行時に設定され、無制限ウィンドウの評価手順の実行時に使用されます。この値は、すべての評価呼び出しで使用できます。

Sybase IQ は、マルチユーザ・アプリケーションです。多くのユーザが同じ UDF を同時に実行できます。特定の OLAP クエリによって、UDF が同じクエリ内で何度も実行されたり、ときには並列で実行されたりします。

参照：

- スカラ UDF と集合 UDF のコールバック関数 (94 ページ)
- スカラ UDF のパターン呼び出し (95 ページ)

単純な非グループ化集合

単純な非グループ化集合のパターン呼び出しは、すべてのローの入力値を合計して、結果を生成します。

クエリ

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

パターン呼び出し

```
_start_extfn(cntxt)
_reset_extfn(cntxt)
_next_value_extfn(cntxt, args)  -- input a=1
_next_value_extfn(cntxt, args)  -- input a=2
_next_value_extfn(cntxt, args)  -- input a=3
_next_value_extfn(cntxt, args)  -- input a=4
_next_value_extfn(cntxt, args)  -- input a=5
_next_value_extfn(cntxt, args)  -- input a=6
_evaluate_extfn(cntxt, args)    -- returns 21
_finish_extfn(cntxt)
```

結果

```
my_sum(a)
21
```

単純なグループ化集合

単純なグループ化集合のパターン呼び出しは、グループのすべてのローの入力値を合計して、結果を生成します。**_reset_extfn** は、グループの先頭を示します。

クエリ

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

パターン呼び出し

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

結果

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

無制限ウィンドウにおける OLAP スタイルの集合のパターン呼び出し

"b" での分割は、"b" でのグループ化と同じパーティションを作成します。無制限ウィンドウでは、パーティションのローごとに "a" 値が評価されます。これは無制限クエリであるため、すべての値は最初に UDF に渡され、その後に評価サイクルが続きます。_window_has_unbounded_preceding および _window_has_unbounded_following のコンテキスト・インジケータは 1 に設定されます。

クエリ

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

パターン呼び出し

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

結果

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

OLAP スタイルの非最適化累積ウィンドウの集合

_evaluate_cumulative_extfn が指定されない場合、次のパターン呼び出しにより、この累積和が評価されます。これは、_evaluate_cumulative_extfn よりも効率が低下します。

クエリ

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

パターン呼び出し

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

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

結果

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

OLAP スタイルの最適化累積ウィンドウの集合

_evaluate_cumulative_extfn が指定され、next_value/evaluate シーケンスが各パーティション内の各ローの 1 つの _evaluate_cumulative_extfn 呼び出しに結合される場合、この累積和が評価されます。

クエリ

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

パターン呼び出し

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

結果

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

OLAP スタイルの非最適化移動ウィンドウの集合

_drop_value_extfn 関数が指定されていない場合、この移動ウィンドウの和が評価されます。これは、_drop_value_extfn を使用する場合よりも効率が大幅に低下します。

クエリ

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

パターン呼び出し

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

結果

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

OLAP スタイルの最適化移動ウィンドウの集合

_drop_value_extfn 関数が指定された場合、次のパターン呼び出しを使用して、この移動ウィンドウの和が評価されます。これは、_drop_value_extfn を使用する場合よりも効率的です。

クエリ

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

パターン呼び出し

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

結果

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

後続の OLAP スタイルの非最適化移動ウィンドウの集合

_drop_value_extfn 関数が指定されていない場合、次のパターン呼び出しを使用して、この移動ウィンドウの和が評価されます。この例は前述の移動ウィンドウの例と似ていますが、評価されるローは、next_value 関数で指定された最後のローではありません。

クエリ

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

パターン呼び出し

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

結果

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

後続の OLAP スタイルの最適化移動ウィンドウの集合

_drop_value_extfn 関数が指定されている場合、次のパターン呼び出しを使用して、この移動ウィンドウの和が評価されます。この例も前述の移動ウィンドウの例と似ていますが、評価されるローは、next_value 関数で指定された最後のローではありません。

クエリ

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

パターン呼び出し

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

結果

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

現在のローのない OLAP スタイルの非最適化移動ウィンドウ

UDF の my_sum が組み込みの SUM と同様に動作することを前提とします。_drop_value_extfn 関数が指定されていない場合、次のパターン呼び出しを使用して、この移動ウィンドウのカウントが評価されます。この例は前述の移動ウィンドウの例と似ていますが、現在のローがウィンドウ・フレームに含まれません。

クエリ

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

パターン呼び出し

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

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

結果

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

現在のローのない OLAP スタイルの最適化移動ウィンドウ

_drop_value_extfn 関数が指定されている場合、次のパターン呼び出しを使用して、この移動ウィンドウのカウントが評価されます。この例は前述の移動ウィンドウの例と似ていますが、現在のローがウィンドウ・フレームに含まれません。

クエリ

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

パターン呼び出し

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

```
_next_value_extfn(cntxt, args)      input a=5
_evaluate_extfn(cntxt, args)          returns 12
_finish_extfn(cntxt)
```

結果

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

外部関数のプロトタイプ

API は、Sybase IQ のインストール・ディレクトリのサブディレクトリにある `extfnapiv3.h` (v4 API では `extfnapiv4.h`) という名前のヘッダ・ファイルで定義します。 このヘッダ・ファイルは、外部関数のプロトタイプのプラットフォーム依存機能を処理します。

ライブラリを記述する際に古い API が使用されていないことをデータベース・サーバに通知するため、次のような関数を使用します。

```
uint32 extfn_use_new_api( )
```

この関数は、32 ビットの符号なし整数を返します。戻り値がゼロ以外の場合、データベース・サーバでは、新しい API が使用されているものと見なします。

DLL がこの関数をエクスポートしない場合は、データベース・サーバでは、古い API が使用されているものと見なします。 新しい API を使用しているときには、戻り値は `extfnapi.v4h` で定義されている API バージョン番号でなくてはなりません。

各ライブラリは、この関数の実装とエクスポートを次のように行います。

```
unsigned int extfn_use_new_api(void)
    {
        return EXTFN_V4_API;
    }
```

この関数が存在し、`EXTFN_V4_API` を返すことで、このマニュアルで説明している新しい API 向けに作成された UDF がライブラリに含まれていることが Sybase IQ エンジンに通知されます。

関数プロトタイプ

関数の名前は、**CREATE PROCEDURE** 文または **CREATE FUNCTION** 文で参照されるものと一致しなくてはなりません。次のように関数を宣言します。

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

関数の戻り値は void とします。引数は 2 つです。1 つは、引数の受け渡しに使用する構造、もう 1 つは、SQL プロシージャから渡された引数へのハンドルです。

an_extfn_api 構造の形式は、次のとおりです。

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

注意： get_piece コールバックは、v3 と v4 のスカラ UDF と集合 UDF で有効です。v4 のテーブル UDF と TPF については、BLOB (a_v4_extfn_blob) および BLOB 入力ストリーム (a_v4_extfn_blob_istream) の構造体を代わりに使用します。

an_extfn_value 構造の形式は、次のとおりです。

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

注意事項

OUT パラメータに対して get_value を呼び出すと、引数のデータ型が返り、データとして NULL が返ります。

引数に対して get_piece 関数を呼び出せるのは、同じ引数に対して get_value 関数を呼び出した直後に限られます。

NULL を返すには、an_extfn_value のデータを NULL に設定します。

set_value の append フィールドは、データを置き換えるかどうかを決定します。false の場合は、既存のデータを指定されたデータに置き換え、true の場合は、既存のデータに指定されたデータを追加します。 set_value を append=TRUE で呼び出す前に、同じ引数を append=FALSE で呼び出しておく必要があります。固定長データ型の場合、append フィールドは無視されます。

ヘッダ・ファイル自体にも、補足的な注意書きが記述されています。

参照：

- BLOB (a_v4_extfn_blob) (219 ページ)
- BLOB 入力ストリーム (a_v4_extfn_blob_istream) (222 ページ)

テーブル UDF と TPF

テーブル UDF は、C、C++、または Java によるユーザ定義の外部テーブル関数です。スカラ UDF や集合 UDF と違い、テーブル UDF はローのセットを出力として生成します。SQL クエリではテーブル UDF の出力セットをテーブル式として使用します。

スカラ UDF と集合 UDF では v3 API または v4 API を使用できますが、テーブル UDF で使用できるのは v4 のみです。

テーブル UDF の SQL 関数の宣言には **CREATE PROCEDURE** 文を使用します。スカラ UDF と集合 UDF で使用するのは **CREATE FUNCTION** 文です。

テーブル・パラメータ関数 (TPF) は、テーブル UDF を拡張したもので、スカラ値またはロー・セットを入力として受け取ります。

参照：
- テーブル・パラメータ関数 (152 ページ)
- スカラ UDF の宣言と定義 (39 ページ)
- 集合 UDF の宣言と定義 (54 ページ)
- 学習ロードマップ： 外部 C と C++ UDF の種類 (6 ページ)
- Java テーブル UDF の作成 (386 ページ)

ユーザの役割

テーブル UDF を扱うユーザは 2 種類に分かれます。UDF 開発者と SQL アナリストです。

- **UDF 開発者** – C または C++ でテーブル UDF を開発します。
- **SQL アナリスト** – **FROM** 句でテーブル式を参照する SQL クエリを開発および分析します。テーブル式は、テーブル UDF によって生成されるローのセットです。

参照：
- テーブル UDF 開発者向けのロードマップ (110 ページ)
- SQL アナリスト向けのロードマップ (111 ページ)

テーブル UDF 開発者向けのロードマップ

コメント付きの例を使用して、C または C++ のテーブル UDF の開発方法を学習します。開発作業が完了したら、SQL アナリストがその UDF を SQL クエリで参照できます。

このロードマップの前提は次のとおりです。

- マシンに C または C++ の開発環境がある。
- 標準的なプログラミング手法を理解している。

タスク	参照先
テーブル UDF と TPF の制限について理解します。	テーブル UDF の制限 (111 ページ)
テーブル UDF を作成します。	テーブル UDF の開発 (115 ページ)
(オプション) 分散クエリ処理 (DQP: Distributed Query Processing) 用のライブラリ・バージョン・バリデータを定義します。	ライブラリ・バージョン (extfn_get_library_version) (21 ページ) ライブラリ・バージョンの互換性 (extfn_check_version_compatibility) (22 ページ)
ソース・コードをコンパイルおよびリンクします。	ダイナミック・リンク・ライブラリを構築するためのソース・コードのコンパイルとリンク (24 ページ)
CREATE PROCEDURE 文を使用して、サーバに対して UDF を宣言します。それらの文をコマンドとして記述および実行するか、Sybase Central または Sybase Control Center を使用して **CREATE** 文を発行します。	SQL アナリスト向けのロードマップ (111 ページ)

SQL アナリスト向けのロードマップ

C または C++ のテーブル UDF を SQL クエリで参照します。

タスク	参照先
`.dll` ファイルまたは `.so` ファイル (たとえば `myudf.dll`) を UDF 開発者から入手します。 `.dll` ファイルは `bin64` ディレクトリに配置します。`.so` ファイルは `lib64` または `LD_LIBRARY_PATH` ディレクトリに配置します。	なし。

CREATE PROCEDURE 文を定義します。その中で、`.dll` ファイルとコールバック関数を参照します。
例を示します。
`CREATE PROCEDURE my_udf( IN num_row INT )`
`RESULT( id INT )`
`EXTERNAL NAME 'udf_rg_proc@myudf.dll'`
	CREATE PROCEDURE 文 (テーブル UDF) (188 ページ)

UDF からローを選択します。 例を示します。 `SELECT * FROM my_udf(5)`	SELECT 文 (207 ページ) FROM 句 (200 ページ)

参照：

- テーブル UDF と TPF クエリ用 SQL リファレンス (186 ページ)

テーブル UDF の制限

テーブル UDF と TPF にはいくつかの制限があります。

- IQ_IDA ライセンスのユーザは、テーブル UDF と TPF をマルチプレックスのリーダ・ノードでのみ実行できます。
- 外部プロシージャには **TEMPORARY PROCEDURE** 句は使用できません。テンポラリ外部プロシージャを作成しようとすると、作成時にエラーになります。
- **NO RESULT SET** 句は使用できません。テーブル UDF と TPF では結果の内容を明示的に宣言してください。
- オプションの **DYNAMIC RESULT SETS integer-expression** 句を指定する場合、値は 1 に設定する必要があります。テーブル UDF と TPF は複数の結果セットを返しません。

- SQL 文 **CALL** および埋め込み SQL 文 **EXEC** ではテーブル UDF と TPF を参照できません。テーブル UDF と TPF を参照できるのは SQL 文の **FROM** 句のみです。
- **LANGUAGE** 句はテーブル UDF と TPF には使用できません。**LANGUAGE** 句がある場合、実行時に構文エラーがレポートされます。
- **parameter** 句はキーワード **IN** に限定されます。キーワード **INOUT** および **OUT** はテーブル UDF と TPF ではサポートされていません。
- **EXTERNAL NAME** 句は、スカラ UDF および集合 UDF と同じ構文です。

最初の作業

テーブル UDF と TPF の開発に入る前に、サンプル・ファイル、概念、制限について理解します。

サンプル・ファイル

テーブル UDF のサンプル・ファイルはサーバとともにインストールされます。独自のテーブル UDF を定義するときには、これらのサンプルをモデルとして使用します。

サンプル・ファイルは次の場所にあります。

- `%ALLUSERSPROFILE%¥samples¥udf` (Windows)
- `$IQDIR15/samples/udf` (UNIX)

ファイル	説明
`apache_log_reader.cxx`	Apache のログ・ファイルを読み取り、そのファイルから取得したローをテーブル形式で提示するテーブル UDF の実装。この UDF は、コンピュータが生成したデータを SQL クエリの作成者がリアルタイムで使用できるように UDF を活用する方法を示す実際的な例です。
`build.sh / build.bat`	`samples/udf` ディレクトリに格納されているサンプルのスカラ UDF、集合 UDF、テーブル UDF、TPF をコンパイルおよびリンクするスクリプト。
`my_md5.cxx`	入力ファイル (LOB のバイナリ引数) の MD5 ハッシュ値を計算する、単純な決定的スカラ UDF。
`tpf_agg.cxx`	入力テーブルのローを使用して、入力データの集約を実行し、ローをサーバに返します。

ファイル	説明
tpf_blob.cxx	入力テーブルから LOB データを読み込み、その中に含まれている指定の文字または数字が偶数個の場合にはデータを結果セットに引き渡す TPF の実装。この TPF は、LOB データを読み込む方法と、LOB データ型を結果セットに引き渡す方法を示します。
tpf_dt.cxx	
tpf_filt.cxx	TPF を使用してローをフィルタする方法を示します。この例は、呼び出し元が指定したロー・ブロックを使用し、入力テーブル・パラメータにそのブロックを引き渡します。入力のテーブル・スキーマはこの関数の出力の結果セットと一致する必要があります。
tpf_oby.cxx	順序付けした出力を TPF で生成して引き渡す方法を示します。
tpf_pby.cxx	パーティション分割した出力を TPF で生成して引き渡す方法を示します。
tpf_rg_1.cxx	テーブル UDF のサンプル `udf_rg_2.cxx.` と同様です。入力パラメータに基づいて複数ローのデータを生成します。
tpf_rg_2.cxx	`tpf_rg_1.cxx` のサンプルを基にしていますが、入力テーブルからのローの読み込みに `fetch_block` ではなく `fetch_into` を使用します。
udf_main.cxx	このファイルはすべての例にリンクされており、v4 API で必須となっている共通のエントリ・ポイントが記述されています。こうすることでコードを再利用でき、それぞれの例でこれらのエントリ・ポイントを記述する必要がありません。
udf_rg_1.cxx	複数ローの整数データを生成する単純なテーブル UDF。
udf_rg_2.cxx	複数ローの整数データを生成する単純なテーブル UDF。`describes` を使用して、SQL で定義されているスキーマと UDF の実装を確実に一致させています。また、オプティマイザの属性も記述しています。
udf_rg_3.cxx	整数データをブロックで生成する単純なテーブル UDF。フェッチ・メソッド `_fetch_block` を使用して、100 のブロックで生成します。
udf_utils.cxx	UDF/TPF 開発者に役立つ一連のユーティリティ関数とマクロ。サンプルの中でこのファイルの要素が使用されています。
udf_utils.h	UDF/TPF 開発者に役立つ一連のユーティリティ関数とマクロのためのヘッダ・ファイル。

プロデューサとコンシューマ

サーバと UDF は、データのローをやり取りするときに、プロデューサとコンシューマという関係を形成します。

プロデューサとコンシューマは、テーブルのローのデータを生成する側と利用する側のどちらであるかを表します。プロデューサはテーブルのローを生成する側、コンシューマは利用する側です。

サーバは、スカラ UDF と集合 UDF については、クエリに一致するそれぞれのローに対して UDF を 1 回ずつ実行します。これらの UDF は、入力のスカラ・パラメータを利用し、単一のスカラ・パラメータを生成して返します。このデータ交換は、`evaluate` メソッドの間に `get_value()` および `set_value()` API を使用して行われます。

しかし、UDF でテーブルを生成または利用することが必要な場合には、スカラ値を生成および利用する方法ではデータ交換が効率的ではありません。テーブルを生成するテーブル UDF と、テーブルを利用する TPF では、v4 API の `row block` データ構造体を使用します。ロー・ブロックは、ロー・データとカラム・データのバルク交換に対応しています。ロー・ブロックにデータを格納するのがプロデューサ、ロー・ブロックを読み込むのがコンシューマです。

次の例では、テーブル UDF `my_table_udf()` がデータのプロデューサ、 サーバの Sybase IQ がデータのコンシューマです。

```
SELECT * FROM my_table_udf()
```

一般に、テーブル UDF は常にデータのプロデューサです。 しかし、次の例のように、サーバは必ずしもコンシューマであるとは限りません。

```
SELECT * FROM my_tpf( TABLE( SELECT * FROM my_table_udf() ) )
```

外側の TPF である `my_tpf()` は、**SELECT * from my_table_udf()** と指定したテーブル入力パラメータのコンシューマです。Sybase IQ は、TPF `my_tpf()` が生成したテーブルのコンシューマです。 したがって、TPF は、コンシューマとプロデューサの両方になり得ます。

TPF がコンシューマとして利用する対象は、テーブル UDF の出力でなくてもかまいません。次の例では、TPF は、内部クエリが生成したテーブル・データを利用しています。この内部クエリは Sybase IQ サーバが生成しています。

```
SELECT * FROM my_tpf( TABLE( SELECT * FROM my_table where my_table.c1
< 10 ) )
```

したがって、TPF では、Sybase IQ はテーブル・データのコンシューマとプロデューサのどちらにもなり得ます。

v4 API では、データの生成先および利用元となるメモリ領域をロー・ブロックにより定義します。一般に、ロー・ブロックのレイアウトは、テーブルのローおよびカラムのフォーマットと概念的に一致します。つまり、ロー・ブロックは複数のローで構成され、それぞれのローは複数のカラムで構成されます。 ロー・ブロックは、プロデューサまたはコンシューマが確保し、適切な時点で解放します。

ローとカラムには、これらにのみ適用される固有の属性があります。たとえばローには、そのローが存在するかどうかを示すステータス・フラグがあります。このフラグを使用することで、TPF はカラム・データを移動することなくローのステータスを変更できます。 またカラムには、データ値が NULL かどうかを示す NULL マスクがあります。ロー・ブロックにも追加的な属性があります。 たとえば、ローの最大数や現在のロー数などです。ロー・ブロックのこれらの属性が役に立つのは、大規模なローのセットを処理するロー・ブロックを UDF で作成しつつも、必要に応じてもっと少ない数のローを生成したいという場合です。

ローを利用する処理は、次の 2 つのフェッチ API のいずれかを通じて行われます。

- `fetch_into`
- `fetch_block`

`fetch_into` は、コンシューマがロー・ブロックを確保してプロデューサに渡す場合に呼び出します。この場合プロデューサは、最大ロー数までの範囲でできるだけ多くのローを格納するように要求されます。 `fetch_block` は、コンシューマがプロデューサにロー・ブロックを確保してほしい場合に呼び出します。`Fetch_block` は、データのローをフィルタする TPF を開発するときに効率的です。 サーバ (コンシューマ) はロー・ブロックを確保し、`fetch_into` API を使用して TPF からフェッチします。 続いて TPF は、`fetch_block` API を使用して、同じロー・ブロックを入力パラメータに渡すことができます。

参照：
- ロー・ブロックのデータ交換 (143 ページ)

テーブル UDF の開発

一般に、テーブル UDF を開発する手順は、入力と出力の決定、v4 ライブラリの宣言、`a_v4_extfn_proc` 記述子の定義、ライブラリのエントリ・ポイント関数の定義、サーバがローの情報を取得する方法の定義、`a_v4_extfn_proc` 構造体の関数の実装、`a_v4_extfn_table_func` 構造体の関数の実装で構成されます。

1. テーブル UDF の入力と出力を決定します。

   入力はプロシージャが受け取るパラメータで定義し、出力はプロシージャの **RESULT** 句の宣言方法により定義します。SQL でのテーブル UDF の宣言は、

テーブル UDF の実装とは分離しています。つまり、テーブル UDF の特定の実装を、特定の宣言と結び付けることができます。テーブル UDF を開発するときには、実装と宣言を一致させてください。

2. ライブラリを v4 ライブラリとして宣言します。

   Sybase IQ がライブラリを v4 ライブラリとして認識するためには、Sybase IQ のインストール・ディレクトリのサブディレクトリにある `extfnapiv4.h` ヘッダ・ファイルをライブラリでインクルードします。

   このヘッダは v4 API の機能と関数を定義しており、v3 API のスーパーセットです。つまり、`extfnapiv4.h` は `extfnapiv3.h` を含んでいます。

   テーブル UDF または TPF を作成するために、ライブラリでは `extfn_use_new_api()` エントリ・ポイントを用意します。v4 ライブラリの場合、`extfn_use_new_api()` は `EXTFN_V4_API` を返します。

3. `a_v4_extfn_proc` 記述子を定義します。

   v4 のテーブル UDF または TPF を開発するときには、サーバが呼び出すことのできる関数をライブラリで宣言します。

   `a_v4_extfn_proc` 型の変数を作成し、この構造体のそれぞれのメンバに、テーブル UDF 内で実装した該当の関数のアドレスを設定します。この変数の情報は、ライブラリのエントリ・ポインタを通じてサーバが使用できます。`a_v4_extfn_proc` のメンバの中には必須ではないものもあります。また、2 つの予約済みフィールドがあり、それらは NULL に設定します。

   独自の関数を開発するときには、次の記述子関数をモデルとして使用してください。

```
static a_v4_extfn_proc udf_proc_descriptor =
{
    udf_proc_start,      // optional
    udf_proc_finish,     // optional
    udf_proc_evaluate,   // required
    udf_proc_describe,   // required
    udf_proc_enter_state,// optional
    udf_proc_leave_state,// optional
    NULL,                // Reserved: must be NULL
    NULL                 // Reserved: must be NULL
};
```

4. ライブラリのエントリ・ポイント関数を定義します。

   テーブル UDF ライブラリに、`a_v4_extfn_proc` 記述子のポインタを返す関数エントリ・ポイントを用意します。この記述子は、手順 3 で説明したものと同じです。

   このコールバック関数が、ライブラリでメインとなる必須のエントリ・ポイントです。

独自のライブラリのエントリ・ポイントを開発するときには、次の関数をモデルとして使用してください。

```
extern "C"
a_v4_extfn_proc * SQL_CALLBACK udf_rg_proc()
/*****************************************/
{
    return &udf_proc_descriptor;
}
```

5. サーバがテーブル UDF からローの情報を取得する方法を定義します。

   v4 のテーブル UDF または TPF を開発するときには、ローの情報をサーバに伝える方法をライブラリで宣言します。

   `a_v4_extfn_table_func` 型の変数を作成し、この構造体のそれぞれのメンバに、テーブル UDF 内で実装した該当の関数のアドレスを設定します。この変数の情報は、サーバが実行時に使用できるようになります。

   `a_v4_extfn_table_func` のメンバの中には必須ではないものもあります。また、2 つの予約済みフィールドがあり、それらは NULL に設定します。

   独自のテーブル UDF を開発するときには、次の記述子をモデルとして使用してください。

```
{
    udf_table_func_open,         // required
    udf_table_func_fetch_into,  // one of fetch_into or
fetch_block required
    udf_table_func_fetch_block, // one of fetch_into or
fetch_block required
    udf_table_func_rewind,       // optional
    udf_table_func_close,        // required
    NULL,                        // Reserved: must be NULL
    NULL                         // Reserved: must be NULL
};
```

   実行を開始する時点で、サーバは `a_v4_extfn_proc` 関数 `_evaluate_extfn` を呼び出します。これにより、テーブル UDF は、自らが実装したテーブル関数をサーバに伝えることができます。そのために、テーブル UDF は、サーバに渡す `a_v4_extfn_table` のインスタンスを作成します。この構造体は、`a_v4_extfn_table_func` 記述子のポインタと、結果セットのカラム数を保持する構造体です。

   独自のテーブル UDF を開発するときには、次の記述子をモデルとして使用してください。

```
static a_v4_extfn_table_udf_rg_table = {
    &udf_table_funcs, // Table function descriptor
    1                 // number_of_columns
};
```

6. `a_v4_extfn_proc` 構造体の関数を実装します。

   テーブル UDF には、手順 3 の `a_v4_extfn_proc` 記述子で宣言した `a_v4_extfn_proc` 関数をそれぞれ実装します。
7. `a_v4_extfn_table_func` 構造体の関数を実装します。

   テーブル UDF には、手順 5 の `a_v4_extfn_table_func` 記述子で宣言した `a_v4_extfn_table_func` 関数をそれぞれ実装します。

参照：

- スカラ UDF と集合 UDF のパターン呼び出し (94 ページ)
- udf_rg_2 (125 ページ)
- udf_rg_3 (128 ページ)
- サンプルのテーブル UDF udf_rg_1 の実装 (119 ページ)
- テーブル UDF の実装例 (118 ページ)
- 外部関数 (a_v4_extfn_proc) (314 ページ)
- テーブル関数 (a_v4_extfn_table_func) (346 ページ)
- _evaluate_extfn (315 ページ)

テーブル UDF の実装例

実装例は、簡単なテーブル UDF から始まり、しだいに複雑になっていきます。

テーブル UDF の実装例はサンプル・ディレクトリにあります。最初は簡単なテーブル UDF の例から始まり、先に進むにつれて複雑さや機能が増していきます。

これらの例は、`libv4apiex` というコンパイル済みのダイナミック・ライブラリで使用できます。(ライブラリ名の拡張子はプラットフォームによって異なります)。このライブラリは `udf_main.cxx` で定義された関数とリンクされています。この中には、**extfn_use_new_api** などのライブラリ・レベル関数が含まれています。`libv4apiex` は、サーバが読み取ることのできるディレクトリに配置してください。

参照：

- udf_rg_1.cxx で実装されたサンプルのテーブル UDF の実行 (124 ページ)
- udf_rg_2.cxx で実装されたサンプルのテーブル UDF の実行 (128 ページ)
- udf_rg_3.cxx で実装されたサンプルのテーブル UDF の実行 (132 ページ)

サンプルのテーブル UDF udf_rg_1 の実装

udf_rg_1 というサンプルのテーブル UDF は、v4 のテーブル UDF で n 個のローのデータを生成する方法を示しています。このテーブル UDF の実装は、サンプル・ディレクトリの udf_rg_1.cxx にあります。

1. テーブル UDF の入力と出力を決定します。

   この例では、入力パラメータの値に基づいて *n* 個のローのデータを生成します。入力は単一の整数パラメータ、出力は integer 型の単一のカラムを持つローです。

   このプロシージャを定義するために必要な **CREATE PROCEDURE** 文は次のとおりです。

   ```
   CREATE OR REPLACE PROCEDURE udf_rg_1( IN num INT )
   RESULT( c1 INT )
   EXTERNAL NAME 'udf_rg_1@libv4apiex'
   ```

2. ライブラリを v4 ライブラリとして宣言します。

   この例では、udf_rg_1.cxx で extfnapiv4.h ヘッダ・ファイルを次のようにインクルードしています。

   ```
   #include "extfnapiv4.h"
   ```

   このライブラリに v4 のテーブル UDF が含まれていることをサーバに伝えるために、次のような関数エクスポートが udf_main.cxx で定義されています。

   ```
   a_sql_uint32 SQL_CALLBACK extfn_use_new_api( void )
   /***********************************************/
   {
       return EXTFN_V4_API;
   }
   ```

3. a_v4_extfn_proc 記述子を定義します。

   必要な記述子を udf_rg_1.cxx で次のように宣言しています。

   ```
   static a_v4_extfn_proc udf_rg_descriptor =
   {
       NULL,             // _start_extfn
       NULL,             // _finish_extfn
       udf_rg_evaluate, // _evaluate_extfn
       udf_rg_describe, // _describe_extfn
       NULL,             // _leave_state_extfn
       NULL,             // _enter_state_extfn
       NULL,             // Reserved: must be NULL
       NULL              // Reserved: must be NULL
   };
   ```

4. ライブラリのエントリ・ポイント関数を定義します。

このコールバック関数ではメインのエントリ・ポイント関数を宣言しています。a_v4_proc_descriptor 変数 *udf_rg_descriptor* へのポインタを返す単純なものです。

```
extern "C"
a_v4_extfn_proc * SQL_CALLBACK udf_rg_1_proc()
/*****************************************/
{
    return &udf_rg_descriptor;
}
```

5. サーバがテーブル UDF からローの情報を取得する方法を定義します。

   テーブル UDF からローのデータを取得する方法をサーバに伝えるための a_v4_extfn_table_func 記述子を次のように宣言しています。

```
static a_v4_extfn_table_func udf_rg_table_funcs =
{
    udf_rg_open,       // _open_extfn
    udf_rg_fetch_into, // _fetch_into_extfn
    NULL,              // _fetch_block_extfn
    NULL,              // _rewind_extfn
    udf_rg_close,      // _close_extfn
    NULL,              // Reserved: must be NULL
    NULL               // Reserved: must be NULL
};
```

   この例では、_fetch_into_extfn 関数を使用してローのデータをサーバに転送します。この方法は、理解と実装が最も簡単なデータ転送方法です。このマニュアルでは、データ転送方法のことを「*ロー・ブロック・データ交換*」と呼びます。ロー・ブロック・データ交換の関数は 2 つあります。_fetch_into_extfn と _fetch_block_extfn です。

   実行時に _evaluate_extfn 関数が呼び出されると、UDF は結果セット・パラメータを設定することでテーブル関数記述子をパブリッシュします。そのために、UDF は a_v4_extfn_table のインスタンスを次のように作成します。

```
static a_v4_extfn_table udf_rg_table = {
    &udf_rg_table_funcs,    // Table function descriptor
    1                    // number_of_columns
};
```

   この構造体は、udf_rg_table_funcs 構造体のポインタと、結果セットのカラム数を保持する構造体です。このテーブル UDF が結果セットで生成するのは単一のカラムです。

6. a_v4_extfn_proc 構造体の関数を実装します。

   この例では、必須の関数 _describe_extfn で何の処理も実行しません。テーブル UDF で describe 関数を使用する方法については他の例で示します。

```
static void UDF_CALLBACK udf_rg_describe(
       a_v4_extfn_proc_context *ctx )
/
*****************************************************************
/
{
   // This required function is not needed in this simple example.
}
```

_evaluate_extfn メソッドでは、UDF から結果セットを取得するための情報をサーバに伝えます。これは、a_v4_extfn_proc_context のメソッド set_value を引数 0 で呼び出すことにより行います。引数 0 は戻り値を表し、テーブル UDF では DT_EXTFN_TABLE です。次のように、メソッドでは、an_extfn_value 構造体を組み立て、データ型は DT_EXTFN_TABLE に設定し、値のポインタは手順 5 で作成した a_v4_extfn_table オブジェクトを指すポインタに設定しています。テーブル UDF では、型は常に DT_EXTFN_TABLE とします。

```
static void UDF_CALLBACK udf_rg_evaluate(
          a_v4_extfn_proc_context *ctx,
        void *args_handle )
/*********************************/
{
    an_extfn_value    result_table = { &udf_rg_table,
                      sizeof( udf_rg_table ),
                      sizeof( udf_rg_table ),
                      DT_EXTFN_TABLE };

    // Tell the server what functions table functions are being
    // implemented and how many columns are in our result set.
    ctx->set_value( args_handle, 0, &result_table );
}
```

7. a_v4_extfn_table_func 構造体の関数を実装します。

   この例では、生成するロー数はテーブル UDF にパラメータで渡されます。テーブル UDF はこの情報を読み込み、後で使用できるようにキャッシュしておく必要があります。パラメータの値が新しくなるごとに _open_extfn メソッドが呼び出されるため、この情報を取得する場所としてはこのメソッドが適切です。

   テーブル UDF では、生成するローの総数に加えて、次にどのローを生成するかについても覚えておきます。サーバは、テーブル UDF からローのフェッチを開始すると、必要に応じて _fetch_into_extfn メソッドを繰り返し呼び出します。したがって、テーブル UDF は、最後に生成したローを覚えておく必要があります。

   udf_rg_1.cxx では、複数の呼び出しにまたがってステータス情報を保持しておくために、次のような構造体を作成しています。

```
struct udf_rg_state {
    a_sql_int32    next_row;  // The next row to produce
    a_sql_int32    max_row;   // The number of rows to generate.
};
```

open メソッドでは、まず a_v4_proc_context のメソッド get_value を使用して、引数 1 の値を読み取っています。そして、a_v4_proc_context の関数 alloc を使用して udf_rg_state のインスタンスを作成しています。テーブル UDF では、a_v4_proc_context 構造体のメモリ管理関数 (alloc および free) を随時使用して、自らのメモリを管理します。作成したステータス・オブジェクトは、a_v4_proc_context の *_user_data* フィールドに保存します。このフィールドに格納したメモリは、実行が完了するまでテーブル UDF が使用できます。

```
static short UDF_CALLBACK udf_rg_open(
          a_v4_extfn_table_context * tctx )
/*************************************/
{
    an_extfn_value     value;
    udf_rg_state *    state = NULL;

    // Read in the value of the input parameter and store it away in a
    // state object.  Save the state object in the context.
    if( !tctx->proc_context->get_value( tctx->args_handle,
                    1,
                    &value ) ) {

    // Send an error to the client if we could not get the value.
    tctx->proc_context->set_error(
        tctx->proc_context,
        17001,
        "Error: Could not get the value of parameter 1" );

    return 0;
    }

    // Allocate memory for the state using the a_v4_extfn_proc_context
    // function alloc.
    state = (udf_rg_state *)
    tctx->proc_context->alloc( tctx->proc_context,
                   sizeof( udf_rg_state ) );

    // Start generating at row zero.
    state->next_row = 0;

    // Save the value of parameter 1
    state->max_row = *(a_sql_int32 *)value.data;

    // Save the state on the context
    tctx->user_data = state;

    return 1;
}
```

_fetch_info_extfn メソッドは、サーバに対してローのデータを返します。このメソッドは false を返すまで繰り返し呼び出されます。この例のテーブル

UDF では、a_v4_extfn_proc_context オブジェクトの *_user_data* フィールドから取得したステータス情報を使用して、次に生成するローと、生成するローの総数を決定しています。このメソッドは、引数のロー・ブロック構造体で示された最大数までのローを生成できます。

この例でテーブル UDF が生成するのは INT 型の単一のカラムです。ステータスに保存してある *next_row* のデータを、先頭カラムのデータ・ポインタにコピーしています。ループの繰り返しごとに、このデータ・ポインタに新しい値をコピーし、生成するローの最大数に達した時点か、またはロー・ブロックが一杯になった時点で繰り返しを終了します。

```
static short UDF_CALLBACK udf_rg_fetch_into(
          a_v4_extfn_table_context *tctx,
          a_v4_extfn_row_block *rb)
/*****************************************/
{
    udf_rg_state    *state = (udf_rg_state *)tctx->user_data;

    // Because we are implementing fetch_into, the server has provided
    // us with a row block. We need to inform the server how many rows
    // this call to _fetch_into has produced.
    rb->num_rows = 0;

    // The server provided row block structure contains a max_rows
    // field.  This field is the maximum number of rows that this row
    // block can handle.  We can not exceed this number.  We will also
    // stop producing rows when we have produced the number of rows
    // required as per the max_row in the state.
    while( rb->num_rows < rb->max_rows && state->next_row < state->max_row ) {

            // Get the current row from the row block data.
            a_v4_extfn_row      &row = rb->row_data[ rb->num_rows ];

            // Get the column data for the current row.
            a_v4_extfn_column_data    &col0 = row.column_data[ 0 ];

            // Copy the integer value for the next row to generate
            // into the column data for the current row.
            memcpy( col0.data, &state->next_row, col0.max_piece_len );

            state->next_row++;
            rb->num_rows++;
    }

    // If we produced any rows, return true.
    return( rb->num_rows > 0 );
}
```

テーブル UDF は、パラメータで渡された新しい値それぞれについて、すべてのローをフェッチした後で _close_extfn メソッドを呼び出します。つまり、_open_extfn の呼び出しそれぞれについて、後で _close_extfn の呼び出しがあります。この例では、テーブル UDF は _open_extfn の呼び出しの中で確保したメモリを解放する必要があります。そのために、a_v4_extfn_proc_context オブジェクトの *_user_data* フィールドからステータスを取得し、free メソッドを呼び出しています。

```
static short UDF_CALLBACK udf_rg_close(
          a_v4_extfn_table_context *tctx)
/************************************/
{
    udf_rg_state *    state = NULL;

    // Retrieve the state that was saved in user_data
    state = (udf_rg_state *)tctx->user_data;

    // Free the memory for the state using the
a_v4_extfn_proc_context
    // function free.
    tctx->proc_context->free( tctx->proc_context, state );
    tctx->user_data = NULL;

    return 1;
}
```

参照：

- udf_rg_2 (125 ページ)
- udf_rg_3 (128 ページ)
- ロー・ブロックのデータ交換 (143 ページ)
- describe の API (227 ページ)
- _evaluate_extfn (315 ページ)
- fetch_into (340 ページ)
- テーブル (a_v4_extfn_table) (337 ページ)
- 外部プロシージャ・コンテキスト (a_v4_extfn_proc_context) (317 ページ)
- _open_extfn (348 ページ)
- _close_extfn (351 ページ)

udf_rg_1.cxx で実装されたサンプルのテーブル UDF の実行

サンプル `udf_rg_1` は、`libv4apiex` というコンパイル済みのダイナミック・ライブラリに含まれています (拡張子はプラットフォームにより異なります)。その実装は、`samples` ディレクトリの `udf_rg_1.cxx` にあります。

1. ライブラリ `libv4apiex` を、サーバが読み取ることのできるディレクトリに配置します。
2. テーブル UDF をサーバに対して宣言するために、次のように指定します。

   ```
   CREATE PROCEDURE udf_rg_1( IN num INT )
   RESULT( c1 INT )
   EXTERNAL NAME 'udf_rg_1@libv4apiex'
   ```

3. テーブル UDF からローを次のように選択します。

   ```
   SELECT * FROM udf_rg_1( 5 );
   ```

udf_rg_2

サンプルのテーブル UDF udf_rg_2 は、udf_rg_1.cxx のサンプルを基にして作成されており、動作は同じです。プロシージャ名は **udf_rg_2** で、実装はサンプル・ディレクトリの udf_rg_2.cxx にあります。

テーブル UDF **udf_rg_2** は、a_v4_extfn_proc 記述子が示す _describe_extfn メソッドの実装に違いがあります。

```
static void UDF_CALLBACK udf_rg_describe(
        a_v4_extfn_proc_context *ctx )
/*************************************************************/
{
    a_sql_int32        desc_rc;

    // The following describes will ensure that the schema defined
    // by the user matches the schema supported by this table udf.
    // This is achieved by telling the server what our schema is
    // using describe_xxxx_set methods.
    if( ctx->current_state == EXTFNAPIV4_STATE_ANNOTATION ) {

    a_sql_data_type        type     = DT_NOTYPE;
    a_sql_uint32        num_cols    = 0;
    a_sql_uint32        num_parms    = 0;


    // Inform the server that we support a single input
    // parameter.
    num_parms = 1;
    desc_rc = ctx->describe_udf_set
        ( ctx,
          EXTFNAPIV4_DESCRIBE_UDF_NUM_PARMS,
          &num_parms,
          sizeof( num_parms ) );

    // Checks the return code and sets an error if the
    // describe was unsuccessful for any reason.
    UDF_CHECK_DESCRIBE( ctx, desc_rc );

    // Inform the server that the type of parameter 1 is int.
    type = DT_INT;
    desc_rc = ctx->describe_parameter_set
        ( ctx,
          1,
          EXTFNAPIV4_DESCRIBE_PARM_TYPE,
          &type,
          sizeof( type ) );

    UDF_CHECK_DESCRIBE( ctx, desc_rc );

    // Inform the server that the number of columns in our
    // result set is 1.
    num_cols = 1;
    desc_rc = ctx->describe_parameter_set
```

```
        ( ctx,
          0,
          EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_COLUMNS,
          &num_cols,
          sizeof( num_cols ) );

    UDF_CHECK_DESCRIBE( ctx, desc_rc );

    // Inform the server that the type of column 1 in our
    // result set is int.
    type = DT_INT;
    desc_rc = ctx->describe_column_set
        ( ctx,
          0,
          1,
          EXTFNAPIV4_DESCRIBE_COL_TYPE,
          &type,
          sizeof( type ) );

    UDF_CHECK_DESCRIBE( ctx, desc_rc );

    }

    // The following describes will inform the server of various
    // optimizer related characteristics.
    if( ctx->current_state == EXTFNAPIV4_STATE_OPTIMIZATION ) {

    an_extfn_value        p1_value;
    a_v4_extfn_estimate    num_rows;

    // If the value of parameter 1 was constant, then we can
    // inform the server how many distinct values will be.
    desc_rc = ctx->describe_parameter_get
        ( ctx,
          1,
          EXTFNAPIV4_DESCRIBE_PARM_CONSTANT_VALUE,
          &p1_value,
          sizeof( p1_value ) );

    UDF_CHECK_DESCRIBE( ctx, desc_rc );

    if( desc_rc != EXTFNAPIV4_DESCRIBE_NOT_AVAILABLE ) {

        // Inform the server that this UDF will produce n rows.
        num_rows.value = *(a_sql_int32 *)p1_value.data;
        num_rows.confidence = 1;
        desc_rc = ctx->describe_parameter_set
        ( ctx,
          0,
          EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_ROWS,
          &num_rows,
          sizeof( num_rows ) );

        UDF_CHECK_DESCRIBE( ctx, desc_rc );

        // Inform the server that this UDF will produce n distinct
```

```
        // values for column 1 of its result set.
        desc_rc = ctx->describe_column_set
        ( ctx,
          0,
          1,
          EXTFNAPIV4_DESCRIBE_COL_DISTINCT_VALUES,
          &num_rows,
          sizeof( num_rows ) );

        UDF_CHECK_DESCRIBE( ctx, desc_rc );

    }

    }
}
```

この `describe` メソッドの主な機能は次の 2 つです。

- サポートしているスキーマをサーバに伝える。
- 既知の最適化の属性についてサーバに伝える。

`describe` 関数はいくつかの状態のときに呼び出されます。しかし、`describe` のすべての属性をすべての状態で使用できるわけではありません。`describe` メソッドでは、`a_v4_extfn_proc` 構造体の *current_state* 変数を確認することで、現在の実行中の状態を判断しています。

注釈状態のときには、テーブル UDF **udf_rg_2** は、この関数が `INTEGER` 型のパラメータ 1 つを取るということと、結果セットは `INTEGER` 型の単一のカラムで構成されるということをサーバに伝えます。これは、次の属性を設定することにより行われます。

- EXTFNAPIV4_DESCRIBE_UDF_NUM_PARMS
- EXTFNAPIV4_DESCRIBE_PARM_TYPE
- EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_COLUMNS
- EXTFNAPIV4_DESCRIBE_COL_TYPE

これらの `describe` メソッドで設定した情報が **CREATE PROCEDURE** 文のプロシージャ定義と一致しない場合、`describe_parameter_set` メソッドと `describe_column_set` メソッドは `EXTFNAPIV4_DESCRIBE_INVALID_ATTRIBUTE_VALUE` を返します。その場合 describe メソッドは、不一致があることをクライアントに示すためのエラーを設定します。

この例では、describe の戻り値の確認と、不一致がある場合のエラーの設定に、マクロ `UDF_CHECK_DESCRIBE` を使用しています。このマクロは `udf_utils.h` で定義されています。

最適化の段階では、テーブル UDF **udf_rg_2** はサーバに対し、パラメータ 1 が示すのと同じ数のローを返すということを伝えます。生成するローはインクリメント

していくため、値は一意でもあります。最適化のときには、定数値を持つパラメータのみを使用できます。定数パラメータの値の取得には、`describe` の属性 `EXTFNAPIV4_DESCRIBE_PARM_CONSTANT_VALUE` を使用します。テーブル UDF **udf_rg_2** は、属性値を使用できることを確認すると、取得した値に `EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_ROWS` と `EXTFNAPIV4_DESCRIBE_COL_DISTINCT_VALUES` を設定します。

参照：

- udf_rg_3 (128 ページ)
- サンプルのテーブル UDF udf_rg_1 の実装 (119 ページ)

udf_rg_2.cxx で実装されたサンプルのテーブル UDF の実行

サンプル `udf_rg_2` は、`libv4apiex` というコンパイル済みのダイナミック・ライブラリに含まれています (拡張子はプラットフォームにより異なります)。その実装は、`samples` ディレクトリの `udf_rg_2.cxx` にあります。

1. テーブル UDF をサーバに対して宣言するために、次のように指定します。

```
CREATE OR REPLACE PROCEDURE udf_rg_2( IN num INT )
RESULT( c1 INT )
EXTERNAL NAME 'udf_rg_2@libv4apiex'
```

2. テーブル UDF からローを次のように選択します。

```
SELECT * FROM udf_rg_2( 5 );
```

3. `describe` による動作の違いを確認するために、テーブル UDF が通知するものとは異なるスキーマの **CREATE PROCEDURE** 文を発行します。例を示します。

```
CREATE OR REPLACE PROCEDURE udf_rg_2( IN num INT, IN extra INT )
RESULT( c1 INT )
EXTERNAL NAME 'udf_rg_2@libv4apiex'
```

4. テーブル UDF からローを次のように選択します。

```
SELECT * FROM udf_rg_2( 5 );
```

   IQ はエラーを返します。

udf_rg_3

サンプルのテーブル UDF **udf_rg_3** は、**udf_rg_2** を基にして作成されており、動作は似ています。プロシージャ名は **udf_rg_3** で、実装は `samples` ディレクトリの `udf_rg_3.cxx` にあります。

テーブル UDF **udf_rg_3** と **udf_rg_2** の動作の違いは、**udf_rg_3** が生成するのは値 0 ～ 99 までのユニークな値 100 個のみで、その数値の並びを必要なだけ繰り返すということです。このテーブル UDF は `_start_extfn` メソッドと

_finish_extfn メソッドを備えています。また、関数の動作の違いに合わせて、_describe_extfn にも修正が加わっています。

fetch_into ではなく fetch_block を使用していることから、このテーブル UDF はロー・ブロック構造体を自ら保持でき、独自のデータ・レイアウトを使用できます。その例として、生成する値をあらかじめ配列に格納しています。フェッチの実行時には、サーバが提供したロー・ブロックにデータをコピーするのではなく、ロー・ブロック・データのポインタをデータの格納先のメモリに直接設定しています。こうすることで、余分なコピーを回避しています。

次に示す補助的な構造体には、数値の配列を格納しています。この構造体は、確保したロー・ブロックのポインタも保持しており、このロー・ブロックを解放します。

```
#define MAX_ROWS 100
struct RowData {

    a_sql_int32            numbers[MAX_ROWS];
    a_sql_uint32           piece_len;
    a_v4_extfn_row_block * rows;

    void Init()
    {
           rows = NULL;
           piece_len = sizeof( a_sql_int32 );
           for( int i = 0; i < MAX_ROWS; i++ ) {
               numbers[i] = i;
           }
    }
};
```

この構造体は、テーブル UDF の実行を開始するときに確保し、実行を終了するときに解放します。それには、a_v4_extfn_proc_context の _start_extfn メソッドと _finish_extfn メソッドを使用します。

```
static void UDF_CALLBACK udf_rg_start(
         a_v4_extfn_proc_context *ctx )
/***********************************/
{
    // The start_extfn method is a good place to allocate our row
    // data.  This method is called only once at the beginning of
    // execution.
    RowData *row_data = (RowData *)
          ctx->alloc( ctx, sizeof( RowData ) );
    row_data->Init();
    ctx->_user_data = row_data;
}
```

finish メソッドには次の 2 つの役割があります。

- RowData 構造体を解放する。

- テーブル UDF でフェッチ時にエラーが発生してロー・ブロックを破棄できなかった場合に、ここでロー・ブロックを破棄する。

```
static void UDF_CALLBACK udf_rg_finish(
          a_v4_extfn_proc_context *ctx )
/*************************************/
{
    if( ctx->_user_data != NULL ) {

           RowData *row_data = (RowData *)ctx->_user_data;


           // If rows is non-null here, it means an error occurred and
           // fetch_block did not complete.
           if( row_data->rows != NULL ) {
               DestroyRowBlock( ctx, row_data->rows, 0, false );
           }

           ctx->free( ctx, ctx->_user_data );
           ctx->_user_data = NULL;
    }
}
```

`fetch_block` メソッドは次のとおりです。

```
static short UDF_CALLBACK udf_rg_fetch_block(
          a_v4_extfn_table_context *tctx,
          a_v4_extfn_row_block **rows )
/*****************************************/
{
    udf_rg_state *    state         = (udf_rg_state*)tctx->user_data;
    RowData *        row_data    = (RowData *)tctx->proc_context->_user_data;

    // First call, we need to build the row block
    if( *rows == NULL ) {

          // This function will build a row block structure that holds
          // MAX_ROWS rows of data.  See udf_utils.cxx for details.
          *rows = BuildRowBlock( tctx->proc_context, 0, MAX_ROWS, false );

          // This pointer gets saved here because in some circumstances
          // when an error occurs, its possible we may have allocated
          // the rowblock structure but then never called back into
          // fetch_block to deallocate it.  In this case, when the finish
          // method is called, we will end up deallocating it there.
          row_data->rows = *rows;
    }

    (*rows)->num_rows = 0;

    // The row block we allocated contains a max_rows member that was
    // set to the macro MAX_ROWS (100 in this case). This field is the
    // maximum number of rows that this row block can handle.  We can
    // not exceed this number.  We will also stop producing rows when
    // we have produced the number of rows required as per the max_row
    // in the state.
    while( (*rows)->num_rows < (*rows)->max_rows &&
             state->next_row < state->max_row ) {

          a_v4_extfn_row      &row = (*rows)->row_data[ (*rows)->num_rows ];
```

```
          a_v4_extfn_column_data    &col0 = row.column_data[ 0 ];

          // Row generation here is a matter of pointing the data
          // pointer in the rowblock to our pre-allocated array of
          // integers that was stored in the proc_context.
          col0.data = &row_data->numbers[(*rows)->num_rows % MAX_ROWS];
          col0.max_piece_len = sizeof( a_sql_int32 );
          col0.piece_len = &row_data->piece_len;
          state->next_row++;
          (*rows)->num_rows++;
    }
    if( (*rows)->num_rows > 0 ) {
          return 1;
    } else {
         // When we are finished generating data, we can destroy the
         // row block structure.
         DestroyRowBlock( tctx->proc_context, *rows, 0, false );
         row_data->rows = NULL;
         return 0;
    }
}
```

このメソッドが最初に呼び出されたときには、ヘルパー関数 **BuildRowBlock** を使用してロー・ブロックを確保しています。この関数は `udf_utils.cxx` にあります。後で使用するために、このロー・ブロックのポインタは `RowData` 構造体に保存しています。

ローの生成は、カラム・データのポインタを設定することにより行っています。設定するアドレスは、事前に確保した数値配列の並びで次に来る数値のアドレスです。カラム・データの `piece_len` ポインタも初期化する必要があり、`RowData` の `piece_len` メンバのアドレスに設定しています。ローのデータ長は固定なので、この値はすべてのローで同じです。

フェッチが最後に呼び出され、生成するデータがないときには、`udf_utils.cxx` にあるヘルパー関数 **DestroyRowBlock** を使用してロー・ブロック構造体を破棄しています。

このテーブル UDF が生成するユニークな値は 100 個のみであるという点に対応するために、`EXTFNAPIV4_DESCRIBE_COL_DISTINCT_VALUES` は値 100 に設定します。次のコードは、describe メソッドからその部分を抜粋したものです。

```
static void UDF_CALLBACK udf_rg_describe(
          a_v4_extfn_proc_context *ctx )
/*************************************************************/
{
...
...
...

    a_v4_extfn_estimate distinct = {
          MAX_ROWS, 1.0
    };

    // Inform the server that this UDF will produce MAX_ROWS
```

```
    // distinct values for column 1 of its result set.
    desc_rc = ctx->describe_column_set
          ( ctx,
            0,
            1,
            EXTFNAPIV4_DESCRIBE_COL_DISTINCT_VALUES,
            &distinct,
            sizeof( distinct ) );

    UDF_CHECK_DESCRIBE( ctx, desc_rc );
...
...
...

}
```

参照：

- udf_rg_2 (125 ページ)
- サンプルのテーブル UDF udf_rg_1 の実装 (119 ページ)

udf_rg_3.cxx で実装されたサンプルのテーブル UDF の実行

サンプル `udf_rg_3` は、`libv4apiex` というコンパイル済みのダイナミック・ライブラリに含まれています (拡張子はプラットフォームにより異なります)。その実装は、`samples` ディレクトリの `udf_rg_3.cxx` にあります。

1. テーブル UDF をサーバに対して宣言するために、次のように指定します。

   ```
   CREATE OR REPLACE PROCEDURE udf_rg_3( IN num INT )
   RESULT( c1 INT )
   EXTERNAL NAME 'udf_rg_3@libv4apiex'
   ```

2. テーブル UDF からローを次のように選択します。

   ```
   SELECT * FROM udf_rg_3( 200 );
   ```

   このクエリで `c1` として生成されるのは、0 ～ 99 の並びの後に再び 0 ～ 99 が並んだ値です。

apache_log_reader

サンプルのテーブル UDF `apache_log_reader` は、Apache のアクセス・ログまたは Apache のエラー・ログの内容をテーブル・データに読み込むものです。`samples` ディレクトリのファイル `apache_log_reader.cxx` で実装されています。

`samples` ディレクトリには、サンプルのアクセス・ログ (`apache_access.log`) とサンプルのエラー・ログ (`apache_error.log`) があります。

サンプル `apache_log_reader` の `_open_extfn` メソッドではログ・ファイルを開きます。`_fetch_into_extfn` メソッドではデータを読み込んで解析し、プロ

シージャがサポートするスキーマに変えます。そして、_close_extfn メソッドでログ・ファイルを閉じます。

参照：
- _open_extfn (348 ページ)
- _fetch_into_extfn (349 ページ)
- _close_extfn (351 ページ)

apache_log_reader.cxx で実装されたサンプルのテーブル UDF の実行

サンプル apache_log_reader は、libv4apiex というコンパイル済みのダイナミック・ライブラリに含まれています (拡張子はプラットフォームにより異なります)。その実装は、samples ディレクトリの apache_log_reader.cxx にあります。

1. テーブル UDF をサーバに対して宣言するために、次のように指定します。
```
create procedure apache_log_reader
(
    in file_name  varchar(4000),
    in log_format varchar(32),
    in ip_padding varchar(1)
)
result
(
    ip_address  varchar(15),
    log_name    varchar(4000),
    user_name   varchar(4000),
    access_time datetime,
    time_zone   int,
    request     varchar(4000),
    response    int,
    bytes_sent  int,
    referer     varchar(4000),
    browser     varchar(4000),
    error_type  varchar(4000),
    error_msg   varchar(4000)
)
external name 'apache_log_reader@libv4apiex'
```
2. テーブル UDF からローを次のように選択します。SQL クエリを実行するときに、アクセス・ログのフル・パスを指定します。
```
SELECT * FROM apache_log_reader( 'apache_access.log', 'access',
null );
```

udf_blob

サンプルのテーブル UDF udf_blob は、blob API を使用してテーブル UDF または TPF で LOB の入力パラメータを読み込む方法を示しています。

udf_blob は、最初の入力パラメータの中で特定の文字の出現数を数えます。パラメータ 1 で使用できるデータ型は LONG VARCHAR または VARCHAR(64) です。

型が LONG VARCHAR の場合は、テーブル UDF は blob API を使用して値を取得します。型が VARCHAR(64) の場合は、get_value を使用して値全体を取得できます。

次のコードは _open_extfn メソッドからの抜粋です。パラメータ **1** を blob API で読み込む方法を示しています。

```
static short UDF_CALLBACK udf_blob_open(
    a_v4_extfn_table_context * tctx )
/*************************************/
{
…
…
    a_v4_extfn_blob     *blob         = NULL;

    ret = tctx->proc_context->get_value( tctx->args_handle, 2,
&value );
    UDF_SQLERROR_RT( tctx->proc_context,
             "get_value for argument 2 failed",
             ret == 1,
             0 );

    letter_to_find = *(char *)value.data;

    ret = tctx->proc_context->get_value( tctx->args_handle, 1,
&value );
    UDF_SQLERROR_RT( tctx->proc_context,
             "get_value for argument 1 failed",
             ret == 1,
             0 );

    if( EXTFN_IS_NULL(value) || EXTFN_IS_EMPTY(value) ) {
    state->return_value = 0;
    return 1;
    }

    if( EXTFN_IS_INCOMPLETE(value) ) {
    // If the value is incomplete, then that means we
// are dealing with a blob.
    tctx->proc_context->get_blob( tctx->args_handle, 1, &blob );
    return_value = ProcessBlob( tctx->proc_context,
 blob,
 letter_to_find );
    blob->release( blob );
    } else {
    // The entire value was put into the value pointer.
    return_value = CountNum( (char *)value.data,
value.piece_len,
letter_to_find );
    }
…
…
}
```

パラメータ 1 を get_value で取得しています。値が空または NULL の場合は、それ以上の処理は不要です。マクロ EXTFN_IS_INCOMPLETE により値が blob と判別された場合は、a_v4_extfn_proc_context の get_blob メソッドを使用して a_v4_extfn_blob のインスタンスを取得しています。ProcessBlob メソッドでは、blob を読み込んで、指定した文字の出現数を判別しています。

参照：

- BLOB (a_v4_extfn_blob) (219 ページ)
- _open_extfn (348 ページ)
- get_blob (330 ページ)
- 外部プロシージャ・コンテキスト (a_v4_extfn_proc_context) (317 ページ)

サンプルのテーブル UDF udf_blob.cxx の実行

サンプル udf_blob は、libv4apiex というコンパイル済みのダイナミック・ライブラリに含まれています (拡張子はプラットフォームにより異なります)。その実装は、samples ディレクトリの udf_blob.cxx にあります。

1. テーブル UDF をサーバに対して宣言するために、次のように指定します。

```
CREATE PROCEDURE udf_blob( IN data long varchar, letter char(1) )
RESULT ( c1 BIGINT )
EXTERNAL NAME 'udf_blob@libv4apiex'
```

2. テーブル UDF からローを次のように選択します。

```
set temporary option Enable_LOB_Variables = 'On';
create variable testblob long varchar;
set testblob = 'aaaaaaaaaabbbbbbbbbbb';
select * from udf_blob(testblob, 'a');
```

   指定した文字列には、文字 "a" が 10 回出現します。

クエリ処理の状態

UDF を参照する SQL 文は、Sybase IQ サーバの中でクエリ処理の複数の状態を遷移します。それぞれの状態において、サーバは v4 API を使用して UDF とやり取りします。

参照：

- describe_column の一般的なエラー (353 ページ)
- EXTFNAPIV4_DESCRIBE_COL_TYPE (set) (249 ページ)
- EXTFNAPIV4_DESCRIBE_COL_TYPE (get) (231 ページ)

初期状態

サーバでの初期状態です。初期状態の間に呼び出される UDF メソッドは _start_extfn のみです。

サーバは、作成した UDF の各インスタンスについて start メソッドを呼び出します。クエリを単一のサーバ・スレッドで実行する場合には、start メソッドの呼び出しは 1 回です。クエリを複数のスレッドで処理する場合や、複数のノードに分散する場合には、サーバは個別の UDF インスタンスを作成するため、start メソッドの呼び出しは複数回です。

UDF では、関数のインスタンス・レベルのデータを、start メソッドの引数となっている a_v4_extfn_proc_context 構造体の _user_data フィールドに設定できます。

注釈状態

注釈状態では、サーバは効率よく正確にクエリを最適化するために必要なメタデータを使用して解析ツリーを更新します。

[_enter_state] メソッド、_describe_extfn メソッド、[_leave_state] メソッドが呼び出されます。_enter_state メソッドと _leave_state メソッドはオプションです。UDF で指定している場合に呼び出されます。

注釈状態は、v4 API では次のように a_v4_extfn_state 列挙型の EXTFNAPIV4_STATE_ANNOTATION で表されます。

```
typedef enum a_v4_extfn_state {
    … EXTFNAPIV4_STATE_ANNOTATION, …
} a_v4_extfn_state;
```

UDF では、このフェーズでスキーマの最初のネゴシエーションを実行できます。スキーマのネゴシエーションは、UDF 側がサポートするスキーマを UDF からサーバに伝えるか、サーバ側での宣言の内容を UDF がサーバに問い合わせるかのいずれかで行います。

UDF からサーバに伝える方法の場合は、サーバが不一致を検出し、クライアントに SQL エラーを返します。たとえば、4 つのパラメータが必要だと UDF が伝えた場合に、SQL での UDF 宣言にパラメータが 2 つしかないときには、サーバがそのことを検出し、クライアントに SQL エラーを返します。

UDF がサーバに宣言の内容を問い合わせ、UDF 自身が検証を実行する方法の場合は、それに応じて実行時の処理を調整します。宣言を一致させるか、set_error v4 API を使用してエラーを返すかのどちらかです。たとえば、入力値として最大 5 個のスカラ整数を受け取り、その合計値を返す UDF を作成するとします。この

UDF では、指定された入力パラメータの数を実行時に判別し、それに応じて内部ロジックを調整します。この場合、SQL アナリストは次のようなプロシージャを作成できます。

```
CREATE PROCEDURE my_sum_2( IN a INT, IN b INT ) EXTERNAL
"my_sum@my_lib"
CREATE PROCEDURE my_sum_3( IN a INT, IN b INT, IN c INT ) EXTERNAL
"my_sum@my_lib"
```

どちらのプロシージャも、基盤として使用している `my_sum` の実装は同じです。`my_sum_2` については、パラメータが 2 つしかないことを UDF が認識し、パラメータ 1 とパラメータ 2 の合計を求めます。`my_sum_3` については、パラメータ 1、2、3 の合計を求めます。

UDF で注釈状態のときに取得できる値は定数のリテラル・パラメータ値のみです。その他の値は実行状態まで取得できません。注釈状態のときにパラメータ値を取得するには、`describe_parameter_get` メソッド、および `PARM_CONSTANT_VALUE` 属性と `PARM_IS_CONSTANT` 属性を使用します。

注釈状態のときに UDF が使用できる `describe` のスキーマの属性は次のとおりです。

- `EXTFNAPIV4_DESCRIBE_UDF_NUM_PARMS`
- `EXTFNAPIV4_DESCRIBE_PARM_NAME`
- `EXTFNAPIV4_DESCRIBE_PARM_TYPE`
- `EXTFNAPIV4_DESCRIBE_PARM_WIDTH`
- `EXTFNAPIV4_DESCRIBE_PARM_SCALE`
- `EXTFNAPIV4_DESCRIBE_PARM_IS_CONSTANT`
- `EXTFNAPIV4_DESCRIBE_PARM_CONSTANT_VALUE`
- `EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_COLUMNS`
- `EXTFNAPIV4_DESCRIBE_COL_NAME`
- `EXTFNAPIV4_DESCRIBE_COL_TYPE`
- `EXTFNAPIV4_DESCRIBE_COL_WIDTH`
- `EXTFNAPIV4_DESCRIBE_COL_SCALE`
- `EXTFNAPIV4_DESCRIBE_COL_IS_CONSTANT`
- `EXTFNAPIV4_DESCRIBE_COL_CONSTANT_VALUE`

注釈のフェーズでは、UDF はスキーマの定義をサーバに伝えるために上記の値を設定できます。UDF がサーバに伝えた内容と SQL のプロシージャ宣言との間に不一致があることを検出したサーバは、エラーを返します。この方法のことを「*自己記述*」といいます。

もう 1 つの方法として、UDF 側で実行できるスキーマ検証があります。スキーマの記述タイプの値を UDF が取得し、不一致を検出した場合にはエラーを設定するという方法です。この方法では、検証が UDF に委ねられますが、UDF は単一の実装で複数のスキーマをサポートできるようになります (たとえば、特定のパラメー

タで複数のデータ型をサポートしたり、可変個のパラメータをサポートしたりできます)。

参照：

- EXTFNAPIV4_DESCRIBE_UDF_NUM_PARMS 属性 (get) (304 ページ)
- EXTFNAPIV4_DESCRIBE_UDF_NUM_PARMS 属性 (set) (306 ページ)
- EXTFNAPIV4_DESCRIBE_PARM_NAME 属性 (get) (266 ページ)
- EXTFNAPIV4_DESCRIBE_PARM_NAME 属性 (set) (287 ページ)
- EXTFNAPIV4_DESCRIBE_PARM_TYPE 属性 (get) (267 ページ)
- EXTFNAPIV4_DESCRIBE_PARM_TYPE 属性 (set) (288 ページ)
- EXTFNAPIV4_DESCRIBE_PARM_WIDTH 属性 (get) (268 ページ)
- EXTFNAPIV4_DESCRIBE_PARM_WIDTH 属性 (set) (289 ページ)
- EXTFNAPIV4_DESCRIBE_PARM_SCALE 属性 (get) (270 ページ)
- EXTFNAPIV4_DESCRIBE_PARM_SCALE 属性 (set) (290 ページ)
- EXTFNAPIV4_DESCRIBE_PARM_IS_CONSTANT 属性 (get) (274 ページ)
- EXTFNAPIV4_DESCRIBE_PARM_IS_CONSTANT 属性 (set) (292 ページ)
- EXTFNAPIV4_DESCRIBE_PARM_CONSTANT_VALUE 属性 (get) (276 ページ)
- EXTFNAPIV4_DESCRIBE_PARM_CONSTANT_VALUE 属性 (set) (293 ページ)
- EXTFNAPIV4_DESCRIBE_COL_NAME (get) (230 ページ)
- EXTFNAPIV4_DESCRIBE_COL_NAME (set) (248 ページ)
- EXTFNAPIV4_DESCRIBE_COL_TYPE (get) (231 ページ)
- EXTFNAPIV4_DESCRIBE_COL_TYPE (set) (249 ページ)
- EXTFNAPIV4_DESCRIBE_COL_WIDTH (get) (231 ページ)
- EXTFNAPIV4_DESCRIBE_COL_WIDTH (set) (250 ページ)
- EXTFNAPIV4_DESCRIBE_COL_SCALE (get) (232 ページ)
- EXTFNAPIV4_DESCRIBE_COL_SCALE (set) (251 ページ)
- EXTFNAPIV4_DESCRIBE_COL_IS_CONSTANT (get) (237 ページ)
- EXTFNAPIV4_DESCRIBE_COL_IS_CONSTANT (set) (256 ページ)
- EXTFNAPIV4_DESCRIBE_COL_CONSTANT_VALUE (get) (238 ページ)
- EXTFNAPIV4_DESCRIBE_COL_CONSTANT_VALUE (set) (256 ページ)

クエリ最適化状態

最適化状態のときには、サーバはクエリ・プランを構築する最初の処理の段階にあります。サーバはスキーマ情報と予備的な統計情報を収集します。

`[_enter_state]` メソッド、`_describe_extfn` メソッド、`[_leave_state]` メソッドが呼び出されます。`_enter_state` メソッドと `_leave_state` メソッドはオプションです。UDF で指定している場合に呼び出されます。

注釈状態は、v4 API では次のように `a_v4_extfn_state` 列挙型の `EXTFNAPIV4_STATE_OPTIMIZATION` で表されます。

```
typedef enum a_v4_extfn_state {
    … EXTFNAPIV4_STATE_OPTIMIZATION, …
} a_v4_extfn_state;
```

クエリ最適化状態の間には次のようなネゴシエーションが実行されます。

- サーバと UDF が、入力テーブルに対して指定済みのパーティション分割／順序／クラスタを判断する。
- サーバと UDF が、入力テーブルに必要なパーティション分割／順序を判断する。
- UDF が結果テーブルの物理プロパティ (順序プロパティなど) を宣言する。
- UDF が、クエリ最適化の処理で使用できるプロパティと統計情報 (推定コストなど) を伝える。
  - テーブル・スコープの推定には以下が含まれる。
    - **ローの数** – 実行状態の間に UDF に存在するローの総数。この値は入力テーブル・パラメータと結果テーブルの両方に対して使用できる。
    - **ローのサイズ** – 各ローの平均バイト数の推定。
  - カラム・スコープの推定には以下が含まれる。
    - **個別カウント** – テーブルのローの総数の中でカラムに含まれる重複しない値の数。この値は入力テーブル・パラメータと結果テーブルの両方に対して使用できる。

最適化状態のときに UDF が使用できる `describe` の属性は次のとおりです。

- `EXTFNAPIV4_DESCRIBE_PARM_NAME`
- `EXTFNAPIV4_DESCRIBE_PARM_TYPE`
- `EXTFNAPIV4_DESCRIBE_PARM_WIDTH`
- `EXTFNAPIV4_DESCRIBE_PARM_SCALE`
- `EXTFNAPIV4_DESCRIBE_PARM_IS_CONSTANT`
- `EXTFNAPIV4_DESCRIBE_PARM_CONSTANT_VALUE`
- `EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_COLUMNS`
- `EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_ROWS`

- EXTFNAPIV4_DESCRIBE_PARM_TABLE_ORDERBY
- EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY
- EXTFNAPIV4_DESCRIBE_PARM_TABLE_REQUEST_REWIND
- EXTFNAPIV4_DESCRIBE_PARM_TABLE_HAS_REWIND
- EXTFNAPIV4_DESCRIBE_COL_NAME
- EXTFNAPIV4_DESCRIBE_COL_TYPE
- EXTFNAPIV4_DESCRIBE_COL_WIDTH
- EXTFNAPIV4_DESCRIBE_COL_SCALE
- EXTFNAPIV4_DESCRIBE_COL_CAN_BE_NULL
- EXTFNAPIV4_DESCRIBE_COL_IS_CONSTANT
- EXTFNAPIV4_DESCRIBE_COL_CONSTANT_VALUE
- EXTFNAPIV4_DESCRIBE_COL_IS_USED_BY_CONSUMER
- EXTFNAPIV4_DESCRIBE_COL_VALUES_SUBSET_OF_INPUT

参照：

- DEFAULT_TABLE_UDF_ROW_COUNT オプション (199 ページ)
- EXTFNAPIV4_DESCRIBE_PARM_NAME 属性 (get) (266 ページ)
- EXTFNAPIV4_DESCRIBE_PARM_NAME 属性 (set) (287 ページ)
- EXTFNAPIV4_DESCRIBE_PARM_TYPE 属性 (get) (267 ページ)
- EXTFNAPIV4_DESCRIBE_PARM_TYPE 属性 (set) (288 ページ)
- EXTFNAPIV4_DESCRIBE_PARM_WIDTH 属性 (get) (268 ページ)
- EXTFNAPIV4_DESCRIBE_PARM_WIDTH 属性 (set) (289 ページ)
- EXTFNAPIV4_DESCRIBE_PARM_SCALE 属性 (get) (270 ページ)
- EXTFNAPIV4_DESCRIBE_PARM_SCALE 属性 (set) (290 ページ)
- EXTFNAPIV4_DESCRIBE_PARM_IS_CONSTANT 属性 (get) (274 ページ)
- EXTFNAPIV4_DESCRIBE_PARM_IS_CONSTANT 属性 (set) (292 ページ)
- EXTFNAPIV4_DESCRIBE_PARM_CONSTANT_VALUE 属性 (get) (276 ページ)
- EXTFNAPIV4_DESCRIBE_PARM_CONSTANT_VALUE 属性 (set) (293 ページ)
- EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_COLUMNS 属性 (get) (277 ページ)
- EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_COLUMNS 属性 (set) (293 ページ)
- EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_ROWS 属性 (get) (278 ページ)
- EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_ROWS 属性 (set) (294 ページ)
- EXTFNAPIV4_DESCRIBE_PARM_TABLE_ORDERBY 属性 (get) (279 ページ)
- EXTFNAPIV4_DESCRIBE_PARM_TABLE_ORDERBY 属性 (set) (295 ページ)
- EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY (get) (280 ページ)

- EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY (set) (297 ページ)
- EXTFNAPIV4_DESCRIBE_PARM_TABLE_REQUEST_REWIND 属性 (get) (282 ページ)
- EXTFNAPIV4_DESCRIBE_PARM_TABLE_REQUEST_REWIND 属性 (Set) (298 ページ)
- EXTFNAPIV4_DESCRIBE_PARM_TABLE_HAS_REWIND 属性 (get) (283 ページ)
- EXTFNAPIV4_DESCRIBE_PARM_TABLE_HAS_REWIND 属性 (set) (300 ページ)
- EXTFNAPIV4_DESCRIBE_COL_NAME (get) (230 ページ)
- EXTFNAPIV4_DESCRIBE_COL_NAME (set) (248 ページ)
- EXTFNAPIV4_DESCRIBE_COL_TYPE (get) (231 ページ)
- EXTFNAPIV4_DESCRIBE_COL_TYPE (set) (249 ページ)
- EXTFNAPIV4_DESCRIBE_COL_WIDTH (get) (231 ページ)
- EXTFNAPIV4_DESCRIBE_COL_WIDTH (set) (250 ページ)
- EXTFNAPIV4_DESCRIBE_COL_SCALE (get) (232 ページ)
- EXTFNAPIV4_DESCRIBE_COL_SCALE (set) (251 ページ)
- EXTFNAPIV4_DESCRIBE_COL_CAN_BE_NULL (get) (233 ページ)
- EXTFNAPIV4_DESCRIBE_COL_CAN_BE_NULL (set) (252 ページ)
- EXTFNAPIV4_DESCRIBE_COL_IS_CONSTANT (get) (237 ページ)
- EXTFNAPIV4_DESCRIBE_COL_IS_CONSTANT (set) (256 ページ)
- EXTFNAPIV4_DESCRIBE_COL_CONSTANT_VALUE (get) (238 ページ)
- EXTFNAPIV4_DESCRIBE_COL_CONSTANT_VALUE (set) (256 ページ)
- EXTFNAPIV4_DESCRIBE_COL_IS_USED_BY_CONSUMER (get) (240 ページ)
- EXTFNAPIV4_DESCRIBE_COL_IS_USED_BY_CONSUMER (set) (257 ページ)
- EXTFNAPIV4_DESCRIBE_COL_VALUES_SUBSET_OF_INPUT (get) (246 ページ)
- EXTFNAPIV4_DESCRIBE_COL_VALUES_SUBSET_OF_INPUT (set) (263 ページ)

プラン構築状態

プラン構築状態では、サーバはクエリ最適化状態の間に検出した最適なプランに基づいてクエリ実行プランを構築します。

`[_enter_state]` メソッド、`_describe_extfn` メソッド、`[_leave_state]` メソッドが呼び出されます。`_enter_state` メソッドと `_leave_state` メソッドはオプションです。UDF で指定している場合に呼び出されます。

プラン構築状態は、v4 API では次のように a_v4_extfn_state 列挙型の EXTFNAPIV4_STATE_PLAN_BUILDING で表されます。

```
typedef enum a_v4_extfn_state {
    … EXTFNAPIV4_STATE_PLAN_BUILDING, …
} a_v4_extfn_state;
```

この段階のクエリ処理では、サーバは UDF から必要なカラムを判断し、必要なカラムについての情報をテーブル・パラメータに要求します。

UDF が並列処理をサポートしていて、並列処理に適したクエリだとサーバが認める場合には、サーバは分散クエリ処理のために UDF のインスタンスを複数作成します。

プラン構築状態では、UDF は describe のすべての属性を使用できます。

たとえば次のコードでは、使用しているカラムの情報をサーバから取得しています。

```
    a_sql_int32     rc;
    rg_udf         *rgUdf = (rg_udf *)ctx->_user_data;
    rg_table       *rgTable = rgUdf->rgTable;
    a_sql_uint32    buffer_size = 0;

    buffer_size = sizeof(a_v4_extfn_column_list)   ( rgTable-
>number_of_columns - 1 ) * sizeof(a_sql_uint32);
    a_v4_extfn_column_list *ulist = (a_v4_extfn_column_list *)ctx-
>alloc(
                                    ctx,
                                    buffer_size );
    memset(ulist, 0, buffer_size);

    rc = ctx->describe_parameter_get( ctx,
                0,
                EXTFNAPIV4_DESCRIBE_PARM_TABLE_UNUSED_COLUMNS,
                ulist,
                buffer_size );

    if( rc != buffer_size ) {
        ctx->free( ctx, ulist );
       UDF_SQLERROR( PC(ctx), "Describe parameter type get failure.",
rc == buffer_size );
    } else {
        rgTable->unused_col_list = ulist;
    }
```

上記のコードは、結果セットに 4 つのカラムを生成するテーブル UDF からの抜粋だと仮定して、SQL 文は次のようなものだとします。

```
SELECT c1, c2 FROM my_table_proc();
```

この場合、describe API が返すのは c1 と c2 のみです。これにより UDF は、結果セットの値の生成を最適化できます。

参照：

- describe の API (227 ページ)

実行状態

実行状態では、サーバは UDF の実行の呼び出しを行います。

実行状態では、プラン構築状態の間に作成した実行プランを使用して、SQL クエリの結果セットを計算します。

呼び出されるメソッドは、`[_enter_state]`、`_describe_extfn`、`evaluate_extfn`、`_open_extfn`、`_fetch_into_extfn`、`_fetch_block_extfn`、`_close_extfn`、`[_leave_state]`、`_finish_extfn` です。

実行状態は、`a_v4_extfn_state` API では次の列挙型で表されます。

```
typedef enum a_v4_extfn_state {
    … EXTFNAPIV4_STATE_EXECUTING, …
} a_v4_extfn_state;
```

実行状態では次のことが可能です。

- 入力テーブル・パラメータのローと、定数ではないスカラ値の入力パラメータを使用できる。
- UDF が入力テーブル・パラメータを基に結果セットを開き、ローをフェッチできる。

パーティション実行状態
入力テーブル・パラメータが存在し、SQL クエリに **PARTITION BY** 句が含まれている場合には、サーバは使用可能なパーティションごとに 1 回ずつ UDF を呼び出します。

ロー・ブロックのデータ交換

ロー・ブロックとは、プロデューサとコンシューマの間のデータ転送領域です。

テーブル UDF の場合は、プロデューサとしてローの生成のみが可能です。既存のロー・ブロックを使用する方法と、独自のロー・ブロックを構築する方法があります。

TPF の場合は、プロデューサとしてローを生成することと、コンシューマとして利用することの両方が可能です。TPF がプロデューサとしてローを生成する方法はテーブル UDF の場合と同じで、既存のロー・ブロックを使用する方法と、独自のロー・ブロックを構築する方法があります。TPF はコンシューマとして入力

テーブルのローを利用でき、プロデューサにロー・ブロックを提供する方法と、独自のロー・ブロックを作成するようプロデューサに要求する方法があります。

参照：

- ロー・ブロック (a_v4_extfn_row_block) (336 ページ)
- テーブル (a_v4_extfn_table) (337 ページ)
- テーブル関数 (a_v4_extfn_table_func) (346 ページ)
- _open_extfn (348 ページ)
- _fetch_into_extfn (349 ページ)
- _fetch_block_extfn (350 ページ)
- _rewind_extfn (350 ページ)
- _close_extfn (351 ページ)

ロー・ブロックのフェッチ・メソッド

ロー・ブロックのフェッチ・メソッドは `_fetch_into_extfn` と `_fetch_block_extfn` です。これらのメソッドは `a_v4_extfn_table_func` 構造体に含まれています。

テーブル UDF または TPF がプロデューサとしてデータを生成する場合、独自のロー・ブロックを構築するときには、`fetch_block` API メソッドを用意する必要があります。UDF で独自のロー・ブロックを構築しないときには、`fetch_into` API メソッドを用意する必要があります。

TPF がコンシューマとしてデータを利用する場合、独自のロー・ブロックを構築するときには、プロデューサの `fetch_into` メソッドを呼び出します。TPF で独自のロー・ブロックを構築しないときには、プロデューサの `fetch_block` メソッドを呼び出す必要があります。

データの生成と利用にどちらのフェッチ・メソッドを使用するかは UDF で選択できます。一般的なガイドラインは次のとおりです。

- **fetch_into –** データ転送領域のメモリをコンシューマが保持し、その領域を使用するようプロデューサに要求する場合には、この API を使用する。この場合、データ転送領域のセットアップはコンシューマが担い、プロデューサはその領域に対して必要なデータ・コピーを実行する。
- **fetch_block –** データ転送領域のフォーマットについてコンシューマが関知しない場合にはこの API を使用する。`fetch_block` では、データ転送領域の作成をプロデューサに要求し、その領域へのポインタを提供する。メモリはコンシューマが保持し、コンシューマはこの領域からデータをコピーする必要がある。

参照：
- テーブル・パラメータ関数 (152 ページ)
- fetch_into (340 ページ)
- fetch_block (342 ページ)

fetch_block メソッド

`fetch_block` メソッドは、データ記憶領域を内在化させる場合に使用します。

`fetch_block` メソッドは、コンシューマがデータについて特定のフォーマットを必要としていない場合にエントリ・ポイントとして使用します。`fetch_block` ではプロデューサに対し、データ転送領域を作成してその領域のポインタを渡すよう要求します。コンシューマはこのメモリを保持し、この領域からデータをコピーする必要があります。

コンシューマが特定のレイアウトを必要としない場合には、`fetch_block` メソッドは `fetch_into` よりも効率的です。`fetch_block` の呼び出しでは、データを格納できるロー・ブロックを指定し、次の `fetch_block` 呼び出しでもそのブロックを渡します。このメソッドは `a_v4_extfn_table_context` 構造体に含まれています。

基盤となるデータ記憶領域がロー・ブロックの構造に簡単にマッピングされない場合でも、UDF は単純に、ロー・ブロックのポインタを自らのメモリ内のアドレスに設定できます。こうすることで、異なるメモリ・レイアウト方式を満たすための不必要なデータ・コピーを回避できます。

この API で使用するデータ転送領域は `a_v4_extfn_row_block` 構造体で定義されています。領域は複数のローのセットとして定義されており、それぞれのローは複数のカラムのセットとして定義されています。ロー・ブロックの作成では、単一のローまたは複数のローのセットを保持するための十分な記憶領域を確保できます。プロデューサがローを格納するときには、ロー・ブロックに確保したローの最大数を超えることはできません。まだローが残っている場合には、そのことをコンシューマに伝えるために、プロデューサは `fetch` メソッドの戻り値として数値 1 を返します。

フェッチを実行する対象はテーブル・オブジェクトです。これは、テーブル UDF の結果セットとして生成されたオブジェクトか、入力テーブル・パラメータの結果セットとして利用するオブジェクトのどちらかです。

参照：
- ロー・ブロックを使用したデータ生成 (146 ページ)
- fetch_block (342 ページ)

fetch_into メソッド

データ転送領域のメモリをコンシューマが保持し、その領域を使用するようプロデューサに要求する場合には、`fetch_into` API を使用します。

`fetch_into` メソッドは、メモリ内でのデータの配置方法をプロデューサが関知していない場合に便利です。このメソッドは、コンシューマが転送領域を特定のフォーマットで保持している場合にエントリ・ポイントとして使用します。`fetch_into()` 関数はフェッチしたローを指定のロー・ブロックに書き込みます。このメソッドは `a_v4_extfn_table_context` 構造体に含まれています。

この API で使用するデータ転送領域は `a_v4_extfn_row_block` 構造体で定義されています。領域は複数のローのセットとして定義されており、それぞれのローは複数のカラムのセットとして定義されています。ロー・ブロックの作成では、単一のローまたは複数のローのセットを保持するための十分な記憶領域を確保できます。プロデューサがローを格納するときには、ロー・ブロックに確保したローの最大数を超えることはできません。まだローが残っている場合には、そのことをコンシューマに伝えるために、プロデューサは fetch メソッドの戻り値として数値 1 を返します。

この API ではオプションとして、コンシューマがロー・ブロックを構築し、自らのデータ構造体を参照するようにデータ・ポインタを設定できます。こうすると、プロデューサがコンシューマ内のメモリにデータを直接格納できます。先にデータ・クレンジングや検証チェックが必要な場合には、コンシューマにとってこの方法が望ましくないこともあります。

フェッチを実行する対象はテーブル・オブジェクトです。これは、テーブル UDF の結果セットとして生成されたオブジェクトか、入力テーブル・パラメータの結果セットとして利用するオブジェクトのどちらかです。

参照：

- ロー・ブロックを使用したデータ生成 (146 ページ)
- fetch_into (340 ページ)

ロー・ブロックを使用したデータ生成

テーブル UDF および TPF では、ロー・ブロック構造体を使用してデータを生成できます。

`a_v4_extfn_row_block` のロー・ブロックには次の 3 つのフィールドがあります。

- **max_rows –** ロー・ブロックがメモリに格納できるテーブル・ローの最大数。

- **num_rows –** 実際に生成された (または使用可能な) ローの数。`max_rows` 以下でなくてはならない。
- **row_data –** 生成された (または使用可能な) ローの配列。それぞれのローは `a_v4_extfn_row` 構造体。

参照：
- テーブル UDF の実装例 (118 ページ)
- fetch_into (340 ページ)
- fetch_block (342 ページ)
- ロー・ブロック (a_v4_extfn_row_block) (336 ページ)
- ロー (a_v4_extfn_row) (335 ページ)
- カラム・データ (a_v4_extfn_column_data) (224 ページ)

fetch_into を使用したデータ生成

`fetch_into` API メソッドを使用したデータ生成について説明します。

1. `num_rows` を、フェッチ呼び出しで生成されたローの数に基づく値に設定します。
2. 生成された各ローについて、`a_v4_extfn_row` の `row_status` フラグを 1 (使用可能) または 0 (使用不可能) に設定します。デフォルト値は 1 です。
3. ロー・セットの各カラム (`a_v4_extfn_column_data`) は次のとおりです。

オプション	説明
is_null	返された値が NULL の場合は true に設定します。デフォルトは false です。
data	返されたデータはこのポインタにコピーする必要があります。
piece_len	返されたデータの実際の長さ。固定長のデータ型については、*max_piece_len* を超えることはできません。固定長のデータ型のデフォルト値は *max_piece_len* です。

4. それぞれのカラムについて、ローが生成されたことを示す場合は 1 を、それ以外の場合は 0 を返します。

fetch_block を使用したデータ生成

`fetch_block` API メソッドを使用したデータ生成について説明します。

1. `max_rows` を、プロデューサが確保したロー・ブロック構造体が保持できるローの最大数に設定します。

2. 最初のフェッチ呼び出しで、max_rows を保持できるロー・ブロック構造体を確保します。
3. num_rows を、フェッチ呼び出しで生成されたローの数に基づく値に設定します。
4. 生成された各ローについて、a_v4_extfn_row の row_status フラグを 1 (使用可能) または 0 (使用不可能) に設定します。デフォルト値は 1 です。
5. ロー・セットの各カラム (a_v4_extfn_column_data) は次のとおりです。

null_value	NULL を示すために使用する値であることを示します。
null_mask	NULL 値のビットを示します。
is_null	値が NULL の場合は、(*(cd->is_null) & cd->null_mask) == cd->null_value) となるように is_null の値を設定します。
data	返すデータが格納されている、プロデューサのメモリ内の領域を指すようにポインタを設定します。
piece_len	返すデータの実際の長さ。固定長のデータ型については、*max_piece_len* を超えることはできません。固定長のデータ型のデフォルト値は *max_piece_len* です。

6. ローが生成されたことを示す場合は 1 を、それ以外の場合は 0 を fetch_into から返します。最後のフェッチ呼び出しで、このロー・ブロック構造体向けに確保したメモリを解放します。

ロー・ブロックの確保

ロー・ブロックの確保は、プロデューサが fetch_block メソッドを使用してデータを生成するとき、またはコンシューマが fetch_into メソッドを使用してデータを取得するときに必要です。

ロー・ブロックの確保と解放の方法を示すサンプル・コードは udf_utils.cxx にあります。

ロー・ブロックを確保するときには、関連する以下のデータ構造体を使用します。これらは extfnapiv4.h ヘッダ・ファイルに含まれています。

```
typedef struct a_v4_extfn_column_data {
    a_sql_byte          *is_null;
    a_sql_byte          null_mask;
    a_sql_byte          null_value;

    void                *data;
    a_sql_uint32        *piece_len;
```

```
    size_t              max_piece_len;

    void                *blob_handle;
} a_v4_extfn_column_data;

typedef struct a_v4_extfn_row {
    a_sql_uint32                *row_status;
    a_v4_extfn_column_data      *column_data;
} a_v4_extfn_row;

typedef struct a_v4_extfn_row_block {
    a_sql_uint32                max_rows;
    a_sql_uint32                num_rows;
    a_v4_extfn_row              *row_data;
} a_v4_extfn_row_block;
```

ロー・ブロックを確保するときには、そのロー・ブロックで保持できるローの数、それぞれのローが持つカラムの数、それぞれのカラムに必要なバイト数を開発者が決定する必要があります。

ロー・ブロックのサイズが *m* で、それぞれのローが持つカラム数が *n* の場合、開発者は、m 個分の `a_v4_extfn_row` 構造体の配列を確保する必要があります。さらに、この配列のそれぞれのローについて、*n* 個分の `a_v4_extfn_column_data` 構造体を確保する必要があります。

次の表は、ロー・ブロック構造体の各メンバを確保するときに必要な作業の概要を示します。

表 2 : a_v4_extfn_row_block 構造体

フィールド	要件
max_rows	このロー・ブロックが保持できるローの数に設定します。
num_rows	0 で初期化します。使用時に、ロー・ブロックが実際に保持するローの数に設定されます。
*row_data	max_rows 個分の `a_v4_extfn_row` 構造体を保持する配列を確保します。

表 3 : a_v4_extfn_row 構造体

フィールド	要件
*row_status	**a_sql_uint32** を保持できるメモリを確保します。
*column_data	`a_v4_extfn_column_data` 構造体の結果セットでのカラム数を保持する配列を確保します。

表 4 : a_v4_extfn_column_data 構造体

フィールド	要件
*is_null	**a_sql_byte** を保持できるメモリを確保します。
null_mask	(*is_null & null_mask) == null_value という式でカラム値が NULL であることを示すように値を設定します。
null_value	(*is_null & null_mask) == null_value という式でカラム値が NULL であることを示すように値を設定します。
*data	カラムのデータ型に応じたデータを保持できるバイト数の配列を確保します。
*piece_len	**a_sql_uint32** を保持できるメモリを確保します。
max_piece_len	カラムの最大幅に設定します。
*blob_handle	常にサーバが所有します。NULL で初期化してください。

参照：

- SQL データ型 (9 ページ)
- 外部プロシージャ・コンテキスト (a_v4_extfn_proc_context) (317 ページ)

テーブル UDF のクエリ・プラン・オブジェクト

クエリ・プランで参照できるテーブル UDF の値と TPF の値について説明します。

- **フェッチしたブロック数** – UDF が生成した全データを転送するために使用したチャンクの数。この値は、UDF のフェッチ・メソッドをサーバが呼び出した回数に等しい。
- **`_fetch_into_extfn` あたりの最大ロー数** – (UDF が `_fetch_into_extfn` を使用している場合にのみ参照可能)。サーバが決定する `_fetch_into_extfn` のそれぞれの呼び出しで UDF が生成できる最大ロー数を示す。
- **出力カラムの最小値／最大値** – UDF が `extfnapiv4_describe_col_maximum_value` で設定している場合に、カラムあたりの最小値／最大値を示す。最小値／最大値が示されるのは算術データ型のカラムのみ。
- **ORDER BY ノード (TPF のみ)** – TPF については、クエリ・プランで ORDER BY ノードは TPF SubQuery ノードの子として示される。ORDER BY ノードは、テーブル・パラメータに入るときにデータが順序付けされていることを示す。
- **出力ロー幅** – (UDF が `_fetch_into_extfn` を使用している場合にのみ参照可能)。出力カラムの幅をバイト単位で示す。ローの最大数の計算で使用する値。

- **テーブル UDF ノード** – クエリ内のテーブル UDF のインスタンスを表す。テーブル UDF ノードはリーフ・ノードである。
- **TPF ノード (TPF のみ)** – テーブル UDF ノードと同様だが、TPF ノードは入力テーブル・パラメータの使用を認めている点が異なる。テーブル UDF ノードがリーフ・ノードであるのに対し、TPF ノードは内部ノードで、最大で 1 つの子を持つ。
- **TPF SubQuery ノード (TPF のみ)** – TPF ノードの子。入力テーブル引数のサブクエリを表す。
- **UDF ライブラリ** – UDF ライブラリのファイル名。UDF を実装するダイナミック・ライブラリの読み込み元をディスク上でのフル・パスで示す。
- **出力カラムがユニークかどうか** – `extfnapiv4_describe_col_is_unique` が設定する値を反映する。
- **TABLE_UDF_ROW_BLOCK_SIZE_KB** – 128KB 以外の値を指定した場合にクエリ・プランの統計情報で示されるオプション値。

メモリ・トラッキングの有効化

メモリ・トラッキングを有効にすると、UDF でのメモリ・リークを見つけやすくなり、リークしたメモリの解放に役立ちます。メモリ・トラッキングを行うとパフォーマンスは低下します。

メモリ・トラッキングを有効にすると、`a_v4_extfn_proc_context alloc` と `a_v4_extfn_proc_context free` のすべての呼び出しが追跡されます。確保したメモリに対する解放がない場合は、`iqmsg` ファイルにログとして記録されます。

1. **external_UDF_execution_mode** が 1 または 2 (検証モードまたはトレース・モード) に設定されていることを確認します。
2. `a_v4_extfn_proc_context` の `alloc` メソッドと `free` メソッドを使用します。

参照：

- alloc (327 ページ)
- free (328 ページ)

テーブル・パラメータ関数

テーブル・パラメータ関数 (TPF) はテーブル UDF を拡張した関数です。入力パラメータとして、スカラ値のほかにテーブルを受け取ることができます。

TPF に対しては、ユーザ指定のパーティション分割を設定できます。UDF 開発者がデータセットのパーティション分割のスキームを宣言し、クエリ処理を細分化してマルチプレックス・ノードで分散できます。これにより、ロー・セットのパーティションに対して分散サーバ環境で TPF を並列実行できます。クエリ・エンジンは TPF 処理の大規模な並列化をサポートしています。

注意： マルチプレックスには別途ライセンスが必要です。『Sybase IQ Multiplex の使用』を参照してください。

TPF 開発者向けのロードマップ

C または C++ で TPF を開発する方法について説明します。

このロードマップの前提は次のとおりです。

- マシンに C または C++ の開発環境がある。
- オプションのデータ・パーティション機能を使用する場合は、マルチプレックス環境がある。『Sybase IQ Multiplex の使用』を参照してください。

タスク	参照先
テーブル UDF の開発について理解します。	テーブル UDF 開発者向けのロードマップ (110 ページ)
推奨の手順に従って TPF を作成します。	TPF の開発 (153 ページ) TPF の実装例 (174 ページ)
入力テーブルのテーブル・コンテキストを確立し、そのテーブル・ローを利用します。	テーブル・パラメータの利用 (153 ページ)
(オプション) 取得したデータを順序付けします。	入力テーブル・データの順序付け (156 ページ)
(オプション) 取得したデータをパーティション分割して、マルチプレックスで TPF を並列処理できるようにします。	入力データのパーティション分割 (157 ページ)

TPF の開発

TPF の開発で必要となる主な手順を再確認します。

1. テーブル UDF の開発で必要な手順と同じ作業を実行する。
2. 入力パラメータを利用する。
3. (オプション) 入力テーブル・データを順序付けする。
4. (オプション) 入力テーブル・データをパーティション分割する。
5. (オプション) TPF の並列処理を有効にする。

参照：

- テーブル・パラメータの利用 (153 ページ)
- 入力テーブル・データの順序付け (156 ページ)
- 入力データのパーティション分割 (157 ページ)
- _open_extfn (348 ページ)
- _fetch_into_extfn (349 ページ)
- _fetch_block_extfn (350 ページ)
- _rewind_extfn (350 ページ)
- _evaluate_extfn (315 ページ)
- テーブル UDF の開発 (115 ページ)

テーブル・パラメータの利用

テーブル・パラメータは定数ではないパラメータです。したがって、TPF がテーブル・パラメータを取得できるのは実行状態のときに限られます。

TPF では、テーブル・パラメータを以下のメソッドから取得できます。

- `_open_extfn`
- `_fetch_into_extfn`
- `_fetch_block_extfn`
- `_rewind_extfn`
- `_evaluate_extfn`

テーブル・パラメータを利用するために TPF で必要な処理を以下に示します。

テーブル・オブジェクトの取得

TPF でテーブル・パラメータのテーブル・オブジェクトを取得するには、`a_v4_extfn_proc_context` の `get_value` メソッドを使用します。

入力テーブルからのローの取得は、テーブル・オブジェクト (`a_v4_extfn_table`) を使用して開始できます。次のコードは、`get_value` で

パラメータ **1** のテーブル・オブジェクトを取得する方法を示しています。わかりやすくするために、このコードはパラメータ **1** がテーブルであることを前提としています。

```
a_v4_extfn_value        value;
a_v4_extfn_table *      table;

ctx->get_value( args_handle,
                        1,
                        &value );

table = (a_v4_extfn_table *)value.data;
```

参照：

- get_value (319 ページ)

結果セットのオープン

`get_value` を使用してテーブル・オブジェクトを取得できた TPF は、`a_v4_extfn_proc_context` の `open_result_set` メソッドを使用して、テーブル・オブジェクトの結果セットをオープンする必要があります。これでローをフェッチできるようになります。

`open_result_set` を呼び出すと `a_v4_extfn_table_context` のインスタンスが返ります。TPF はこれを使用してテーブル・データを処理できます。また、テーブル・オブジェクトは `a_v4_extfn_table_context` オブジェクトの table メンバに保存します。

次のコードは、ローをフェッチするために `open_result_set` で `a_v4_extfn_table_context` のインスタンスを取得する方法を示しています。

```
a_v4_extfn_table_context *  rs  = NULL;

ctx->open_result_set( ctx,
                    (a_v4_extfn_table *)value.data,
                     &rs ) );
```

参照：

- open_result_set (329 ページ)
- テーブル・コンテキスト (a_v4_extfn_table_context) (337 ページ)

結果セットのフェッチ

TPF は、オープンした結果セットを使用して、入力テーブルからテーブル・データをフェッチします。

フェッチを実行するには、`open_result_set` から取得した `a_v4_extfn_table_context` オブジェクトの `fetch_block` または

fetch_into を呼び出します。使用するフェッチ・メソッドは TPF が選択できます。fetch_block を使用する場合は、ロー・ブロックの確保はサーバが行います。fetch_into を使用する場合は、ロー・ブロックの確保は TPF が行います。

フェッチ・メソッドのそれぞれの呼び出しは、何も返らない (戻り値 false) か、データが設定されたロー・ブロック構造体が返るかのどちらかです。このロー・ブロック構造体を使用することで、テーブル・データを利用できます。

参照：
- fetch_into (340 ページ)
- fetch_block (342 ページ)
- ロー・ブロックのデータ交換 (143 ページ)

ロー・ブロックによるテーブル・データの利用

TPF でテーブル・データを利用するには、fetch_into または fetch_block のロー・ブロック構造体を使用します。

fetch_into または fetch_block の呼び出しが成功すると、a_v4_extfn_row_block 構造体にデータが設定されます。

a_v4_extfn_row_block のメンバは次のとおりです。

- **max_rows –** ロー・ブロックがメモリに格納できるテーブル・ローの数。
- **num_rows –** 実際に生成された (または使用可能な) ローの数。max_rows 以下でなくてはならない。
- **row_data –** 生成された (または使用可能な) ローの配列。それぞれのローは a_v4_extfn_row 構造体。

row_data でテーブル・データの各ローのメンバは次のとおりです。

- **row_stats –** このローの値が存在するかどうかを示す。1 の場合は値が存在し、0 の場合は存在しない。
- **column_data –** このローに対応するカラム・データ。

column_data のメンバは次のとおりです。

メンバ	説明
null_value	NULL を表す値。
null_mask	NULL 値を表すために使用する 1 つまたは複数のビット。
data	カラムのデータのポインタ。フェッチ・メカニズムの種類に応じて、コンシューマ内のアドレスか、UDF 内でデータが格納されているアドレスのどちらかを指す。

メンバ	説明
piece_len	可変長データ型のデータの実際の長さ。
blob	NULL 以外の値の場合、このカラムのデータは blob API を使用して読み込む必要があることを表す。

参照：
- カラム・データ (a_v4_extfn_column_data) (224 ページ)
- ロー・ブロック (a_v4_extfn_row_block) (336 ページ)
- ロー (a_v4_extfn_row) (335 ページ)
- get_blob (345 ページ)

結果セットのクローズ

テーブル・データの処理が完了した TPF は、a_v4_extfn_proc_context の close_result_set メソッドを使用して、オープンした結果セットをクローズします。

次のコードは、close_result_set で結果セットをクローズする方法を示します。

```
ctx->close_result_set( ctx,
                       rs ) );
```

入力テーブル・データの順序付け

SQL アナリストまたは UDF 開発者は、取得するデータを順序付けできます。

SQL アナリストが順序付けを制御するには、**SELECT** 文に **ORDER BY** 句を指定します。

UDF 開発者が順序付けを制御するには、**DESCRIBE_PARM_TABLE_ORDERBY** 属性を使用します。

どちらの方法でも、サーバが取得データを順序付けすることになり、その結果はクエリ・プランの Order ノードで確認できます。

参照：
- EXTFNAPIV4_DESCRIBE_PARM_TABLE_ORDERBY 属性 (get) (279 ページ)
- EXTFNAPIV4_DESCRIBE_PARM_TABLE_ORDERBY 属性 (set) (295 ページ)

入力データのパーティション分割

並列処理の TPF での呼び出しのパーティション分割を指定および宣言するには、**PARTITION BY** 句を使用します。

SQL アナリストは、SQL クエリの **PARTITION BY** 句によりサーバ・クエリの並列処理機能と分散機能を活用することで、システム・リソースを効率よく利用できます。指定する句に応じて、サーバは、データを個別の値ベースのロー・セットにパーティション分割するか、またはロー・ベースのセットにパーティション分割します。

- **値ベースのパーティション** – 式のキー値により決定される。この種類のパーティションは、同じ値のすべてのローを参照して集計することを前提とした計算の場合に値を持つ。
- **ローベースのパーティション** – 計算を複数の処理ストリームに分割するための、簡単で効率的な手段。クエリを並列処理で実行する必要がある場合に使用する。

パーティション分割の構成は、TPF に渡す **TABLE** パラメータの **PARTITION BY <expr>** 句で指定できます。UDF 開発者は、呼び出しの前にパーティション分割が必要であることを UDF のコードで宣言するために、テーブル・パラメータのメタデータ属性 **EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY** を使用できます。UDF は、パーティションの適用またはパーティション分割の動的な実施のための確認を実行できます。

参照：

- EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY を使用した並列処理 TPF の PARTITION BY の例 (160 ページ)
- EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY (get) (280 ページ)
- EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY (set) (297 ページ)
- V4 API の describe_parameter と EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY (157 ページ)

V4 API の describe_parameter と EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY

入力テーブル・パラメータのパーティション分割で必要なカラムに関しては、`describe_parameter_set` と `describe_parameter_get` を使用できます。

宣言

`describe_parameter` API には 2 つの宣言があります。

describe_parameter_set 宣言

```
a_sql_int32 (SQL_CALLBACK *describe_parameter_set)(
            a_v4_extfn_proc_context        *cntxt,
            a_sql_uint32                   arg_num,
            a_v4_extfn_describe_parm_type  describe_type,
            void                           *describe_buffer,
            size_t                         describe_buffer_
            )
```

describe_parameter_get 宣言

```
a_sql_int32 (SQL_CALLBACK *describe_parameter_get)(
            a_v4_extfn_proc_context           *cntxt,
            a_sql_uint32                   arg_num,
            a_v4_extfn_describe_parm_type  describe_type,
            const void                     *describe_buffer,
            size_t                            describe_buffer_
            )
```

使用法

これらの API を使用するには、**arg_num** が TABLE パラメータを参照し、**describe_buffer** がメモリ・ブロックの `a_v4_extfn_column_list` 構造体の型を参照している必要があります。

```
typedef struct a_v4_extfn_column_list {
    a_sql_int32         number_of_columns;
    a_sql_uint32        column_indexes[1];
} a_v4_extfn_column_list;
```

構造体の **number_of_columns** フィールドの値は次のいずれかです。

- 正の整数値 N (N は PARTITION BY のリストに存在するカラム数を示す)。
- 0 (**PARTITION BY ANY** を示す)。
- -1 (**NO PARTITION BY** を示す)。

この列挙型はヘッダ・ファイル `extfnapiv4.h` に定義されています。

```
typedef struct a_v4_extfn_column_list {
    a_sql_int32         number_of_columns;
    a_sql_uint32        column_indexes[1];
} a_v4_extfn_column_list;
```

`v4_extfn_partitionby_col_num` 列挙型を使用してカラム・リストの構造体を組み立て、`describe_parameter_set` および `describe_parameter_get` API を実行することによって、その要件をサーバに伝え、パーティション分割の対象の入力カラムを判断できます。`describe_parameter_set` および `describe_parameter_get` API の実行には次のようなシナリオがあります。

describe_parameter_set のシナリオ

カラム・インデックスのシナリオ	説明
{ 1, 1 }	UDF の要求に従って、入力テーブルのカラム #1 がパーティション分割されます。
{ 2, 3, 1 }	UDF の要求に従って、入力テーブルのカラム #3 と #1 がパーティション分割されます。
{ 0 }	UDF の要求に従って、UDF は任意の形式による入力テーブルのパーティション分割をサポートできます。

describe_parameter_get のシナリオ

カラム・インデックスのシナリオ	説明
{ 1, 2 }	入力テーブルのカラム #2 でパーティション分割されています。
{ 2, 1, 2 }	入力テーブルのカラム #1 と #2 でパーティション分割されています。
{ 0 }	入力テーブルはカラム以外のスキームでパーティション分割されています。
NULL	実行時のパーティション分割は提供されていません。

注意： 入力クエリの select リストには、**PARTITION BY ANY** および **PARTITION BY NONE** 以外の **PARTITION BY** 式が指定されている必要があります。

参照：

- describe の API (227 ページ)
- PARTITION BY のカラム番号 (a_v4_extfn_partitionby_col_num) (333 ページ)
- EXTFNAPIV4_DESCRIBE_PARM_TYPE 属性 (get) (267 ページ)
- EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_COLUMNS 属性 (get) (277 ページ)
- EXTFNAPIV4_DESCRIBE_PARM_TYPE 属性 (set) (288 ページ)
- EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_COLUMNS 属性 (set) (293 ページ)
- EXTFNAPIV4_DESCRIBE_COL_TYPE (get) (231 ページ)
- EXTFNAPIV4_DESCRIBE_COL_TYPE (set) (249 ページ)

EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY を使用した並列処理 TPF の PARTITION BY の例

パーティション分割の開発では、TPF 関数に渡す **TABLE** パラメータで **PARTITION BY <expr>** 句を使用します。UDF 開発者は、呼び出しの前にパーティション分割が必要であることを UDF のコードで宣言するために、テーブル・パラメータのメタデータ属性 EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY を使用します。

以下の例で説明する内容は次のとおりです。

- UDF からサーバにパーティション分割の要件を伝えるときの SQL 作成のさまざまなシナリオ
- 各シナリオで有効なクエリと無効なクエリ (SQL 例外)
- サーバが不一致を検出する方法
- SQL 句 **PARTITION BY** と UDF 属性 EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY の使用で生じるさまざまな組み合わせ

参照：

- 入力データのパーティション分割 (157 ページ)

サンプルのプロシージャ定義

TPF の **PARTITION BY** 句のサンプルに対応するプロシージャ定義について説明します。

この項で説明する TPF の **PARTITION BY** 句のサンプルはすべて、次のプロシージャ定義を最初に実行することを前提としています。

```
CREATE PROCEDURE my_tpf( arg1 TABLE( c1 INT, c2 INT ) )
  RESULTS ( r1 INT, r2 INT, r3 INT )
  EXTERNAL 'my_tpf_proc@mylibrary';

CREATE TABLE T( x INT, y INT, z INT );
```

参照：

- describe_parameter_set の例 # 1： カラム 1 での 1 カラム・パーティション分割 (161 ページ)
- describe_parameter_set の例 # 2： 2 カラム・パーティション分割 (163 ページ)
- describe_parameter_set の例 # 3： 任意カラムのパーティション分割 (166 ページ)
- describe_parameter_set の例 # 4： PARTITION BY ANY 句の非サポート (167 ページ)
- describe_parameter_set の例 # 5： パーティション分割の非サポート (169 ページ)

- describe_parameter_set の例 # 6： カラム 2 での 1 カラム・パーティション分割 (171 ページ)

describe_parameter_set の例 # 1： カラム 1 での 1 カラム・パーティション分割
カラム 1 (c1) でパーティション分割を実行するようサーバに伝える UDF の例です。

```
void UDF_CALLBACK my_tpf_proc_describe( a_v4_extfn_proc_context
*ctx )
{
    if( ctx->current_state == EXTFNAPIV4_STATE_ANNOTATION ) {
        a_sql_int32            rc = 0;
        a_v4_extfn_column_list pbcol =
            { 1,        // 1 column in the partition by list
              1 };    // column index 1 requires partitioning

        // Describe partitioning for argument 1 (the table)
        rc = ctx->describe_parameter_set(
ctx,
                  1,
                  EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY,
                  &pbcol,
                  sizeof(pbcol) );

    if( rc == 0 ) {
          ctx->set_error( ctx, 17000,
            “Runtime error, unable set partitioning requirements for
column.” );
      }
   }
}
```

参照：
- サンプルのプロシージャ定義 (160 ページ)
- describe_parameter_set の例 # 2： 2 カラム・パーティション分割 (163 ページ)
- describe_parameter_set の例 # 3： 任意カラムのパーティション分割 (166 ページ)
- describe_parameter_set の例 # 4： PARTITION BY ANY 句の非サポート (167 ページ)
- describe_parameter_set の例 # 5： パーティション分割の非サポート (169 ページ)
- describe_parameter_set の例 # 6： カラム 2 での 1 カラム・パーティション分割 (171 ページ)

カラム 1 の 1 カラム・パーティション分割での SQL 作成のセマンティクス
カラム 1 (c1) での 1 カラム・パーティション分割で有効なクエリの例です。

例 1

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER( PARTITION BY T.x ) )
```

この例では、UDF はデータを最初のカラム (T.x) でパーティション分割することをサーバに伝えており、SQL でも同じカラムでのパーティション分割を明示的に要求しています。両方のカラムが一致する場合、次のようにネゴシエーションされたクエリを使用して、上記の処理はエラーなしで進みます。

```
my_tpf( TABLE( SELECT T.x, T.y FROM T )
     OVER ( PARTITION BY T.x ) )
V4 describe_parameter_get API returns: { 1, 1 }
```

例 2

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER( PARTITION BY ANY ) )
```

この例では、UDF はデータを最初のカラム (T.x) でパーティション分割することをサーバに伝えており、SQL は UDF をパーティション分割で実行することのみをクエリ・エンジンに要求しています。サーバは UDF のパーティション分割の意向をふまえ、結果として例 1 と実質的に同じクエリを実行します。

例 3

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T ) )

SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER ( PARTITION BY DEFAULT ) )
```

この例では、SQL の入力テーブルのクエリで **PARTITION BY** 句や **PARTITION BY DEFAULT** 句を指定していません。この場合、UDF が要求した仕様が適用され、カラム T.x でパーティション分割が実行されます。

カラム 1 の 1 カラム・パーティション分割での SQL 例外
カラム 1 (c1) での 1 カラム・パーティション分割で無効なクエリの例です。 いずれの例でも SQL 例外が発生します。

例 1

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER( PARTITION BY T.y ))
```

この例では、UDF はデータを最初のカラム (T.x) でパーティション分割することをサーバに伝えていますが、SQL は別のカラム (T.y) でのパーティション分割を明示的に要求しています。SQL のこの要求は UDF からの要求と競合します。この結果、サーバは SQL エラーを返します。

例 2

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER( NO PARTITION BY ))
```

この例では、SQL は入力テーブルをパーティション分割しないよう求めており、UDF からの要求と競合します。この結果、サーバは SQL エラーを返します。

例 3

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER( PARTITION BY T.x, T.y ))
```

この例では、UDF はデータを最初のカラム (T.x) でパーティション分割することをサーバに伝えていますが、SQL は複数のカラム (T.x と T.y) でのパーティション分割を明示的に要求しています。SQL のこの要求は UDF からの要求と競合します。この結果、サーバは SQL エラーを返します。

describe_parameter_set の例 # 2 : 2 カラム・パーティション分割
カラム 1 (c1) とカラム 2 (c2) でパーティション分割を実行するようサーバに伝える UDF の例です。

```
void UDF_CALLBACK my_tpf_proc_describe( a_v4_extfn_proc_context
*ctx )
{
    if( ctx->current_state == EXTFNAPIV4_STATE_ANNOTATION ) {
        a_sql_int32            rc = 0;
        a_v4_extfn_column_list pbcol =
{ EXTFNAPIV4_PARTITION_BY_COLUMN_ANY };
```

```
        // Describe partitioning for argument 1 (the table)
        rc = ctx->describe_parameter_set(
ctx,
                  1,
                  EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY,
                  &pbcol,
                  sizeof(pbcol) );

    if( rc == 0 ) {
          ctx->set_error( ctx, 17000,
            “Runtime error, unable set partitioning requirements for
column.” );
       }
   }
}
```

参照：

- サンプルのプロシージャ定義 (160 ページ)
- describe_parameter_set の例 # 1： カラム 1 での 1 カラム・パーティション分割 (161 ページ)
- describe_parameter_set の例 # 3： 任意カラムのパーティション分割 (166 ページ)
- describe_parameter_set の例 # 4： PARTITION BY ANY 句の非サポート (167 ページ)
- describe_parameter_set の例 # 5： パーティション分割の非サポート (169 ページ)
- describe_parameter_set の例 # 6： カラム 2 での 1 カラム・パーティション分割 (171 ページ)

2 カラム・パーティション分割での SQL 作成のセマンティクス

カラム 1 (c1) とカラム 2 (c2) での 2 カラム・パーティション分割で有効なクエリの例です。

例 1

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER( PARTITION BY T.y, T.x ))
```

この例では、UDF はデータをカラム T.y と T.x でパーティション分割することをサーバに伝えており、SQL でも同じカラムでのパーティション分割を明示的に要求しています。両方のカラムが一致する場合、次のようにネゴシエーションされたクエリを使用して、上記の処理はエラーなしで進みます。

```
my_tpf( TABLE( SELECT T.x, T.y FROM T )
     OVER ( PARTITION BY T.y, T.x ) )
V4 describe_parameter_get API returns: { 2, 2, 1 }
```

例 2

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER( PARTITION BY ANY ))
```

この例では、SQL でパーティション分割のカラムを明示的に指定していません。入力テーブルをパーティション分割することのみを指定しています。UDF はカラム T.y と T.x でのパーティション分割を要求しています。この結果、サーバはカラム T.y と T.x で入力データをパーティション分割します。

例 3

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )

SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER ( PARTITION BY DEFAULT ))
```

この例では、SQL で **PARTITION BY** 句や **PARTITION BY DEFAULT** 句を指定していません。サーバは UDF から要求されたパーティション分割を使用します。UDF はカラム T.y と T.x でパーティション分割することを伝えているため、サーバはカラム T.y と T.x でパーティション分割してクエリを実行します。

例 4

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER( PARTITION BY T.x,T.y))
```

この例のセマンティクスは例 1 と同じです。2 つのカラムの順序が変わっていますが、各パーティション内でのカラム T.x と T.y の値は同じままです。カラム (T.x, T.y) でもカラム (T.y, T.x) でも、データの論理パーティション分割は同じになります。

2 カラム・パーティション分割での SQL 例外

カラム 1 (c1) とカラム 2 (c2) での 2 カラム・パーティション分割で無効なクエリの例です。 いずれの例でも SQL 例外が発生します。

例 1

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER( NO PARTITION BY ) )
```

この例では、SQL は入力テーブルをパーティション分割しないよう求めており、UDF からの要求と競合しています。この結果、サーバは SQL エラーを返します。

例 2

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER( PARTITION BY T.x ) )

SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER( PARTITION BY T.y ) )
```

この例では、UDF はデータをカラム T.y と T.x でパーティション分割することをサーバに伝えていますが、SQL ではカラム T.y または T.x のどちらか一方でのパーティション分割を要求しています。SQL のこの要求は UDF からの要求と競合します。この結果、サーバは SQL エラーを返します。

describe_parameter_set の例＃3： 任意カラムのパーティション分割

任意のカラムでパーティション分割を実行できることをサーバに伝える UDF の例です。

```
void UDF_CALLBACK my_tpf_proc_describe( a_v4_extfn_proc_context
*ctx )
{
    if( ctx->current_state == EXTFNAPIV4_STATE_ANNOTATION ) {
        a_sql_int32            rc = 0;
        a_v4_extfn_column_list pbcol =
{ EXTFNAPIV4_PARTITION_BY_COLUMN_ANY };


        // Describe partitioning for argument 1 (the table)
        rc = ctx->describe_parameter_set(
ctx,
                  1,
                  EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY,
                  &pbcol,
                  sizeof(pbcol) );

    if( rc == 0 ) {
          ctx->set_error( ctx, 17000,
            “Runtime error, unable set partitioning requirements for
column.” );
      }
   }
}
```

参照：

- サンプルのプロシージャ定義 (160 ページ)

- describe_parameter_set の例 # 1： カラム 1 での 1 カラム・パーティション分割 (161 ページ)
- describe_parameter_set の例 # 2： 2 カラム・パーティション分割 (163 ページ)
- describe_parameter_set の例 # 4： PARTITION BY ANY 句の非サポート (167 ページ)
- describe_parameter_set の例 # 5： パーティション分割の非サポート (169 ページ)
- describe_parameter_set の例 # 6： カラム 2 での 1 カラム・パーティション分割 (171 ページ)

任意カラムのパーティション分割での SQL 作成のセマンティクス
任意カラムのパーティション分割で有効なクエリの例です。

例 1

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER( PARTITION BY T.x ) )
```

この例では、UDF はデータを最初のカラム (T.x) でパーティション分割することをサーバに伝えており、SQL でも同じカラムでのパーティション分割を明示的に要求しています。両方のカラムが一致する場合、次のようにネゴシエーションされたクエリを使用して、上記の処理はエラーなしで進みます。

```
my_tpf( TABLE( SELECT T.x, T.y FROM T )
     OVER ( PARTITION BY T.y, T.x ) )
V4 describe_parameter_get API returns: { 2, 2, 1 }
```

例 2

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER( PARTITION BY ANY ) )
```

この例では、SQL も UDF もパーティション分割のカラムを明示的に指定していません。SQL では入力テーブルをパーティション分割することのみを指定しています。この結果、サーバは値以外をベースとするスキームでパーティション分割を準備し、データはローの範囲に対してパーティション分割されます。

describe_parameter_set の例 # 4： PARTITION BY ANY 句の非サポート
PARTITION BY ANY 句をサポートしていないため、どのカラムでもパーティション分割を実行できないことをサーバに伝える UDF の例です。

```
void UDF_CALLBACK my_tpf_proc_describe( a_v4_extfn_proc_context
*ctx )
{
    // No describe calls
}
```

参照：

- サンプルのプロシージャ定義 (160 ページ)
- describe_parameter_set の例 # 1： カラム 1 での 1 カラム・パーティション分割 (161 ページ)
- describe_parameter_set の例 # 2： 2 カラム・パーティション分割 (163 ページ)
- describe_parameter_set の例 # 3： 任意カラムのパーティション分割 (166 ページ)
- describe_parameter_set の例 # 5： パーティション分割の非サポート (169 ページ)
- describe_parameter_set の例 # 6： カラム 2 での 1 カラム・パーティション分割 (171 ページ)

PARTITION BY ANY 句をサポートしていない場合の SQL 作成のセマンティクス

UDF が **PARTITION BY ANY** 句をサポートしていない場合に有効なクエリの例です。

例 1

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T ))
```

この例では、SQL で **PARTITION BY** 句を指定していません。サーバは UDF が要求するパーティションを使用しますが、この UDF がパーティション分割の要件をサポートしていないことから、サーバはパーティション分割なしでクエリを実行します。

例 2

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER( NO PARTITION BY ))
```

この例では、SQL で入力テーブル・クエリの指定に **NO PARTITION BY** 句を記述しています。この結果、サーバは実行時のパーティション分割なしでクエリを実行します。

例 3

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER( PARTITION BY T.x))
```

この例では、UDF はパーティション分割の要件を指定していません。一方 SQL では、カラム T.x でのパーティション分割を要求しています。この結果、サーバはカラム T.x でパーティション分割を実行してクエリを実行します。

例 4

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER( PARTITION BY T.y))
```

この例では、UDF はパーティション分割の要件を指定していません。一方 SQL では、カラム T.y でのパーティション分割を要求しています。この結果、サーバはカラム T.y でパーティション分割を実行してクエリを実行します。

例 5

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER( PARTITION BY T.y, T.x))
```

この例では、UDF はパーティション分割の要件を指定していません。一方 SQL では、カラム T.y と T.x でのパーティション分割を要求しています。この結果、サーバはカラム T.y と T.x でパーティション分割を実行してクエリを実行します。

例 6

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER( PARTITION BY ANY ))
```

この例では、SQL は **PARTITION BY ANY** のパーティション分割を要求しています。一方、UDF はパーティション分割の要件をサポートしていません。この結果、サーバはロー範囲のパーティション分割を実行してクエリを実行します。

describe_parameter_set の例 # 5 : パーティション分割の非サポート

パーティション分割をサポートしていないことをサーバに伝える UDF の例です。

```
void UDF_CALLBACK my_tpf_proc_describe( a_v4_extfn_proc_context
*ctx )
{
    if( ctx->current_state == EXTFNAPIV4_STATE_ANNOTATION ) {
        a_sql_int32            rc = 0;
        a_v4_extfn_column_list pbcol =
{ EXTFNAPIV4_PARTITION_BY_COLUMN_NONE };


        // Describe partitioning for argument 1 (the table)
        rc = ctx->describe_parameter_set(
ctx,
                  1,
                  EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY,
                  &pbcol,
                  sizeof(pbcol) );
```

```
    if( rc == 0 ) {
          ctx->set_error( ctx, 17000,
            “Runtime error, unable set partitioning requirements for
column.” );
       }
   }
}
```

参照：

- サンプルのプロシージャ定義 (160 ページ)
- describe_parameter_set の例 # 1： カラム 1 での 1 カラム・パーティション分割 (161 ページ)
- describe_parameter_set の例 # 2： 2 カラム・パーティション分割 (163 ページ)
- describe_parameter_set の例 # 3： 任意カラムのパーティション分割 (166 ページ)
- describe_parameter_set の例 # 4： PARTITION BY ANY 句の非サポート (167 ページ)
- describe_parameter_set の例 # 6： カラム 2 での 1 カラム・パーティション分割 (171 ページ)

パーティション分割をサポートしていない場合の SQL 作成のセマンティクス

UDF がパーティション分割をサポートしていない場合に有効なクエリの例です。

例 1

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER( PARTITION BY ANY )
```

この例では、SQL は **PARTITION BY ANY** のパーティション分割を要求しています。一方、UDF はパーティション分割をサポートしていません。この結果、サーバは実行時のパーティション分割なしでクエリを実行します。

例 2

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )

SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER ( PARTITION BY DEFAULT )
```

この例では、SQL で **PARTITION BY** 句や **PARTITION BY DEFAULT** 句を指定していません。サーバは UDF が要求するパーティションを使用しますが、この UDF がパーティション分割をサポートしていないことから、サーバはパーティション分割なしでクエリを実行します。

例 3

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER( NO PARTITION BY )
```

この例では、SQL でパーティション分割なしと要求しています。この結果、サーバは実行時のパーティション分割なしでクエリを実行します。

パーティション分割をサポートしていない場合の SQL 例外
UDF がパーティション分割をサポートしていない場合に無効なクエリの例です。いずれの例でも SQL 例外が発生します。

例 1

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER( PARTITION BY T.x ))
```

この例では、SQL はカラム T.x でのパーティション分割を要求していますが、UDF はどのカラムでもパーティション分割をサポートしていないため、SQL エラーになります。

例 2

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER( PARTITION BY T.y ))
```

この例では、SQL はカラム T.y でのパーティション分割を要求していますが、UDF はどのカラムでもパーティション分割をサポートしていないため、SQL エラーになります。

例 3

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER( PARTITION BY T.y, T.x ))
```

この例では、SQL はカラム T.y と T.x でのパーティション分割を要求していますが、UDF はどのカラムでもパーティション分割をサポートしていないため、SQL エラーになります。

describe_parameter_set の例 # 6： カラム 2 での 1 カラム・パーティション分割
カラム 2 (c2) でパーティション分割を実行するようサーバに伝える UDF の例です。

```
void UDF_CALLBACK my_tpf_proc_describe( a_v4_extfn_proc_context
*ctx )
```

```
{
    if( ctx->current_state == EXTFNAPIV4_STATE_ANNOTATION ) {
        a_sql_int32            rc = 0;
        a_v4_extfn_column_list pbcol =
            { 1,        // 1 column in the partition by list
              2 };    // column index 2 requires partitioning

        // Describe partitioning for argument 1 (the table)
        rc = ctx->describe_parameter_set(
ctx,
                  1,
                  EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY,
                  &pbcol,
                  sizeof(pbcol) );

    if( rc == 0 ) {
          ctx->set_error( ctx, 17000,
            “Runtime error, unable set partitioning requirements for
column.” );
      }
   }
}
```

参照：

- サンプルのプロシージャ定義 (160 ページ)
- describe_parameter_set の例 # 1： カラム 1 での 1 カラム・パーティション分割 (161 ページ)
- describe_parameter_set の例 # 2： 2 カラム・パーティション分割 (163 ページ)
- describe_parameter_set の例 # 3： 任意カラムのパーティション分割 (166 ページ)
- describe_parameter_set の例 # 4： PARTITION BY ANY 句の非サポート (167 ページ)
- describe_parameter_set の例 # 5： パーティション分割の非サポート (169 ページ)

カラム 2 の 1 カラム・パーティション分割での SQL 作成のセマンティクス

カラム 2 (c2) での 1 カラム・パーティション分割で有効なクエリの例です。

例 1

```
SELECT * FROM my_tpf(
TABLE( SELECT T.x, T.y FROM T )
OVER( PARTITION BY T.y )
```

この例では、UDF はデータを最初のカラム (T.y) でパーティション分割することをサーバに伝えており、SQL でも同じカラムでのパーティション分割を明示的に要求しています。両方のカラムが一致する場合、次のようにネゴシエーションされたクエリを使用して、上記の処理はエラーなしで進みます。

```
my_tpf( TABLE( SELECT T.x, T.y FROM T )
     OVER ( PARTITION BY T.y ) )
V4 describe_parameter_get API returns: { 1, 2 }
```

例 2

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER( PARTITION BY ANY )
```

この例では、SQL でパーティション分割のカラムを明示的に指定していません。入力テーブルをパーティション分割することのみを指定しています。UDF はカラム T.y でのパーティション分割を要求しています。この結果、サーバはカラム T.y で入力データをパーティション分割します。

例 3

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )

SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER ( PARTITION BY DEFAULT )
```

この例では、SQL の入力テーブルのクエリで **PARTITION BY** 句や **PARTITION BY DEFAULT** 句を指定していません。この場合、UDF が要求した仕様が適用され、カラム T.y でパーティション分割が実行されます。

カラム 2 の 1 カラム・パーティション分割での SQL 例外
カラム 2 (c2) での 1 カラム・パーティション分割で無効なクエリの例です。 いずれの例でも SQL 例外が発生します。

例 1

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER( PARTITION BY T.x )
```

この例では、UDF はデータを最初のカラム (T.y) でパーティション分割することをサーバに伝えていますが、SQL は別のカラム (T.x) でのパーティション分割を明示的に要求しています。SQL のこの要求は UDF の要求と競合します。この結果、サーバは SQL エラーを返します。

例 2

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER( NO PARTITION BY )
```

この例では、SQL は入力テーブルをパーティション分割しないよう求めており、UDF からの要求と競合しています。この結果、サーバは SQL エラーを返します。

例 3

```
SELECT * FROM my_tpf(
  TABLE( SELECT T.x, T.y FROM T )
  OVER( PARTITION BY T.x, T.y )
```

この例では、UDF はデータを最初のカラム (T.y) でパーティション分割することをサーバに伝えていますが、SQL は複数のカラム (T.x と T.y) でのパーティション分割を要求しています。SQL のこの要求は UDF の要求と競合します。この結果、サーバは SQL エラーを返します。

TPF の実装例

実装例は、簡単な TPF から始まり、先に進むにつれて複雑さや機能が増していきます。

TPF の実装例は `samples` ディレクトリにあります。

これらの例は、`libv4apiex` というコンパイル済みのダイナミック・ライブラリで使用できます。ライブラリ名の拡張子はプラットフォームによって異なります。このライブラリは `udf_main.cxx` で定義された関数を含んでいます。この中には、`extfn_use_new_api` などのライブラリ・レベル関数が含まれています。`libv4apiex` は、サーバが読み取ることのできるディレクトリに配置してください。

tpf_rg_1

TPF のサンプル `tpf_rg_1.cxx` は、テーブル UDF のサンプル `udf_rg_2.cxx` と似ています。入力パラメータに基づいて複数ローのデータを生成します。

生成されるローの数は、単一の入力テーブルに含まれているローの値の合計です。出力は `udf_rg_2.cxx` と同じです。

このサンプルのコードは大部分が `udf_rg_2.cxx` と同じです。主な違いは次のとおりです。

- 実装する関数名のプレフィクスが `udf_rg` ではなく `tpf_rg` となっている。詳細については `tpf_rg_1.cxx` ファイルを参照。
- この例のスキーマは `_describe_extfn` の実装で検証するが、生成するロー数の推定は行わない。
- `_open_extfn` の実装で、入力テーブルのローを読み込み、生成するロー数を判断する。

_describe_extfn メソッドは、udf_rg_2.cxx とこの例とのスキーマの違いを反映しています。具体的には、パラメータ 1 は整数のカラム 1 つを持つテーブルとなっています。次のコードは _describe_extfn からの抜粋です。

```
static void UDF_CALLBACK tpf_rg_describe(
        a_v4_extfn_proc_context *ctx )
/*************************************/
{
    a_sql_int32        desc_rc;

    // The following describes will ensure that the schema defined
    // by the user matches the schema supported by this TPF
    // This is achieved by telling the server what our schema is
    // using describe_xxxx_set methods.
    if( ctx->current_state == EXTFNAPIV4_STATE_ANNOTATION ) {

        …
…

        // Inform the server that the type of parameter 1 is a TABLE
        type = DT_EXTFN_TABLE;
        desc_rc = ctx->describe_parameter_set
            ( ctx,
                  1,
              EXTFNAPIV4_DESCRIBE_PARM_TYPE,
              &type,
                  sizeof( type ) );

        UDF_CHECK_DESCRIBE( ctx, desc_rc );

        // Inform the server that the input table should have a single
        // column.
        num_cols = 1;
        desc_rc = ctx->describe_parameter_set
            ( ctx,
                  1,
              EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_COLUMNS,
              &num_cols,
                  sizeof( num_cols ) );

        UDF_CHECK_DESCRIBE( ctx, desc_rc );

        // Inform the server that the input table column is an integer
        type = DT_INT;
        desc_rc = ctx->describe_column_set
            ( ctx,
              1,
                  1,
              EXTFNAPIV4_DESCRIBE_COL_TYPE,
              &type,
                  sizeof( type ) );

        UDF_CHECK_DESCRIBE( ctx, desc_rc );

…
```

```
…
    }
}
```

udf_rg_2.cxx では、UDF が生成するローの数は、値が定数の場合には describe のフェーズで取得できます。テーブル引数の場合は定数ではないため、その値は実行状態まで取得できません。このため、describe のフェーズでは、生成するロー数についてのオプティマイザの推定はありません。

この例では、describe の呼び出しが有効なのは注釈状態の間のみです。他の状態での呼び出しでは何も処理を行いません。

_open_extfn メソッドでは、入力テーブルからローを読み込んで値の合計値を計算します。udf_rg_2.cxx の例と同様に、最初の入力パラメータからの値の取得には get_value を使用します。今回の例で異なるのは、パラメータの型が a_v4_extfn_table のポインタであることです。次のコードは _open_extfn からの抜粋です。

```
static short UDF_CALLBACK tpf_rg_open(
    a_v4_extfn_table_context * tctx )
/*************************************/
{
    an_extfn_value             value;
    tpf_rg_state *            state             = NULL;
    a_v4_extfn_table_context *    rs             = NULL;
    a_sql_uint32                num_to_generate    = 0;

    // Read in the value of the input parameter and store it away in a
    // state object.  Save the state object in the context.
    if( !tctx->proc_context->get_value( tctx->args_handle,
                    1,
                    &value ) ) {

        // Send an error to the client if we could not get the value.
        tctx->proc_context->set_error(
            tctx->proc_context,
            17001,
            "Error: Could not get the value of parameter 1" );

        return 0;
    }

    // Open a result set for the input table.
    if( !tctx->proc_context->open_result_set( tctx->proc_context,
                        ( a_v4_extfn_table * )value.data,
                        &rs ) ) {
        // Send an error to the client if we could not open the result
        // set.
        tctx->proc_context->set_error(
            tctx->proc_context,
            17001,
            "Error: Could not open result set on input table." );
```

```
        return 0;
    }

    a_v4_extfn_row_block *          rbfb            = NULL;
    a_v4_extfn_row *                rfb            = NULL;
    a_v4_extfn_column_data *        cdfb            = NULL;
    // When using fetch block to read rows from an input table, the
    // server will manage the row block allocation.
    while( rs->fetch_block( rs, &rbfb ) ) {

        // Each successful call to fetch will fill rows in the server
        // allocated row block.  The number of rows retrieved is
        // indicated by the num_rows member.
        for( unsigned int i = 0; i < rbfb->num_rows; i++ ) {
            rfb = &(rbfb->row_data[i]);
            cdfb = &(rfb->column_data[0]);

                // Only consider non-null values.  To determine null we
                // have to use the following logic.
             if( (*cdfb->is_null & cdfb->null_mask) != cdfb->null_value )
{
            num_to_generate += *(a_sql_int32 *)cdfb->data;
                }
}
    }

   if( !tctx->proc_context->close_result_set( tctx->proc_context, rs ) )
{
        // Send an error to the client if we could not close the
        // result set.
        tctx->proc_context->set_error(
            tctx->proc_context,
            17001,
            "Error: Could not close result set on input table." );

        return 0;
    }

    // Allocate memory for the state using the a_v4_extfn_proc_context
    // function alloc.
    state = (tpf_rg_state *)
    tctx->proc_context->alloc( tctx->proc_context,
                   sizeof( tpf_rg_state ) );
    // Start generating at row zero.
    state->next_row = 0;

    // Save the value of parameter 1
    state->max_row = num_to_generate;

    // Save the state on the context
    tctx->user_data = state;

    return 1;
}
```

get_value でテーブル・オブジェクトを取得したら、open_result_set を呼び出し、テーブルからローのデータを読み込みます。

UDF で入力テーブルからローを読み込むには、fetch_into または fetch_block を使用できます。UDF が入力テーブルからローをフェッチするときには、UDF はデータのコンシューマとなります。コンシューマ (ここでは UDF) がロー・ブロック構造体を自ら管理したい場合には、ロー・ブロック構造体を独自に確保し、fetch_into を使用してデータを取得します。一方、コンシューマがプロデューサ (ここではサーバ) にロー・ブロック構造体の管理を委ねたい場合には、fetch_block を使用します。tpf_rg_1 は後者の方法の例です。

tpf_rg_1 では、オープンした結果セットを使用して、fetch_block を繰り返し呼び出してローのデータをサーバから取得します。fetch_block への呼び出しが成功するごとに、サーバが確保したロー・ブロック構造体に最大 num_rows 個のローのデータが格納されていきます。tpf_rg_1 では、それぞれのローのカラム 1 の値を加算して合計値を求めます。udf_rg_2.cxx の例と同様に、この合計値は a_v4_extfn_proc_context のステータスに保存して後で使用できるようにしています。

参照：

- describe の API (227 ページ)
- _open_extfn (348 ページ)
- テーブル (a_v4_extfn_table) (337 ページ)
- _fetch_block_extfn (350 ページ)

tpf_rg_1 のサンプル TPF の実行

サンプル tpf_rg_1 は、libv4apiex というコンパイル済みのダイナミック・ライブラリに含まれています (拡張子はプラットフォームにより異なります)。その実装は、samples ディレクトリの tpf_rg_1.cxx にあります。

1. サーバに対して TPF を宣言します。

```
CREATE OR REPLACE PROCEDURE tpf_rg_1( IN tab TABLE( num INT ) )
RESULT( c1 INT )
EXTERNAL NAME 'tpf_rg_1@libv4apiex';
```

2. TPF の入力として使用するテーブルを宣言します。

```
CREATE TABLE test_table( val int );
```

3. テーブルにローを挿入します。

```
INSERT INTO test_table values(1);
INSERT INTO test_table values(2);
```

```
INSERT INTO test_table values(3);
COMMIT;
```

4. TPF からローを選択します。
   テーブル test_table は、値が 1、2、3 という 3 つのローを持ちます。これらの値の合計値は 6 です。この例が生成するローの数は 6 となります。

```
SELECT * from tpf_rg_1( TABLE( select val from test_table ) );
```

   a) `describe` による動作の違いを確認するために、TPF が `describe` で伝えるものとは異なるスキーマの **CREATE PROCEDURE** 文を発行します。

```
CREATE OR REPLACE PROCEDURE tpf_rg_1( IN tab TABLE( num INT,
num2 INT ) )
RESULT( c1 INT )
EXTERNAL NAME 'tpf_rg_1@libv4apiex';
```

   b) TPF からローを選択します。

```
// This will return an error that the number of columns in
select list
does not match input table param schema
SELECT * from tpf_rg_1( TABLE( select val from test_table ) );
```

tpf_rg_2

TPF のサンプル `tpf_rg_2.cxx` は、`tpf_rg_1.cxx` のサンプルを基にして作成されており、動作は同じです。入力パラメータに基づいて複数ローのデータを生成します。

このサンプルは、`a_v4_extfn_func` 記述子が示す `_open_extfn` メソッドの実装に違いがあります。動作は `tpf_rg_1` と同じですが、入力テーブルからのローの読み込みに `fetch_block` ではなく `fetch_into` を使用します。

次のコードは `_open_extfn` メソッドからの抜粋です。`fetch_into` を使用して入力テーブルからローを取得しています。

```
static short UDF_CALLBACK tpf_rg_open(
    a_v4_extfn_table_context * tctx )
/*************************************/
{
…
…
…
   // This block of code will create a statically allocated row block
    // that can contain at most 1 row of data.

    a_sql_uint32            c1_data;
    a_sql_byte            c1_null     = 0x0;
    a_sql_uint32            c1_len     = 0;
    a_sql_byte            null_mask    = 0x1;
    a_sql_byte            null_value     = 0x1;
    a_v4_extfn_column_data    cd[1] =
    {
    { &c1_null,        // is_null
      null_mask,        // null_mask
```

```
      null_value,         // null_value
      &c1_data,         // data
      &c1_len,        // piece_len
      sizeof(c1_data),    // max_piece_len
      NULL            // blob
    }
    };

    a_sql_uint32        r_status;

    a_v4_extfn_row        row =
    {
    &r_status, &cd[0]
    };

    a_v4_extfn_row_block    rb =
    {
    1, 0, &row
    };

    // We are providing a row block structure that was statically
    // allocated to have a single row.  This means that each call to
    // fetch_into will return at most 1 row.
    while( rs->fetch_into( rs, &rb ) ) {

// Only consider non-null rows.  They way the column data has
// been defined allows us to treat c1_null as a boolean.
if( !c1_null ) {
    num_to_generate += c1_data;
}

    }

    …
    …
}
```

`fetch_into` を使用して入力テーブルからローを取得する場合には、TPF がロー・ブロック構造体を管理します。この例では、静的なロー・ブロック構造体を作成してローを 1 つずつ取り出しています。代わりに、任意の数のローを同時に扱える動的な構造体を確保するという方法もあります。

このコードでは、入力テーブルのカラムの値を変数 *c1_data* に格納するようにロー・ブロック構造体を定義しています。NULL のローが出現した場合、そのことを示すために、変数 *c1_null* を 1 に設定します。

参照：

- _open_extfn (348 ページ)
- _fetch_into_extfn (349 ページ)

tpf_rg_2 のサンプル TPF の実行

サンプル `tpf_rg_2` は、`libv4apiex` というコンパイル済みのダイナミック・ライブラリに含まれています (拡張子はプラットフォームにより異なります)。その実装は、`samples` ディレクトリの `tpf_rg_2.cxx` にあります。

1. **CREATE PROCEDURE** 文を使用して、サーバに対して TPF を宣言します。

```
CREATE OR REPLACE PROCEDURE tpf_rg_2( IN tab TABLE( num INT ) )
RESULT( c1 INT )
EXTERNAL NAME 'tpf_rg_2@libv4apiex';
```

2. **CREATE TABLE** 文を使用して、TPF の入力として使用するテーブルを宣言します。

```
CREATE TABLE test_table( val INT );
```

3. テーブルにローを挿入します。

```
INSERT INTO test_table VALUES(1);
INSERT INTO test_table VALUES(2);
INSERT INTO test_table VALUES(3);
COMMIT;
```

4. TPF からローを選択します。

```
SELECT * FROM tpf_rg_2( TABLE( SELECT val FROM test_table ) );
```

テーブル test_table は、値が 1、2、3 という 3 つのローを持ちます。これらの値の合計値は 6 です。この例が生成するローの数は 6 となります。

tpf_blob のパススルー TPF

TPF のサンプル `tpf_blob.cxx` は、UDF での LOB と CLOB の高度な処理の例です。このサンプルは `samples` ディレクトリにあります。ここでは、`tpf_blob` のうちで、簡単な例 `tpf_rg_1` と `tpf_rg_2` では説明しなかった概念について説明し、関係する部分のみを取り上げます。

テーブル UDF および TPF では、LOB データや CLOB データは生成できません。しかし、*パススルー*という概念を使用すると、入力テーブルの LOB データや CLOB データを出力テーブルに引き渡すことができます。厳密に言うと、パススルーでは入力テーブルから結果セットに対して任意のデータ型を渡せます。これを使用すると、TPF でローを*フィルタリング*でき、入力テーブルのローのサブセットを出力できます。

`tpf_blob` がサポートする **CREATE PROCEDURE** 文は次のとおりです。

```
CREATE PROCEDURE tpf_blob( IN tab TABLE( num INT, s [LONG] <VARCHAR |
BINARY >,
                           IN pattern char(1) )
RESULT SET ( num INT, s [LONG] <VARCHAR | BINARY > )
EXTERNAL NAME ‘tpf_blob@libv4apiex’
```

このプロシージャは複数のスキーマをサポートしています。入力テーブルと結果セットのカラム **s** で使用できるデータ型は、VARCHAR、BINARY、LONG VARCHAR、または LONG BINARY のいずれかです。

動的スキーマのサポート

tpf_blob プロシージャのスキーマは動的です。

入力テーブルと結果セットのカラム **s** で使用できるデータ型は、VARCHAR、BINARY、LONG VARCHAR、または LONG BINARY のいずれかです。これは、a_v4_extfn_proc_context の describe_column_get メソッドを使用して入力テーブル・カラムのデータ型を取得することによって実現できます。TPF の実装は、スキーマの実際の定義に従って調整しています。プロシージャの *pattern* 引数の解釈は、カラム **s** のデータ型に応じて異なります。文字データ型の場合は、引数は文字として解釈されます。バイナリ・データ型の場合は数字として解釈されます。

参照：

- 外部プロシージャ・コンテキスト (a_v4_extfn_proc_context) (317 ページ)
- *describe_column_get (228 ページ)

入力テーブルの LOB および CLOB カラムの処理

tpf_blob の例で、入力テーブルの各データ・ローでのパターンの出現数を数える方法について説明します。

プロシージャでカラム **s** を LONG VARCHAR または LONG BINARY として定義する場合、そのデータの処理には blob API を使用する必要があります。次のコードは fetch_into_extfn メソッドからの抜粋です。TPF で blob API を使用して入力テーブルの LOB および CLOB データを処理する方法を示しています。

```
if( EXTFN_COL_IS_BLOB(cd, 1) ) {
    ret = state->rs->get_blob( state->rs, &cd[1], &blob );

    UDF_SQLERROR_RT( tctx->proc_context,
             "Failed to get blob",
             (ret == 1 && blob != NULL),
             0 );

    if( state->data_type == DT_VARCHAR ||
        state->data_type == DT_LONGVARCHAR ) {

       num = ProcessBlob( tctx->proc_context, blob, state->pattern );
    } else {
        char i = (char)atoi( &(state->pattern) );
        num = ProcessBlob( tctx->proc_context, blob, i );
    }
    ret = blob->release( blob );
    UDF_SQLERROR_RT( tctx->proc_context,
```

```
            "Failed to release blob",
            (ret == 1),
            0 );
    } else {
    if( state->data_type == DT_VARCHAR ||
        state->data_type == DT_LONGVARCHAR ) {
        num = CountNum( (char *)cd[1].data,
*(cd[1].piece_len),
state>pattern );
    } else {
        char i = (char)atoi( &(state->pattern) );
        num = CountNum( (char *)cd[1].data, *(cd[1].piece_len), i );
    }
    }
```

この TPF では、入力テーブルの各ローについて、BLOB かどうかをマクロ `EXTFN_COL_IS_BLOB` でチェックしています。BLOB の場合は、`a_v4_extfn_table_context` の `get_blob` メソッドを使用して、指定したカラムに対応する BLOB オブジェクトを作成します。`get_blob` メソッドは、処理が成功した場合は `a_v4_extfn_blob` のインスタンスを TPF に渡します。TPF はこのインスタンスを使用して BLOB データを読み込みます。BLOB の処理が完了したら、このインスタンスの `release` を呼び出す必要があります。

`ProcessBlob` メソッドは、BLOB オブジェクトでデータを処理する方法を示します。

```
static a_sql_uint64 ProcessBlob(
    a_v4_extfn_proc_context *ctx,
    a_v4_extfn_blob *blob,
    char pattern)
/*****************************/
{
    char buffer[BLOB_ISTREAM_BUFFER_LEN];
    size_t len = 0;
    short ret = 0;

    a_sql_uint64 num = 0;

    a_v4_extfn_blob_istream *is = NULL;

    ret = blob->open_istream( blob, &is );
    UDF_SQLERROR_RT( ctx,
             "Failed to open blob istream",
             (ret == 1 && is != NULL),
             0 );

    for(;;) {
    len = is->get( is, buffer, BLOB_ISTREAM_BUFFER_LEN );
    if( len == 0 ) {
        break;
    }
    num += CountNum( buffer, len, pattern );
    }
```

```
    ret = blob->close_istream( blob, is );
    UDF_SQLERROR_RT( ctx,
             "Failed to close blob istream",
             (ret == 1),
             0 );
    return num;
}
```

BLOB オブジェクトの `open_istream` メソッドでは `a_v4_extfn_blob_istream` のインスタンスが作成されます。その後で、このインスタンスの `get` メソッドを使用して、BLOB から指定サイズのデータをバッファに読み込むことができます。

参照：

- BLOB 入力ストリーム (a_v4_extfn_blob_istream) (222 ページ)
- BLOB (a_v4_extfn_blob) (219 ページ)
- get_blob (345 ページ)
- fetch_into (340 ページ)

結果セットへの入力テーブル・カラムの引き渡し

`tpf_blob` の例は、TPF が入力テーブルのローを結果テーブルに引き渡す方法と、ローが存在するかどうかを `row_status` フラグで表す方法について示しています。

この方法により、TPF は不要なローをフィルタして除外できます。

1. TPF は、describe のフェーズで、`EXTFNAPIV4_DESCRIBE_COL_VALUES_SUBSET_OF_INPUT` の `describe_column_set` メソッドを使用して、結果セットの特定のローが入力テーブルのローのサブセットであることをサーバに伝える必要があります。次のコードは `describe_extfn` メソッドからの抜粋で、フィルタリングの方法を示しています。

```
else if( ctx->current_state == EXTFNAPIV4_STATE_OPTIMIZATION ) {

    // The output columns of this TPF are the same as the first
    // argument's input table columns.  The following describe
    // informs the consumer of this fact.
    a_v4_extfn_col_subset_of_input colMap;

    for( short i = 1; i <= 2; i++ ) {
        colMap.source_table_parameter_arg_num = 1;
        colMap.source_column_number = i;

        desc_rc = ctx->describe_column_set( ctx,
            0, i,
            EXTFNAPIV4_DESCRIBE_COL_VALUES_SUBSET_OF_INPUT,
```

```
            &colMap, sizeof(a_v4_extfn_col_subset_of_input) );

        UDF_CHECK_DESCRIBE( ctx, desc_rc );
    }
}
```

2. 入力テーブルに対する `fetch_into` の呼び出しで、`fetch_into_extfn` メソッドに渡したのと同じロー・ブロック構造体を渡します。これにより、結果セットのロー・ブロック構造体が入力テーブルのものと同じになります。

参照：

- EXTFNAPIV4_DESCRIBE_COL_VALUES_SUBSET_OF_INPUT (set) (263 ページ)
- fetch_into (340 ページ)
- _fetch_into_extfn (349 ページ)

tpf_blob.cxx のサンプル TPF の実行

サンプル `tpf_blob` は、`libv4apiex` というコンパイル済みのダイナミック・ライブラリに含まれています (拡張子はプラットフォームにより異なります)。その実装は、`samples` ディレクトリの `tpf_blob.cxx` にあります。

1. サーバに対して TPF を宣言します。

```
CREATE OR REPLACE PROCEDURE tpf_blob( IN tab TABLE( num INT,
                                                    s long
varchar ),
                                          IN pattern char(1) )
RESULT( num INT, s long varchar )
EXTERNAL NAME 'tpf_blob@libv4apiex';
```

2. TPF の入力として使用するテーブルを宣言します。

```
CREATE TABLE test_table( val INT, str LONG VARCHAR );
```

3. テーブルにローを挿入します。

```
INSERT INTO test_table VALUES(1, 'aaaaaaaaaabbbbbbbbbb');
INSERT INTO test_table VALUES(2, 'aaaaaaaaaaabbbbbbbbbbb');
INSERT INTO test_table VALUES(3, 'aaaaaaaaaaaabbbbbbbbbbbb');
INSERT INTO test_table VALUES(4, 'aaaaaaaaaaaaabbbbbbbbbbbbb');
INSERT INTO test_table VALUES(5, 'aaaaaaaaaaaaaabbbbbbbbbbbbbb');
COMMIT;
```

4. TPF からローを選択します。

```
SELECT * FROM tpf_blob( TABLE( SELECT val,str FROM test_table ),
'a' );
```

テーブル test_table には、文字 **a** の数が偶数個のローが 3 つあります。ロー 1 は 10 個、ロー 3 は 12 個、ロー 5 は 14 個です。

テーブル UDF と TPF クエリ用 SQL リファレンス

テーブル UDF と TPF を参照するクエリ用の SQL 文のリファレンスです。

ALTER PROCEDURE 文

既存のプロシージャを修正したものに置き換えます。このとき、修正したプロシージャ全体を **ALTER PROCEDURE** 文にインクルードして、プロシージャに対するユーザ・パーミッションを再割り当てします。

構文

構文 1

```
ALTER PROCEDURE [ owner.]procedure-name procedure-definition
```

構文 2

```
ALTER PROCEDURE [ owner.]procedure-name
REPLICATE { ON | OFF }
```

構文 3

```
ALTER PROCEDURE [ owner.]procedure-name
SET HIDDEN
```

構文 4

```
ALTER PROCEDURE [ owner.]procedure-name
RECOMPILE
```

構文 5

```
ALTER PROCEDURE
[ owner.]procedure-name ( [ parameter, …] )
[ RESULT (result-column, ...)]
EXTERNAL NAME 'external-call' [ LANGUAGE environment-name ]
```

パラメータ

- **procedure-definition –** 名前の後に **CREATE PROCEDURE** 構文
- **environment-name – JAVA [DISALLOW | ALLOW SERVER SIDE REQUESTS]**
- **external-call –** [*column-name*:]*function-name@library*, ...

使用法

ALTER PROCEDURE 文には、新しいプロシージャ全体を含めてください。 **PROC** を **PROCEDURE** の同義語として使用できます。

ALTER PROCEDURE 文の構文は、**CREATE PROCEDURE** 文の構文とまったく同じです。

- **構文 1 – ALTER PROCEDURE** 文の構文は、最初の 1 語を除き、**CREATE PROCEDURE** 文の構文とまったく同じです。 Watcom-SQL ダイアレクトと Transact-SQL ダイアレクトのプロシージャは、いずれも **ALTER PROCEDURE** を使用して変更できます。

  ALTER PROCEDURE, では、プロシージャの既存のパーミッションは変更されません。 **DROP PROCEDURE** に続けて **CREATE PROCEDURE** を実行した場合は、実行パーミッションが再割り当てされます。
- **構文 2 –** プロシージャを Sybase Replication Server で他のサイトに移動する必要がある場合は、プロシージャに REPLICATE ON を設定します。
- **構文 3 – SET HIDDEN** を使用して、関連するプロシージャの定義を難読化し、解読できないようにします。 このプロシージャはアンロードして、他のデータベースに再ロードできます。

  注意： この設定は、元に戻せません。 元のプロシージャ定義をデータベースの外部に保持することを強くおすすめします。

  SET HIDDEN を使用すると、デバッガを使用したデバッグでもプロシージャ定義は表示されず、プロシージャ・プロファイリングでもプロシージャ定義は使用できません。

構文 1 と構文 2 を組み合わせることはできません。

- **構文 4 – RECOMPILE** 構文を使用して、ストアド・プロシージャを再コンパイルします。 ストアド・プロシージャを再コンパイルすると、カタログに格納された定義が再解析され、構文が検証されます。 結果セットを生成するものの **RESULT** 句を含まないプロシージャの場合は、データベース・サーバがプロシージャの結果セットの特性を判別しようとし、カタログに情報を格納します。 これは、プロシージャの作成後に、プロシージャの参照先テーブルがカラムの追加、削除、または名前変更するように変更されている場合に役立つことがあります。

  プロシージャの定義は、再コンパイルしても変わりません。**SET HIDDEN** 句で隠された定義を持つプロシージャは再コンパイルできますが、これらの定義は隠されたままになります。
- **構文 5 –** この構文は Java UDF に使用します。

  iq-environment-name: JAVA [DISALLOW | ALLOW SERVER SIDE REQUESTS]:

  DISALLOW がデフォルトです。

  ALLOW はサーバ側接続を許可することを示します。

注意： **ALLOW** は、必要な場合以外は指定しないでください。**ALLOW** を設定すると、特定の種類の Sybase IQ テーブル・ジョインの速度が低下します。

UDF を使用するときに **ALLOW SERVER SIDE REQUESTS** と **DISALLOW SERVER SIDE REQUESTS** の両方を同じクエリの中で指定しないでください。

テーブル UDF に対して **ALTER PROCEDURE** 文を使用するときには、**CREATE PROCEDURE** 文 (外部プロシージャ) の場合と同じ制限が適用されます。

標準

- SQL—ISO/ANSI SQL 文法のベンダ拡張。
- Sybase—Adaptive Server Enterprise ではサポートされていません。

パーミッション

プロシージャの所有者であるか、DBA 権限を持っている必要があります。オートコミット。

参照：
- テーブル UDF の制限 (111 ページ)
- CREATE PROCEDURE 文 (テーブル UDF) (188 ページ)

CREATE PROCEDURE 文 (テーブル UDF)

外部のテーブル・ユーザ定義関数 (テーブル UDF) に対するインタフェースを作成します。テーブル UDF を使用するための正規のライセンスを取得している必要があります。

テーブル UDF の定義には `a_v4_extfn` API を使用します。`a_v3_extfn` API および `a_v4_extfn` API を使用しない外部プロシージャ向けの **CREATE PROCEDURE** 文のリファレンス情報については、別のトピックを参照してください。Java UDF 向けの **CREATE PROCEDURE** 文のリファレンス情報については、別のトピックを参照してください。

構文

```
CREATE [ OR REPLACE ] PROCEDURE
[ owner.]procedure-name ( [ parameter[, …]] )
| RESULT  result-column [, …] )
[ SQL SECURITY { INVOKER | DEFINER } ]
EXTERNAL NAME 'external-call'
```

パラメータ

- **parameter：** – [**IN**] *parameter-namedata-type* [**DEFAULT***expression*]
  | [**IN**] *parameter-nametable-type*
- **table-type：** – TABLE(*column-namedata-type* [, ...])
- **external-call：** – [*column-name*:]*function-name@library*; ...

使用法

CREATE PROCEDURE 文はデータベースにプロシージャを作成します。 DBA 権限を持つユーザは、owner を指定することにより他のユーザのプロシージャを作成できます。

ストアド・プロシージャが結果セットを返す場合は、あわせて出力パラメータを設定したり戻り値を返したりすることはできません。

複数のプロシージャからテンポラリ・テーブルを参照しているときに、テンポラリ・テーブルの定義が矛盾していて、テーブルを参照している文がキャッシュされている場合、問題が発生する可能性があります。 プロシージャ内でテンポラリ・テーブルを参照するときは注意してください。

CREATE PROCEDURE 文を使用して、SQL とは異なるプログラミング言語で実装された外部のテーブル UDF を作成できます。 ただし、外部 UDF を作成する前に、テーブル UDF の制限について注意してください。

スカラ・パラメータ、結果カラム、および TABLE パラメータのカラムのデータ型は、有効な SQL データ型であることが必要です。

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。 これらは有効な SQL データ型である必要があります。

TPF は、複数のスカラ・パラメータの混在と、単一のテーブル・パラメータをサポートしています。TABLE パラメータでは、UDF が処理する入力ロー・セットのスキーマを定義する必要があります。 TABLE パラメータの定義にはカラム名とカラムのデータ型が含まれます。

```
TABLE(c1 INT, c2 CHAR(20))
```

上の例では、INT 型と CHAR(20) 型の 2 つのカラム c1 と c2 を使用してスキーマを定義しています。 UDF が処理する各ローは 2 つの値の組でなくてはなりません。スカラ・パラメータと違い、テーブル・パラメータにはデフォルト値を割り当てることはできません。

- **IN キーワード** – パラメータには、プレフィクスとしてキーワード IN を付けることができます。

- IN—このパラメータは、スカラ・パラメータの値、または TABLE パラメータの複数の値のセットを UDF に与えるオブジェクトです。

注意： テーブル・パラメータを INOUT または OUT として宣言することはできません。テーブル・パラメータは 1 つのみ使用できます (位置は重要ではありません)。

- **OR REPLACE – OR REPLACE (CREATE OR REPLACE PROCEDURE)** を指定すると、新しいプロシージャが作成されるか、同じ名前の既存のプロシージャが置き換えられます。 この句によって、プロシージャの定義は変更されますが、既存のパーミッションは維持されます。置き換え対象のプロシージャが使用中の場合は、エラーが返されます。
- **RESULT –** 外部 UDF の結果セットのカラムの名前とデータ型を宣言します。カラムのデータ型は有効な SQL データ型である必要があります (たとえば、結果セットのカラムではデータ型として TABLE を使用できません)。結果に含まれる複数のデータ項目は暗黙的に TABLE を意味します。 外部 UDF では型 TABLE の結果セットを 1 つのみ使用できます。

注意： TABLE は出力値ではありません。テーブル UDF では結果セットに LONG VARBINARY および LONG VARCHAR データ型は使用できませんが、テーブル・パラメータ関数 (TPF) では結果セットにラージ・オブジェクト (LOB) データを使用できます。

TPF では、LOB データを生成することはできませんが、結果セットに LOB データ型のカラムを持つことはできます。ただし、出力に LOB データを渡すには、入力テーブルのカラムを出力テーブルに渡す方法しかありません。これは describe の属性 `EXTFNAPIV4_DESCRIBE_COL_VALUES_SUBSET_OF_INPUT` を使用することで実現でき、その例はサンプル・ファイル `tpf_blob.cxx` にあります。

プロシージャから返される結果セットの詳細については、 『システム管理ガイド：第 2 巻』の「プロシージャとバッチの使用」を参照してください。

- **SQL SECURITY –** INVOKER (UDF を呼び出しているユーザ) または DEFINER (UDF を所有しているユーザ) のどちらとしてプロシージャが実行されるかを定義します。 デフォルトは DEFINER です。

SQL SECURITY INVOKER が指定されていると、プロシージャを呼び出す各ユーザに対して注釈を付ける必要があるため、必要なメモリも多くなります。また、SQL SECURITY INVOKER が指定されていると、名前解決も INVOKER として実行されます。 したがって、適切な所有者を使用して、すべてのオブジェクト名 (テーブルやプロシージャなど) の条件を満たすように注意してください。 たとえば、user1 が次のプロシージャを作成したとします。

```
CREATE PROCEDURE user1.myProcedure()
    RESULT( columnA INT )
    SQL SECURITY INVOKER
    BEGIN
        SELECT columnA FROM table1;
    END;
```

user2 がこのプロシージャを実行しようとして、user2.table1 テーブルが存在しない場合、テーブルのルックアップでエラーが発生します。 さらに、user2.table1 が存在する場合は、意図した user1.table1 ではなくそのテーブルが使用されます。 この状況を防ぐため、文ではテーブル参照を修飾してください (table1 だけでなく user1.table1 とします)。

- **EXTERNAL NAME** – EXTERNAL NAME *'external-call'*: *external-call*: [*operating-system*:]*function-name@library*; ....

  外部 UDF には、C などのプログラミング言語で記述された関数へのインタフェースを定義する **EXTERNAL NAME** 句が必要です。この関数は、データベース・サーバによってそのアドレス空間にロードされます。

  ライブラリ名にはファイル拡張子を付けることができます。この拡張子は通常、Windows では .dll、UNIX では .so です。 拡張子が付いていなければ、ソフトウェアがプラットフォーム固有のデフォルトのファイル拡張子をライブラリに付加します。 正式に記述した例を次に示します。

```
CREATE PROCEDURE mystring( IN instr CHAR(255),
    IN input_table TABLE(A INT) )
    RESULT (CHAR(255))
EXTERNAL NAME
'mystring@mylib.dll;Unix:mystring@mylib.so'
```

  プラットフォーム固有のデフォルトを使用して、前述の EXTERNAL NAME 句をより簡単に記述する方法を次に示します。

```
CREATE PROCEDURE mystring( IN instr CHAR(255),
    IN input_table TABLE(A INT) )
    RESULT (CHAR(255))
EXTERNAL NAME 'mystring@mylib'
```

標準

- SQL—ISO/ANSI SQL 準拠。
- Sybase—Transact-SQL の **CREATE PROCEDURE** 文は異なります。
- SQLJ—Java 結果セットの構文拡張は、推奨される SQLJ1 規格に指定されています。

パーミッション

テンポラリ・プロシージャを作成する場合を除き、RESOURCE 権限を持っている必要があります。 DBA 権限を持つユーザは、owner を指定することにより他の

ユーザの UDF を作成できます。 外部 UDF を作成するか、または他のユーザ向けの外部 UDF を作成するには、DBA 権限が必要です。

参照：

- サンプル・ファイル (112 ページ)

CREATE FUNCTION 文

新しい関数をデータベースに作成します。

構文

構文 1

```
CREATE [ OR REPLACE ] [ TEMPORARY ] FUNCTION [ owner.]function-name
( [ parameter, … ] )
 RETURNS data-type routine-characteristics
 [ SQL SECURITY { INVOKER | DEFINER } ]
 { compound-statement
  | AS tsql-compound-statement
  | external-name }
```

構文 2

```
CREATE FUNCTION [ owner.]function-name ( [ parameter, … ] )
 RETURNS data-type
 URL url-string
 [ HEADER header-string ]
 [ SOAPHEADER soap-header-string ]
 [ TYPE { 'HTTP[:{ GET | POST } ] ' | 'SOAP[:{ RPC | DOC } ]' } ]
 [ NAMESPACE namespace-string ]
 [ CERTIFICATE certificate-string ]
 [ CLIENTPORT clientport-string ]
 [ PROXY proxy-string ]
```

パラメータ

- **url-string：** – ' { **HTTP** | **HTTPS** | **HTTPS_FIPS** }://[*user:password@*]*hostname*[:*port*][/*path*] '
- **parameter：** – **IN***parameter-name**data-type* [**DEFAULT***expression*]
- **routine-characteristics：** – **ONEXCEPTIONRESUME** | [**NOT**] **DETERMINISTIC**
- **tsql-compound-statement：** – *sql-statement sql-statement …*
- **external-name：** – **EXTERNALNAME***library-call* | **EXTERNALNAME***java-call***LANGUAGEJAVA**
- **library-call：** – '[*operating-system*:]*function-name@library*; …'
- **operating-system：** – **UNIX**
- **java-call：** – '[*package-name*.]*class-name.method-name**method-signature*'

- **method-signature：** **–** ([*field-descriptor*, ….]) *return-descriptor*
- **field-descriptor and return-descriptor：** **– Z** | **B** | **S** | **I** | **J** | **F** | **D** | **C** | **V** | **[** *descriptor* | **L** *class-name*;

例

- **例 1 –** 次の例は、文字列 `firstname` と文字列 `lastname` を連結します。

```
CREATE FUNCTION fullname (
  firstname CHAR(30),
  lastname CHAR(30) )
RETURNS CHAR(61)
BEGIN
  DECLARE name CHAR(61);
  SET name = firstname || ' ' || lastname;
  RETURN (name);
END
```

fullname 関数は、次の例のように使用します。

- 2 つの提供された文字列からフル・ネームを戻します。

```
SELECT fullname ('joe','smith')
```

fullname('joe', 'smith')
joe smith

- 全従業員の名前をリストします。

```
SELECT fullname (givenname, surname)
FROM Employees
```

fullname (givenname, surname)
Fran Whitney
Matthew Cobb
Philip Chin
Julie Jordan
Robert Breault
...

- **例 2 –** 次の例では、Transact-SQL 構文を使用しています。

```
CREATE FUNCTION DoubleIt ( @Input INT )
RETURNS INT
AS
DECLARE @Result INT
SELECT @Result = @Input * 2
RETURN @Result
```

文 SELECT DoubleIt(5) は、10 の値を返します。

- **例 3 –** 次の文は、Java で記述された外部関数を作成します。

```
CREATE FUNCTION dba.encrypt( IN name char(254) )
RETURNS VARCHAR
EXTERNAL NAME
'Scramble.encrypt (Ljava/lang/String;)Ljava/lang/String;'
LANGUAGE JAVA
```

使用法

CREATE FUNCTION 文は、データベース内でユーザ定義関数を作成します。所有者名を指定することにより、別のユーザが使用する関数を作成できます。パーミッションがあれば、ユーザ定義関数も他の非集合関数とまったく同じように使用できます。

CREATE FUNCTION—パラメータの名前は、データベース識別子の規則に従っている必要があります。 有効な SQL データ型を持たなくてはなりません。また、この引数が関数へ値を提供する式であることを示すキーワード **IN** を先頭に付ける必要があります。

CREATE OR REPLACE FUNCTION を指定すると、新しい関数が作成されるか、同じ名前の既存の関数が置き換えられます。関数が置き換えられた場合、関数の定義は変更されますが、既存のパーミッションは維持されます。

OR REPLACE 句をテンポラリ・プロシージャで使用することはできません。

関数を実行する場合は、必ずしもすべてのパラメータを指定する必要はありません。**CREATE FUNCTION** 文の中にデフォルト値がある場合、不明のパラメータにデフォルト値を割り当てます。引数が呼び出し元から提供されておらず、デフォルトが設定されていない場合、エラーが返されます。

TEMPORARY (**CREATETEMPORARYFUNCTION**) を指定すると、作成した接続でのみ参照できる関数になり、接続を削除すると関数も自動的に削除されます。テンポラリ関数を明示的に削除することもできます。テンポラリ関数に対して **ALTER**、**GRANT**、または **REVOKE** は実行できません。また他の関数とは異なり、テンポラリ関数はカタログやトランザクション・ログに記録されていません。

テンポラリ関数は、作成者 (現在のユーザ) のパーミッションで実行されます。また作成者のみが所有できます。そのため、テンポラリ関数を作成するときは所有者を指定しないでください。

読み込み専用のデータベースに接続するときに、テンポラリ関数の作成と削除を行うことができます。

SQL SECURITY—関数が INVOKER (関数を呼び出しているユーザ)、または DEFINER (関数を所有しているユーザ) として実行されているかを指定します。デフォルトは DEFINER です。

SQL SECURITY INVOKER が指定されていると、プロシージャを呼び出す各ユーザに対して注釈を付ける必要があるため、必要なメモリも多くなります。また、**SQL SECURITY INVOKER** が指定されていると、名前解決も INVOKER として実行されます。したがって、適切な所有者を使用して、すべてのオブジェクト名 (テーブルやプロシージャなど) の条件を満たすように注意してください。

compound-statement—**BEGIN** と **END** で囲まれ、セミコロンで区切られた SQL 文のセット。「BEGIN … END 文」を参照してください。

tsql-compound-statement—ひとまとまりの Transact-SQL 文。『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「他の Sybase データベースとの互換性」>「Transact-SQL のプロシージャ言語の概要」>「Transact-SQL のバッチの概要」と「CREATE PROCEDURE 文 [T-SQL]」を参照してください。

EXTERNAL NAME—**EXTERNAL NAME** 句を使用する関数は、外部ライブラリにある関数への呼び出しのラッパです。**EXTERNAL NAME** を使用する関数では、**RETURNS** 句の後に他の句を置くことはできません。ライブラリ名にはファイル拡張子が付く場合があります。この拡張子は通常、Windows では `.dll`、UNIX では `.so` です。拡張子が付いていなければ、ソフトウェアがプラットフォーム固有のデフォルトのファイル拡張子をライブラリに付加します。

EXTERNAL NAME 句はテンポラリ関数ではサポートされていません。『SQL Anywhere サーバー – プログラミング』の「SQL Anywhere 外部呼び出しインターフェイス」を参照してください。

注意： 参照先は SQL Anywhere のマニュアルです。

EXTERNAL NAME LANGUAGE JAVA—**LANGUAGE JAVA** 句が付いた **EXTERNAL NAME** を使用する関数は、Java メソッドのラッパです。Java プロシージャの呼び出しについては、「CREATE PROCEDURE 文」を参照してください。

ON EXCEPTION RESUME—Transact-SQL に似たエラー処理です。「CREATE PROCEDURE 文」を参照してください。

NOT DETERMINISTIC—**NOT DETERMINISTIC** として指定された関数は、クエリから呼び出されるたびに再評価されます。このように指定されない関数では、パフォーマンスを高めるために結果がキャッシュされ、クエリの評価中その関数が同じパラメータで呼び出されるたびに再利用されます。

実行時に、データベースのデータを変更するような関数には、 **NOT DETERMINISTIC** を宣言してください。たとえば、プライマリ・キー値を生成し、**INSERT … SELECT** 文で使用される関数は、次のように **NOT DETERMINISTIC** として宣言してください。

```
CREATE FUNCTION keygen( increment INTEGER )
RETURNS INTEGER
```

```
NOT DETERMINISTIC
BEGIN
  DECLARE keyval INTEGER;
  UPDATE counter SET x = x + increment;
  SELECT counter.x INTO keyval FROM counter;
  RETURN keyval
END
INSERT INTO new_table
SELECT keygen(1), ...
FROM old_table
```

特定の入力パラメータに対して常に同じ値を返す関数は、**DETERMINISTIC** として宣言できます。

NOT DETERMINISTIC が宣言されていないかぎり、すべてのユーザ定義関数は決定的として扱われます。DETERMINISTIC 関数は同じパラメータに対して一定の結果を返すため、二次的な影響を心配する必要がありません。つまり、データベース・サーバは同じ関数を同じパラメータで 2 回連続して呼び出したとしても、返される結果が同じであると想定するため、クエリのセマンティクスにおいて厄介な二次的影響が生じることはなくなります。

注意： CLR、ESQL および ODBC、Perl、PHP の外部環境のユーザ定義関数は、カタログ・ストア (SQL Anywhere) によって処理されます。 Sybase IQ のパフォーマンス機能は使用されません。 Java のユーザ定義関数は Sybase IQ によって処理され、パフォーマンスが向上します。

特定のケースで、SQL Anywhere と Sybase IQ のセマンティクスの違いによって、ユーザ定義関数から発行されたクエリの結果に違いが生じることがあります。 たとえば、Sybase IQ は CHAR と VARCHAR を区別し、異なるデータ型として扱いますが、SQL Anywhere は CHAR データを VARCHAR と同じように扱います。

ユーザ定義関数を修正したり、定義をスクランブルすることによって関数の内容を隠したりするには、**ALTER FUNCTION** 文を使用します。

URL—**HTTP** または **SOAP** Web サービス・クライアント関数を定義する場合にのみ使用します。Web サービスの URL を指定します。オプションのユーザ名とパスワードのパラメータは、**HTTP** 基本認証に必要なクレデンシャルとして機能します。**HTTP** 基本認証は、ユーザとパスワードの情報を base-64 でエンコードし、**HTTP** 要求の "Authentication" ヘッダーに渡します。

Web サービス・クライアント関数の場合、**SOAP** 関数と **HTTP** 関数の戻り型は、VARCHAR などの文字データ型の 1 つです。返される値は、**HTTP** 応答の本体です。**HTTP** ヘッダー情報は含まれません。 ステータス情報などの詳細情報が必要な場合は、関数の代わりにプロシージャを使用します。

パラメータ値は要求の一部として渡されます。使用される構文は、要求のタイプにより異なります。**HTTP:GET** では、パラメータは URL の一部として渡されます。

HTTP:**POST** 要求では、値が要求の本体として配置されます。**SOAP** 要求へのパラメータは、常に要求本文にバンドルされます。

HEADER—**HTTP** サービス・クライアント関数を作成する場合は、この句を使用して、**HTTP** 要求ヘッダーのエントリを追加または変更します。**HTTP** ヘッダーに指定できるのは印字可能な ASCII 文字のみで、大文字と小文字は区別されません。この句の使用法の詳細については、「CREATE PROCEDURE 文」の **HEADER** 句を参照してください。

HTTP ヘッダーの使用法の詳細については、『SQL Anywhere サーバー – プログラミング』の「HTTP Web サービス」>「Web クライアントを使用した Web サービスへのアクセス」>「Web クライアントアプリケーションの開発」>「Web クライアント関数とプロシージャーの要件と推奨事項」>「HTTP 要求ヘッダーの管理」を参照してください。

注意： 参照先は SQL Anywhere のマニュアルです。

SOAPDHEADER—**SOAP** Web サービスを関数として宣言する場合は、この句を使用して 1 つ以上の **SOAP** 要求ヘッダー・エントリを指定します。 **SOAP** ヘッダーは、静的定数として宣言したり、代入パラメータ・メカニズムを使用して動的に設定したりできます (hd1、hd2 などに **IN**、**OUT**、または **INOUT** パラメータを宣言)。Web サービス関数では、1 つ以上の **IN** モード代入パラメータを定義できますが、**INOUT** または **OUT** 代入パラメータは定義できません。この句の使用法の詳細については、『SQL Anywhere サーバー – SQL リファレンス』>「SQL 文」>「SQL 文」>「CREATE PROCEDURE 文 (Web クライアント)」の **SOAPHEADER** 句を参照してください。

注意： 参照先は SQL Anywhere のマニュアルです。

TYPE—Web サービス要求を行う場合に使用するフォーマットを指定します。**SOAP** が指定されている場合、または type 句が含まれていない場合は、デフォルトのタイプである **SOAP**:**RPC** が使用されます。**HTTP** は **HTTP**:**POST** を暗黙的に指定します。**SOAP** 要求は常に XML 文書として送信されるため、**HTTP**:**POST** が **SOAP** 要求を送信するのに常に使用されます。

NAMESPACE—**SOAP** クライアント関数にのみ適用されます。この句は **SOAP**:**RPC** 要求と **SOAP**:**DOC** 要求の両方に通常必要なメソッド・ネームスペースを識別します。要求を処理する **SOAP** サーバは、このネームスペースを使用して、**SOAP** 要求メッセージ本文内のエンティティの名前を解釈します。ネームスペースは、Web サービス・サーバから使用できる **SOAP** サービスの WSDL 記述から取得できます。デフォルト値は、プロシージャの URL のオプションのパス・コンポーネントの直前までです。

CERTIFICATE—安全な (HTTPS) 要求を行うには、HTTPS サーバで使用される証明書にクライアントがアクセスできる必要があります。必要な情報は、セミコロンで区切られたキー／値のペアの文字列で指定されます。証明書はファイルに置かれ、file キーを使用して提供されるファイルの名前、または証明書全体を文字列に配置できますが、両方は配置できません。次のキーを使用できます。

キー	省略形	説明
file		証明書のファイル名
certificate	cert	証明書自体
company	co	証明書で指定された会社
unit		証明書で指定された会社の部門
name		証明書で指定された共通名

証明書は、HTTPS サーバに対する直接の要求、または安全でないサーバから安全なサーバにリダイレクトされる可能性がある要求に対してのみ必要です。

CLIENTPORT—HTTP クライアント・プロシージャが TCP/IP を使用して通信するポート番号を識別します。この句は、ファイアウォールを介する通信のためのものであり、このような通信にのみおすすめします。ファイアウォールは TCP/UDP ポートに従ってフィルタします。単一のポート番号、ポート番号の範囲、または両方の組み合わせを指定できます。たとえば CLIENTPORT '85,90-97' を指定できます。

『システム管理ガイド： 第 1 巻』の「接続パラメータと通信パラメータ」>「ネットワーク通信のパラメータ」>「ClientPort パラメータ (CPort)」を参照してください。

PROXY—プロキシ・サーバの URL を指定します。クライアントがプロキシを介してネットワークにアクセスする場合に使用します。プロシージャがプロキシ・サーバに接続し、そのプロキシ・サーバを介して Web サービスに要求を送信することを示します。

関連する動作

- オートコミット。

標準

- SQL—ISO/ANSI SQL 準拠。
- Sybase—Adaptive Server Enterprise でのサポートなし。

パーミッション

RESOURCE 権限が必要です。

Java 関数を含む外部関数には、DBA 権限が必要です。

DEFAULT_TABLE_UDF_ROW_COUNT オプション

テーブル UDF (C、C++、または Java のいずれかのテーブル UDF) が返すローの推定数のデフォルト値を上書きできます。

指定できる値
0 〜 4294967295

デフォルト値
200000

スコープ
このオプションを設定するには、DBA パーミッションが必要です。PUBLIC グループにはテンポラリのみです。

説明
テーブル UDF では、**DEFAULT_TABLE_UDF_ROW_COUNT** オプションを使用して、テーブル UDF が返すローの推定数をクエリ・プロセッサに伝達できます。Java のテーブル UDF でこの情報を伝達するには、これが唯一の方法です。一方、C または C++ のテーブル UDF では、`describe` フェーズで `EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_ROWS` パラメータを使用して、返すロー数の予想値を伝達することを検討します。`EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_ROWS` の値は、**DEFAULT_PROXY_TABLE_UDF_ROW_COUNT** オプションの値を常に上書きします。

参照：
- クエリ処理の状態 (135 ページ)

TABLE_UDF_ROW_BLOCK_SIZE_KB オプション

サーバが確保するロー・ブロックのサイズをキロバイト単位で制御します。ロー・ブロックはテーブル UDF と TPF によって使用されます。

指定できる値
0 〜 4294967295

デフォルト値
128

スコープ
このオプションを設定するには、DBA パーミッションが必要です。PUBLIC グループにはテンポラリのみです。

説明
サーバからフェッチするロー・ブロックのサイズをキロバイト単位で指定します。

`fetch_into` を使用してテーブル UDF からローをフェッチするときと、`fetch_block` を使用して TPF の入力テーブルからローをフェッチするときに、サーバによってロー・ブロックが確保されます。

ロー・ブロックには、指定のサイズに収まる範囲でできるだけ多くのローが格納されます。指定したロー・ブロックのサイズが、単一のローに必要なサイズに満たない場合には、サーバはロー 1 つ分のサイズを確保します。

FROM 句

SELECT 文に必要なデータベース・テーブルまたはビューを指定します。

構文

```
... FROM table-expression [, …]
```

```
table-expression :
    table-name
    | view-name
    | procedure-name
    | common-table-expression
    | (subquery) [[ AS] derived-table-name [column_name, ...)]]
    | derived-table
    | join-expression
    | ( table-expression, ... )
    | openstring-expression
    | apply-expression
    | contains-expression
    | dml-derived-table
```

```
table-name :
    [ userid.]table-name ]
    [ [ AS ] correlation-name ]
    [ FORCE INDEX ( index-name ) ]
```

```
view-name :
    [ userid.]view-name [ [ AS ] correlation-name ]
```

```
procedure-name :
    [  owner, ] procedure-name ([  parameter, ...])
    [  WITH( column-name datatype,)]
    [ [   AS] correlation-name ]

parameter :
scalar-expression | table-parameter

table-parameter :
TABLE(select-statement) [ OVER (table-parameter-over)]

table-parameter-over :
[ PARTITION BY {ANY| NONE| table-expression } ]
```

```
[ ORDER BY { expression | integer } [ ASC | DESC ] [, …] ]
```

```
derived-table :
    ( select-statement )
    [ AS ] correlation-name [ ( column-name, ... ) ]
```

```
join-expression :
    table-expression join-operator table-expression
    [ ON join-condition ]
```

```
join-operator :
    [ KEY | NATURAL ] [ join-type ] JOIN
    | CROSS JOIN
```

```
join-type :
    INNER
    | LEFT [ OUTER ]
    | RIGHT [ OUTER ]
    | FULL [ OUTER ]
```

```
openstring-expression :
    OPENSTRING ( { FILE | VALUE } string-expression )
    WITH ( rowset-schema )
    [ OPTION ( scan-option ...  ) ]
    [ AS ] correlation-name
```

```
apply-expression :
    table-expression { CROSS | OUTER } APPLY table-expression
```

```
contains-expression :
    { table-name  | view-name } CONTAINS ( column-name [,...], contains-
query ) [ [ AS ] score-correlation-name ]
```

```
rowset-schema :
    column-schema-list
    | TABLE [owner.]table-name [ ( column-list ) ]
```

```
column-schema-list :
    { column-name user-or-base-type |  filler( ) } [ , ... ]
```

```
column-list :
    { column-name | filler( ) } [ , ... ]
```

```
scan-option :
    BYTE ORDER MARK { ON | OFF }
    | COMMENTS INTRODUCED BY comment-prefix
    | DELIMITED BY string
    | ENCODING encoding
    | ESCAPE CHARACTER character
    | ESCAPES { ON | OFF }
    | FORMAT { TEXT  | BCP  }
    | HEXADECIMAL { ON | OFF }
    | QUOTE string
    | QUOTES { ON | OFF }
    | ROW DELIMITED BY string
    | SKIP integer
```

```
    | STRIP { ON | OFF | LTRIM | RTRIM | BOTH }
```

```
contains-query :
    string
```

```
dml-derived-table :
    ( dml-statement ) REFERENCING ( [ table-version-names | NONE ] )
```

```
dml-statement :
    insert-statement
    delete-statement
    update-statement
    merge-statement
```

```
table-version-names :
    OLD [ AS ] correlation-name [ FINAL [ AS ] correlation-name ]
    | FINAL [ AS ] correlation-name
```

例

- **例 1 –** 次は、有効な **FROM** 句です。

```
...
FROM Employees
...
...
FROM Employees NATURAL JOIN Departments
...
...
FROM Customers
KEY JOIN SalesOrders
KEY JOIN SalesOrderItems
KEY JOIN Products
...
```

- **例 2 –** 次のクエリは、クエリ内での抽出テーブルの使用法を示します。

```
SELECT Surname, GivenName, number_of_orders
FROM Customers JOIN
     ( SELECT CustomerID, count(*)
       FROM SalesOrders
        GROUP BY CustomerID )
     AS sales_order_counts ( CustomerID,
                             number_of_orders )
ON ( Customers.ID = sales_order_counts.cust_id )
WHERE number_of_orders > 3
```

使用法

SELECT 文には、文で使用するテーブルを指定するためのテーブル・リストが必要です。

注意： ここでの説明はテーブルに関するものですが、特にことわりがないかぎりビューにも当てはまります。

FROM テーブル・リストは、指定した全テーブルからのすべてのカラムで構成する結果セットを作成します。 最初に、コンポーネント・テーブルのすべてのローの組み合わせが結果セットの中に入ります。次に、組み合わせの数は、通常、ジョイン条件か **WHERE** 条件、またはその両方によって減ります。

- **SCALAR –** *scalar-parameter* は有効な SQL データ型の任意のオブジェクトです。
- **TABLE – TABLE** パラメータの指定には、テーブル、ビュー、または共通の *table-expression* 名を使用できます。これらのオブジェクトが **TABLE** パラメータの外でも使用されている場合には、同じオブジェクトの新しいインスタンスとして扱われます。

  次のクエリは、有効な **FROM** 句の使用例です。同じテーブル T に対する 2 つの参照は、同じテーブル T の別々のインスタンス 2 つとして扱われます。

```
SELECT * FROM T, my_proc(TABLE(SELECT T.Z, T.X FROM T)
OVER(PARTITION BY T.Z));
```

  テーブル・パラメータ関数 (TPF) の例—次のクエリは、有効な **FROM** 句の使用例です。

```
SELECT * FROM R, SELECT * FROM my_udf(1);
SELECT * FROM my_tpf(1, TABLE(SELECT c1, c2 FROM t))
   (my_proc(R.X, TABLE T OVER PARTITION BY T.X)) AS XX;
```

  TABLE パラメータの定義でサブクエリを使用する場合には、次の制約があります。

  - **TABLE** パラメータは **IN** 型である必要があります。
  - **PARTITION BY** 句または **ORDER BY** 句では、抽出テーブルと外部参照のカラムを参照する必要があります。 *expression-list* の式では整数 *K* を使用して **TABLE** 入力パラメータの K 番目のカラムを参照できます。

注意： テーブル UDF を参照できるのは SQL 文の **FROM** 句のみです。

- **PARTITION BY – PARTITION BY** 句では、実行エンジンでの関数呼び出しの実行方法を論理的に指定します。 実行エンジンは各パーティションについて関数を呼び出し、関数はそれぞれの呼び出しでパーティション全体を処理する必要があります。

  PARTITION BY では、入力データのパーティション分割の方法も指定し、関数の各呼び出しで処理するのが 1 つのパーティションのデータとなるようにします。 関数の呼び出し回数とパーティション数が同じにならなくてはなりません。TPF では、並列処理の特性は、サーバと UDF の間の動的なネゴシエーションによって実行時に確立されます。 TPF を並列処理できる場合には、*N* 個の入力パーティションに対して、関数を *M* 回インスタンス化でき、*M <=N* という関係になります。 関数のそれぞれのインスタンス化は複数回呼び出すことができ、それぞれの呼び出しで使用するパーティションは必ず 1 つです。

注意： 実行エンジンは任意の順序のパーティションで関数を呼び出すことができ、関数はパーティションの順序に関係なく同じ結果セットを返すことを前提としています。 パーティションを 2 回の関数呼び出しに分割することはできません。

PARTITION BY*expression-list* または **PARTITION BY ANY** 句に指定できる **TABLE** 入力パラメータは 1 つのみです。 その他のすべての **TABLE** 入力パラメータには、明示的または暗黙的な **PARTITION BY NONE** 句を指定する必要があります。

- **ORDER BY – ORDER BY** 句は、各パーティションの入力データを実行エンジンが *expression-list* によりソートするものと想定することを示します。 UDF は、各パーティションがこの物理プロパティを持つものと想定しています。 存在するパーティションが 1 つのみの場合は、**ORDER BY** の指定に基づいて入力データ全体が順序付けされます。**ORDER BY** 句は、**PARTITION BY NONE** 句の指定があるか **PARTITION BY** 句の指定のない任意の **TABLE** 入力パラメータに対して指定できます。

  全文検索において **FROM** 句で使用される *contains-expression* については、『Sybase IQ の非構造化データ分析の概要』を参照してください。
- **ジョイン –** *join-type* のキーワードには次のようなものがあります。

表 5 : FROM 句の join-type のキーワード

join-type のキーワード	説明
CROSS JOIN	2 つのソース・テーブルの直積 (クロス積) を返す
NATURAL JOIN	2 つのテーブルの間で、同じ名前を持つすべての対応するカラムの等価性を比較する (等価ジョインの特殊なケースであり、カラムの長さとデータ型は同じ)
KEY JOIN	最初のテーブルの外部キーの値と、2 つ目のテーブルのプライマリ・キーの値が同じであるものに制限する
INNER JOIN	結果テーブルから、両方のテーブルに一致するローを持つロー以外をすべて破棄する
LEFT OUTER JOIN	左側のテーブルからの一致しないローは残すが、右側のテーブルからの一致しないローは破棄する
RIGHT OUTER JOIN	右側のテーブルからの一致しないローは残すが、左側のテーブルからの一致しないローは破棄する
FULL OUTER JOIN	左と右の両方のテーブルからの一致しないローを維持する

カンマ形式のジョインとキーワード形式のジョインを、**FROM** 句に混在させないでください。同じクエリを、それぞれいずれかの形式を使用した 2 とおりの

方法で記述できます。 ANSI 構文のキーワード形式の方が望ましいとされています。

次のクエリには、カンマ形式のジョインが使用されています。

```
SELECT *
  FROM Products pr, SalesOrders so, SalesOrderItems si
  WHERE pr.ProductID = so.ProductID
    AND pr.ProductID = si.ProductID;
```

同じクエリを、望ましいとされるキーワード形式のジョインで記述すると次のようになります。

```
SELECT *
  FROM Products pr INNER JOIN SalesOrders so
    ON (pr.ProductID = so.ProductID)
  INNER JOIN SalesOrderItems si
    ON (pr.ProductID = si.ProductID);
```

ON 句は内部ジョイン、左ジョイン、右ジョイン、フル・ジョインのデータをフィルタします。 クロス・ジョインには **ON** 句を付けません。内部ジョインでは、**ON** 句は **WHERE** 句と同じですが、 外部ジョインでは **ON** 句と **WHERE** 句は違います。 外部ジョインの **ON** 句は、クロス積のローをフィルタし、その結果に NULL で拡張された一致しないローを含めます。 その後、**WHERE** 句によって、外部ジョインで生成された一致するローと一致しないローからローを取り除きます。 一致しないローのうち、必要なものまで **WHERE** 句の述部で取り除いてしまわないように注意が必要です。

外部ジョインの **ON** 句の中に、サブクエリを使用することはできません。

Transact-SQL 互換のジョインの記述方法については、『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「他の Sybase データベースとの互換性」を参照してください。

userid を指定すると、別のユーザが所有するテーブルを修飾できます。 ユーザ ID を指定しない場合は、デフォルトで、現在のユーザの所属グループが所有するテーブルを参照します。

この SQL 文の場合にのみ、相関名を使用してテーブルに一時的な名前が付けられます。 カラムはテーブル名で修飾されている必要がありますが、カラムを参照するときにテーブル名が長くて入力が煩わしい場合、この相関名を使うと便利です。 相関名は、同じクエリで複数回、同じテーブルを参照するときに、テーブル・インスタンス間で区別するためにも必要です。 相関名を指定しない場合は、テーブル名を現在の文の相関名として使用します。

テーブル式の中で同じテーブルに対して同じ相関名を 2 回使用する場合、そのテーブルは、1 回だけリストされたものとして扱われます。 次に例を示します。

```
SELECT *
FROM SalesOrders
KEY JOIN SalesOrderItems,
SalesOrders
KEY JOIN Employees
```

SalesOrders テーブルの 2 つのインスタンスは、1 つのインスタンスとして扱われます。これは、次のものと等しくなります。

```
SELECT *
FROM SalesOrderItems
KEY JOIN SalesOrders
KEY JOIN Employees
```

一方、次の文では、Person テーブルの 2 つのインスタンスとして扱われ、異なる相関名 HUSBAND と WIFE を使います。

```
SELECT *
FROM Person HUSBAND, Person WIFE
```

FROM 句に、1 つまたは複数のテーブルかビューの代わりに **SELECT** 文を指定すると、ビューを作成しなくても、グループ上でグループを、またはグループでジョインを使用できます。 **SELECT** 文のこのような使用法は、抽出テーブルと呼ばれます。

ジョイン・カラムのパフォーマンスを高めるために、データ型は似ているものを使用してください。

- **パフォーマンスに関する考慮事項** – クエリによっては、Sybase IQ で最大 16 から 64 までのテーブルを、オプティマイザをオンにした状態で **FROM** 句に指定できます。ただし、非常に複雑なクエリで、16 ～ 18 よりも多いテーブルを **FROM** 句に指定すると、パフォーマンスが損なわれる可能性があります。

注意： **FROM** 句を省略した場合、またはクエリ内のすべてのテーブルが SYSTEM DB 領域にある場合、クエリは Sybase IQ ではなく SQL Anywhere によって処理されます。このため、特に構文とセマンティックの制限やオプションの設定方法の違いによって、動作が変わる可能性があります。

FROM 句を必要としないクエリがある場合は、**FROM iq_dummy** 句を追加することによって、強制的に Sybase IQ で処理させることができます。この iq_dummy は、ユーザが自分のデータベースに作成する 1 ロー 1 カラムのテーブルです。

標準

- SQL—ISO/ANSI SQL 準拠。

- Sybase—**JOIN** 句は、Adaptive Server Enterprise の一部のバージョンではサポートされていません。代わりに、**WHERE** 句を使用してジョインを作成してください。

パーミッション

データベースに接続しておく必要があります。

SELECT 文

データベースから情報を取り出します。

構文

```
SELECT [ ALL | DISTINCT ] [ FIRST | TOP number-of-rows ] select-list
… [ INTO { host-variable-list | variable-list | table-name } ]
… [ INTO LOCAL TEMPORARY TABLE { table-name } ]
… [ FROM table-list ]
… [ WHERE search-condition ]
… [ GROUP BY [ expression [, …]
      | ROLLUP ( expression [, …] )
      | CUBE ( expression [, …] ) ] ]
… [ HAVING search-condition ]
… [ ORDER BY { expression | integer } [ ASC | DESC ] [, …] ]
… [ row-limitation-option ]
```

パラメータ

- **select-list： –**

  ```
  { column-name
  | expression [ [ AS ] alias-name ]
  | * }
  ```

- ***row-limitation-option: –***

  ```
  LIMIT { [ offset-expression, ] limit-expression
  | limit-expression OFFSET offset-expression }
  ```

- ***limit-expression: –*** *simple-expression*
- ***offset-expression: –*** *simple-expression*
- ***simple-expression: –***

  ```
  integer
  | variable
  | ( simple-expression )
  | ( simple-expression { + | - | * } simple-expression )
  ```

例

- **例 1 –** システム・カタログのすべてのテーブルとビューをリストします。

```
SELECT tname
FROM SYS.SYSCATALOG
WHERE tname LIKE 'SYS%' ;
```

- **例 2 –** すべての顧客とその顧客からの注文の総額をリストします。

```
SELECT CompanyName,
  CAST( sum(SalesOrderItems.Quantity *
  Products.UnitPrice) AS INTEGER) VALUE
FROM Customers
  LEFT OUTER JOIN SalesOrders
  LEFT OUTER JOIN SalesOrderItems
  LEFT OUTER JOIN Products
GROUP BY CompanyName
ORDER BY VALUE DESC
```

- **例 3 –** 従業員の数をリストします。

```
SELECT count(*)
FROM Employees;
```

- **例 4 –** 次に、Embedded SQL の SELECT 文を示します。

```
SELECT count(*) INTO :size FROM Employees;
```

- **例 5 –** 年、モデル、色別に売り上げ合計をリストします。

```
SELECT year, model, color, sum(sales)
FROM sales_tab
GROUP BY ROLLUP (year, model, color);
```

- **例 6 –** 値引きのあるすべての項目を抽出し、テンポラリ・テーブルに格納します。

```
SELECT * INTO #TableTemp FROM lineitem
WHERE l_discount < 0.5
```

- **例 7 –** 従業員を姓でソートした場合の最初の従業員に関する情報を返します。

```
SELECT FIRST *
FROM Employees
ORDER BY Surname;
```

- **例 8 –** 姓でソートした場合の先頭 5 人の従業員を返します。

```
SELECT TOP 5 *
FROM Employees
ORDER BY Surname;
```

```
SELECT *
FROM Employees
ORDER BY Surname
LIMIT 5;
```

- **例 9 –** 姓で降順にソートした場合の 5 番目と 6 番目の従業員を返します。

```
SELECT *
FROM Employees
ORDER BY Surname DESC
LIMIT 4,2;
```

- **例 10 –** 姓でソートした場合の先頭 5 人の従業員を返します。

```
CREATE OR REPLACE VARIABLE atop INT = 10;
```

```
SELECT TOP (atop -5) *
FROM Employees
ORDER BY Surname;
```

```
SELECT *
FROM Employees
ORDER BY Surname
LIMIT (atop-5);
```

- **例 11 –** 姓で降順にソートした場合の 5 番目と 6 番目の従業員を返します。

```
CREATE OR REPLACE VARIABLE atop INT = 10;
```

```
SELECT *
FROM Employees
ORDER BY Surname DESC
LIMIT (atop - 8) OFFSET (atop -2 -3 -1);
```

使用法

SELECT 文を Interactive SQL 内で使用して、データベース内のデータを参照したり、データベースから外部ファイルにデータをエクスポートしたりできます。

SELECT 文は、プロシージャまたは Embedded SQL 内でも使用できます。**SELECT** 文がローを 1 つだけ返す場合は、**INTO** 句のある **SELECT** 文を使ってデータベースから結果を取り出します (**SELECT INTO** で作成されるテーブルは、IDENTITY/AUTOINCREMENT テーブルを継承しません)。複数のローを対象にクエリを実行する場合は、カーソルを使用する必要があります。複数のカラムを選択し、*#table* を使用しなければ、**SELECT INTO** によって永久ベース・テーブルが作成されます。**SELECT INTO** に *#table* を指定すると、カラムの数に関係なく、常にテンポラリ・テーブルが作成されます。テーブルへの **SELECT INTO** で、カラムが 1 つしかない場合は、結果がホスト変数に返されます。

注意： テンポラリ・テーブルに対して **SELECT INTO** を実行するスクリプトとストアド・プロシージャを作成するときは、ベース・カラムではない select リスト項目を **CAST** 式にラップすることをおすすめします。これにより、テンポラリ・テーブルのカラムのデータ型が目的のデータ型になることが保証されます。

名前が同じで所有者が異なるテーブルには、エイリアスを使用する必要があります。エイリアスを使用しないクエリは、間違った結果を返します。

```
SELECT * FROM user1.t1
WHERE NOT EXISTS
(SELECT *
FROM user2.t1
WHERE user2.t1.col1 = user1.t.col1);
```

正しい結果を返すには、各テーブルに次のようにエイリアスを使用します。

```
SELECT * FROM user1.t1 U1
WHERE NOT EXISTS
(SELECT *
```

```
FROM user2.t1 U2
WHERE U2.col1 = U1.col1);
```

variable-list がある **INTO** 句を使用できるのは、プロシージャ内だけです。

SELECT 文内の、ベース・テーブルまたはビューが許可されている場所に、ストアド・プロシージャ呼び出しを記述できます。 CIS 機能補正のパフォーマンスに関する考慮事項が適用されます。 たとえば、**SELECT** 文を使用して、プロシージャから結果セットを返すこともできます。 構文と使用例については、『SQL Anywhere サーバー – SQL リファレンス』の「SQL 文」>「SQL 文」>「FROM 句」を参照してください。

注意： 参照先は SQL Anywhere のマニュアルです。

ストアド・プロシージャ内のテンポラリ・テーブルから選択する処理に影響を及ぼす制限については、『システム管理ガイド：第 2 巻』>「プロシージャとバッチの使用」>「プロシージャ入門」>「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。

ALL または DISTINCT—どちらも指定しない場合、**SELECT** 文の句を満たすすべてのローが取り出されます。**DISTINCT** を指定すると、重複した出力ローが除外されます。これは文の結果の射影と呼びます。多くの場合、**DISTINCT** を指定すると文の実行に時間が非常に長くかかるため、**DISTINCT** は必要な場合だけ使用するようにしてください。

DISTINCT を使用する場合、**DISTINCT** パラメータを持つ集合関数を文に含めることはできません。

FIRST または TOP number-of-rows—クエリから返されるローの数を指定します。**FIRST** を指定すると、クエリから選択された最初のローが返されます。**TOP** では、指定された数のローがクエリから返されます。*number-of-rows* には 1 ～ 2147483647 の値を、整数の定数か整数の変数で指定します。

FIRST と **TOP** は、主に **ORDER BY** 句で使用されます。これらのキーワードが **ORDER BY** 句以外で使用された場合、同じクエリを実行しても、そのたびにオプティマイザが異なるクエリ・プランを選択して、結果が変わる可能性があります。

FIRST と **TOP** は、クエリの最上位レベルの **SELECT** でのみ使用できます。したがって、抽出テーブルやビュー定義では使用できません。**FIRST** や **TOP** をビュー定義で使用すると、クエリがビューで実行されたときに、このキーワードが無視される可能性があります。

FIRST の使用は、ROW_COUNT データベース・オプションを 1 に設定する場合と同じです。**TOP** の使用は、ROW_COUNT オプションをロー数と同じ数に設定する場合

と同じです。**TOP** と ROW_COUNT が両方とも設定された場合、**TOP** の値が優先されます。

ROW_COUNT オプションは、グローバル変数、システム関数、またはプロキシ・テーブルを伴うクエリで使用された場合、一貫性のない結果を生成する可能性があります。詳細については、「ROW_COUNT オプション」を参照してください。

select-list—*select-list* は、カンマで区切った式のリストであり、データベースから何を取り出すかを指定します。アスタリスク (*) を指定すると、**FROM** 句に記述された全テーブルの全カラムを選択することを意味します (table-name は、指定したテーブルのすべてのカラムを選択します)。*select-list* には、集合関数と統計関数を使用できます。『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」を参照してください。

注意： Sybase IQ でも、SQL Anywhere や Adaptive Server Enterprise と同様、スカラ・サブクエリ (ネストされた select) を最上位レベルの **SELECT** の select リストで使用できます。 サブクエリを条件値の式の内部 (**CASE** 文の内部など) で使用することはできません。

WHERE 句や **HAVING** 句の述部 (サポートされる述部のタイプのいずれか) でサブクエリを使用することもできます。 ただし、**WHERE** 句または **HAVING** 句の内部であっても、値の式の内部や **CONTAINS** または **LIKE** の述部の内部でサブクエリを使用することはできません。外部ジョインの **ON** 句、または **GROUP BY** 句でサブクエリを使用することはできません。

サブクエリの使用の詳細については、『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 言語の要素」>「式」>「式内のサブクエリ」と『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」>「検索条件」>「探索条件内のサブクエリ」を参照してください。

---

alias-names はクエリ全体で使用でき、エイリアスの式を表します。 Interactive SQL でも、**SELECT** 文から出力された各カラムの最上部にエイリアス名が表示されます。 オプションの *alias-name* が式の後で指定されていなければ、Interactive SQL は式自体を表示します。 カラムのエイリアスにカラム名と同じ名前、または式を使用する場合、名前はテーブル・カラム名でなく、エイリアス・カラムとして処理されます。

INTO host-variable-list—Embedded SQL 内でのみ使用できます。これは **SELECT** 文の結果が移動する場所を指定します。*select-list* 内の各項目に対して、それぞれ 1 つの *host-variable* 項目が必要です。select リスト項目は、順番にホスト変数の中に置かれます。インジケータ・ホスト変数も各 *host-variable* とともに使用でき、プログラムは select リスト項目が NULL であったかどうかを通知できます。

INTO variable-list—使用できるのは、プロシージャ内だけです。これは **SELECT** 文の結果が移動する場所を指定します。select リスト内の各項目に対して、それぞれ 1 つの変数項目が必要です。select リスト項目は、順番に変数の中に置かれます。

INTO table-name—テーブルを作成して、そこにデータを入力するために使用されます。

テーブル名が # で始まる場合は、テンポラリ・テーブルとして作成されます。そうでなければ、テーブルは永久ベース・テーブルとして作成されます。永久テーブルを作成するには、クエリが次のいずれかの条件を満たしている必要があります。

- *select-list* に複数の項目が指定され、**INTO** ターゲットに *table-name* 識別子が 1 つだけ指定されている。または、
- select リストに * が指定され、**INTO** のターゲットが *owner.table* の形で指定されている。

カラムが 1 つである永久テーブルを作成するには、テーブル名を *owner.table* の形で指定します。所有者の指定を省略すると、テンポラリ・テーブルが作成されます。

この文では、テーブルを作成する副作用として実行前に **COMMIT** が行われます。この文の実行には、RESOURCE 権限が必要です。新しいテーブルに対するパーミッションは与えられません。これは、**CREATE TABLE** とそれに続く **INSERT... SELECT** を短縮して定義できる文です。

ストアド・プロシージャまたは関数から **SELECT INTO** を使用することはできません。これは、**SELECT INTO** がアトミック文であり、アトミック文内では **COMMIT**、**ROLLBACK**、または一部の **ROLLBACK TO SAVEPOINT** 文を実行できないためです。詳細については、『システム管理ガイド： 第 2 巻』の「プロシージャとバッチの使用」>「制御文」>「アトミックな複合文」と『システム管理ガイド： 第 2 巻』の「プロシージャとバッチの使用」>「プロシージャ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。

この文を使用して作成されたテーブルには、プライマリ・キーが定義されていません。**ALTER TABLE** を使用してプライマリ・キーを追加できます。プライマリ・キーは、テーブルに UPDATE または DELETE を適用する前に追加してください。そうしないと、これらの操作によって、影響を受けるローのトランザクション・ログにすべてのカラム値が記録されます。

この句は、SQL Anywhere の有効なクエリにしか使用できません。Sybase IQ 拡張機能はサポートされません。

INTO LOCAL TEMPORARY TABLE—ローカル・テンポラリ・テーブルを作成し、そのテーブルにクエリの結果を格納します。この句を使用する場合、テンポラリ・テーブル名の先頭に # を付ける必要はありません。

FROM table-list—*table-list* 内で指定したテーブルとビューからローを取り出します。ジョイン演算子を使用して、ジョインを指定できます。詳細については、「FROM 句」を参照してください。**FROM** 句を指定しない **SELECT** 文を使って、テーブルから取り出せなかった式の値を表示できます。次に例を示します。

```
SELECT @@version
```

これはグローバル変数 @@version の値を表示します。これは次と同じです。

```
SELECT @@version
FROM DUMMY
```

注意： **FROM** 句を省略した場合、またはクエリ内のすべてのテーブルが SYSTEM DB 領域にある場合、クエリは Sybase IQ ではなく SQL Anywhere によって処理されます。このため、特に構文とセマンティックの制限やオプションの設定方法の違いによって、動作が変わる可能性があります。 処理に適用されるルールについては SQL Anywhere のマニュアルを参照してください。

FROM 句を必要としないクエリがある場合は、"`FROM iq_dummy`" 句を追加することによって、強制的に Sybase IQ で処理させることができます。この `iq_dummy` は、ユーザが自分のデータベースに作成する 1 ロー 1 カラムのテーブルです。

WHERE search-condition—**FROM** 句に指定されたテーブルからどのローを選択するかを指定します。これを使用すると、複数のテーブル間のジョインも行えます。これを行うには、**WHERE** 句内に条件を入れます。この WHERE 句は、一方のテーブルのカラムまたはカラム・グループを、他方のテーブルのカラムまたはカラム・グループと関連付けます。両方のテーブルを **FROM** 句内にリストする必要があります。

同じ **CASE** 文を、グループ化したクエリの **SELECT** と **WHERE** 句の両方で使用することはできません。『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「SQL 言語の要素」>「検索条件」を参照してください。

Sybase IQ では、サブクエリ述部の分離もサポートされています。 各サブクエリは、**WHERE** 句または **HAVING** 句内で他の述部とともに指定し、AND 演算子または OR 演算子を使用して結合できます。『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 言語の要素」>「検索条件」>「探索条件内のサブクエリ」>「サブクエリ述部の分離」を参照してください。

GROUP BY—カラム、エイリアス名、または関数によってグループ化できます。**GROUP BY** 式も select リストの前に入れる必要があります。 クエリ結果には、指定したカラム、エイリアス、または関数内の異なる値セットそれぞれに対してローが 1 つずつ入ります。 テーブル・リストの各ロー・グループに対して 1 つのローが出力されるため、出力される各ローをグループと呼ぶことがあります。 **GROUP**

BY 句があると、すべての NULL 値は同じものとして扱われます。 集合関数をこれらのグループに適用して、意味のある結果を取得できます。

GROUP BY には、定数を 2 つ以上含める必要があります。グループ化するクエリに定数を選択するために、**GROUP BY** 句に定数を追加する必要はありません。**GROUP BY** 式に定数が 1 つしか含まれない場合は、エラーが返されてクエリが拒否されます。

GROUP BY を使用する場合、select リスト、**HAVING** 句、**ORDER BY** 句で参照できるのは、**GROUP BY** 句の中で指定した識別子だけです。ただし、 *select-list* と **HAVING** 句は例外で、集合関数を持つことができます。

ROLLUP 演算子—**GROUP BY** 句で **ROLLUP** 演算子を使用すると、さまざまなレベルの詳細を使用して小計を分析できます。非常に詳細なレベルから総計までロールアップする小計を作成します。

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に指定する必要があります。**ROLLUP** は、最初に **GROUP BY** に指定された標準の集合値を計算します。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+1* レベルの小計を作成します。

ROLLUP 演算子には次の制限があります。

- **ROLLUP** 演算子は **GROUP BY** 句で使用可能なすべての集合関数をサポートしますが、**ROLLUP** は現在 **COUNT DISTINCT** と **SUM DISTINCT** をサポートしていません。
- **ROLLUP** は **SELECT** 文でのみ使用できます。**ROLLUP** は **SELECT** サブクエリで使用できません。
- **ROLLUP**、**CUBE**、**GROUP BY** のカラムを同じ **GROUP BY** 句内で組み合わせた複数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。

GROUPING に **ROLLUP** 演算子を付けて使用すると、格納されていた NULL 値と **ROLLUP** によって作成されたクエリ結果の NULL 値を区別できます。

ROLLUP 構文：

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

演算子の式の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「SQL 言語の要素」>「式」を参照してください。

GROUPING は、カラム名をパラメータとして受け取り、ブール値を返します。

表 6 : 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 演算子によっては作成されていない NULL	0 (偽)
格納されていた NULL	0 (偽)

ROLLUP の例については、『システム管理ガイド：第 2 巻』の「OLAP の使用」を参照してください。

CUBE 演算子—**GROUP BY** 句の **CUBE** 演算子は、データを複数の次元でグループ化することでデータを分析します。**CUBE** にはグループ化式 (次元) を順に並べた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** 文は組み合わせ可能なすべての次元のグループに対して、小計を計算できるようになります。

CUBE 演算子には次の制限があります。

- **CUBE** 演算子は **GROUP BY** 句で使用可能なすべての集合関数をサポートしますが、**CUBE** は現在 **COUNT DISTINCT** と **SUM DISTINCT** をサポートしていません。
- **CUBE** は、逆分散統計関数である **PERCENTILE_CONT** と **PERCENTILE_DISC** を現在はサポートしていません。
- **CUBE** は **SELECT** 文でのみ使用できます。**CUBE** は **SELECT** サブクエリでは使用できません。
- **ROLLUP**、**CUBE**、**GROUP BY** のカラムを同じ **GROUP BY** 句内で組み合わせた複数の **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。

GROUPING に **CUBE** 演算子を付けて使用すると、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。

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、ブール値を返します。

表 7 : 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 の戻り値
CUBE 処理によって作成された NULL	1 (真)

結果値の種類	GROUPING の戻り値
ローが小計であることを示す NULL	1 (真)
CUBE 演算子によっては作成されていない NULL	0 (偽)
格納されていた NULL	0 (偽)

クエリ・プランを生成するとき、Sybase IQ オプティマイザは、**GROUP BY CUBE** ハッシュ操作で生成されるグループの合計数を推定します。`MAX_CUBE_RESULTS` データベース・オプションには、実行可能なハッシュ・アルゴリズムに対してオプティマイザが推測する、ローの予想数の上限を設定します。 実際のロー数が `MAX_CUBE_RESULT` オプションの値を超える場合、オプティマイザはクエリの処理を中止し、`"Estimate number: nnn exceed the DEFAULT_MAX_CUBE_RESULT of GROUP BY CUBE or ROLLUP"` というエラー・メッセージを表示します。*nnn* は、オプティマイザが推定した数字です。`MAX_CUBE_RESULT` オプションの設定の詳細については、「MAX_CUBE_RESULT オプション」を参照してください。

CUBE の例については、『システム管理ガイド：第 2 巻』の「OLAP の使用」を参照してください。

HAVING search-condition—個々のロー値ではなく、グループ値に基づきます。**HAVING** 句を使用できるのは、文が **GROUP BY** 句を持っているか、select リストが集合関数だけで構成されている場合だけです。**HAVING** 句の中で参照されるカラム名は、**GROUP BY** 句の中に入れるか、または **HAVING** 句の中の集合関数に対するパラメータとして使用する必要があります。

ORDER BY—クエリ結果をソートします。**ORDER BY** リストの各項目には、昇順の場合は ASC、降順の場合は DESC のラベルを付けることができます。指定がない場合は、昇順であるとみなします。式が整数 **n** である場合、クエリ結果は select リストの **n** 番目の項目でソートされます。

Embedded SQL の場合は、データベースから結果を取得し、その値を **INTO** 句でホスト変数に格納するために、**SELECT** 文を使用します。**SELECT** 文は、1 つのローだけを返すようにします。複数のローを対象にクエリを実行する場合は、カーソルを使用する必要があります。

Java のクラスを **SELECT** リストに指定することはできませんが、たとえば、Java のクラスのラッパとして機能する関数または変数を作成し、それを指定することは可能です。

row-limitation 句—ローを制限する句を使用することによって、**WHERE** 句を満たすローのサブセットのみを返すことができます。*row-limitation* 句は一度に 1 つしか

指定できません。この句を指定するときには、意味がある形でローを順序付けするために **ORDER BY** 句が必要です。

LIMIT および **OFFSET** 引数には、ホスト変数、整数の定数、または整数の変数に対する単純な算術式を使用できます。**LIMIT** 引数は 0 以上の値に評価されなくてはなりません。**OFFSET** 引数は 0 以上の値に評価されなくてはなりません。*offset-expression* を指定しない場合のデフォルト値は 0 です。

ローを制限する句 **LIMIT***offset-expression*, *limit-expression* は、**LIMIT***limit-expression***OFFSET***offset-expression* と同じです。

LIMIT キーワードはデフォルトでは無効になっています。**LIMIT** キーワードを有効にするには RESERVED_KEYWORDS オプションを使用します。

標準

- SQL—ISO/ANSI SQL 準拠。
- Sybase—Adaptive Server Enterprise でサポートされますが、構文にいくつか相違があります。

パーミッション

指定したテーブルとビューに対する SELECT パーミッションが必要です。

a_v4_extfn の API リファレンス

a_v4_extfn の関数、メソッド、属性のリファレンス情報です。

BLOB (a_v4_extfn_blob)

a_v4_extfn_blob 構造体を使用して、独立した BLOB オブジェクトを表現します。

実装

```
typedef struct a_v4_extfn_blob {
    a_sql_uint64 (SQL_CALLBACK *blob_length)(a_v4_extfn_blob *blob);
    void (SQL_CALLBACK *open_istream)(a_v4_extfn_blob *blob,
a_v4_extfn_blob_istream **is);
    void (SQL_CALLBACK *close_istream)(a_v4_extfn_blob *blob,
a_v4_extfn_blob_istream *is);
    void (SQL_CALLBACK *release)(a_v4_extfn_blob *blob);
} a_v4_extfn_blob;
```

メソッドの概要

メソッド名	データ型	説明
blob_length	a_sql_uint64	指定の BLOB の長さをバイト数で返します。
open_istream	void	指定の BLOB からの読み込みを開始するために使用できる入力ストリームを開きます。
close_istream	void	指定の BLOB の入力ストリームを閉じます。
release	void	呼び出し元がこの BLOB の処理を完了し、BLOB の所有者がリソースを解放できることを示します。 **release()** の後で BLOB を参照するとエラーになります。所有者は通常、**release()** が呼び出されたときにメモリを削除します。

説明

オブジェクト a_v4_extfn_blob を使用するのは次のときです。

- テーブル UDF がスカラ入力値から LOB または CLOB データを読み込む必要がある。
- TPF が入力テーブルのカラムから LOB または CLOB データを読み込む必要がある。

制約と制限事項
なし。

blob_length

v4 API メソッド `blob_length` を使用して、指定の BLOB の長さをバイト数で返します。

宣言

```
a_sql_uint64 blob_length(
    a_v4_extfn_blob *
)
```

使用法
指定の BLOB の長さをバイト数で返します。

パラメータ

パラメータ	説明
blob	長さを返す対象の BLOB。

戻り値
指定の BLOB の長さ。

参照：

- open_istream (220 ページ)
- close_istream (221 ページ)
- release (222 ページ)

open_istream

v4 API メソッド `open_istream` を使用して、BLOB から読み込むための入力ストリームを開きます。

宣言

```
void open_istream(
    a_v4_extfn_blob *blob,
    a_v4_extfn_blob_istream **is
    )
```

使用法
指定の BLOB からの読み込みを開始するために使用できる入力ストリームを開きます。

パラメータ

パラメータ	説明
blob	入力ストリームを開く対象の BLOB。
is	オープンして返された入力ストリームを示す出力パラメータ。

戻り値
なし。

参照：
- blob_length (220 ページ)
- close_istream (221 ページ)
- release (222 ページ)

close_istream

v4 API メソッド `close_istream` を使用して、指定の BLOB の入力ストリームを閉じます。

宣言

```
void close_istream(
    a_v4_extfn_blob *blob,
    a_v4_extfn_blob_istream *is
    )
```

使用法
`open_istream` API で開いた入力ストリームを閉じます。

パラメータ

パラメータ	説明
blob	入力ストリームを閉じる対象の BLOB。
is	閉じる入力ストリームを示すパラメータ。

戻り値
なし。

参照：
- blob_length (220 ページ)
- open_istream (220 ページ)

- release (222 ページ)

release

v4 API メソッド `release` を使用して、現在選択中の BLOB の処理を呼び出し元が完了したことを示します。release によって、所有者がメモリを解放できるようになります。

宣言

```
void release(
a_v4_extfn_blob *blob
)
```

使用法

呼び出し元がこの BLOB の処理を完了し、BLOB の所有者がリソースを解放できることを示します。 **release()** の後で BLOB を参照するとエラーになります。所有者は通常、**release()** が呼び出されたときにメモリを削除します。

パラメータ

パラメータ	説明
blob	解放する対象の BLOB。

戻り値

なし。

参照：

- blob_length (220 ページ)
- open_istream (220 ページ)
- close_istream (221 ページ)

BLOB 入力ストリーム (a_v4_extfn_blob_istream)

`a_v4_extfn_blob_istream` 構造体を使用して、LOB または CLOB のスカラ入力カラムの BLOB データや、入力テーブルの LOB または CLOB カラムを読み込みます。

実装

```
typedef struct a_v4_extfn_blob_istream {
    size_t (SQL_CALLBACK *get)( a_v4_extfn_blob_istream *is, void
*buf, size_t len );
    a_v4_extfn_blob     *blob;
    a_sql_byte          *beg;
```

```
    a_sql_byte          *ptr;
    a_sql_byte          *lim;
} a_v4_extfn_blob_istream;
```

メソッドの概要

メソッド名	データ型	説明
get	size_t	指定の分量のデータを BLOB 入力ストリームから取得します。

データ・メンバとデータ型の概要

データ・メンバ	データ型	説明
Blob	a_v4_extfn_blob	この入力ストリームの作成の基になった BLOB 構造体。
Beg	a_sql_byte	現在のデータ・チャンクの先頭位置のポインタ。
Ptr	a_sql_byte	データ・チャンク内の現在のバイト位置のポインタ。
Lim	a_sql_byte	現在のデータ・チャンクの末尾の位置のポインタ。

get

v4 API メソッド `get` を使用して、指定の分量のデータを BLOB 入力ストリームから取得します。

宣言

```
size_t get(
    a_v4_extfn_blob_istream *is,
    void *buf,
    size_t len
)
```

使用法
指定の分量のデータを BLOB 入力ストリームから取得します。

パラメータ

パラメータ	説明
is	データの取得元の入力ストリーム。
buf	データを格納するバッファ。
len	取得するデータの分量。

戻り値
取得したデータの分量。

カラム・データ (a_v4_extfn_column_data)

構造体 a_v4_extfn_column_data は、カラム 1 つ分のデータを表します。これは、プロデューサが結果セットのデータを生成するとき、またはコンシューマが入力テーブルのカラム・データを読み込むときに使用します。

実装

```
typedef struct a_v4_extfn_column_data {
    a_sql_byte          *is_null;
    a_sql_byte          null_mask;
    a_sql_byte          null_value;

    void                *data;
    a_sql_uint32        *piece_len;
    size_t              max_piece_len;

    void                *blob_handle;
} a_v4_extfn_column_data;
```

データ・メンバとデータ型の概要

データ・メンバ	データ型	説明
is_null	a_sql_byte *	値の NULL 情報が格納されているバイトへのポインタ。
null_mask	a_sql_byte	NULL 値を表すために使用する 1 つまたは複数のビット。
null_value	a_sql_byte	NULL を表す値。
data	void *	カラムのデータのポインタ。フェッチ・メカニズムの種類に応じて、コンシューマ内のアドレスか、UDF 内でデータが格納されているアドレスのどちらかを指す。
piece_len	a_sql_uint32 *	可変長データ型のデータの実際の長さ。
max_piece_len	size_t	このカラムが保持できる最大データ長。
blob_handle	void *	NULL 以外の値の場合、このカラムのデータは blob API を使用して読み込む必要があることを表す。

説明

a_v4_extfn_column_data 構造体は、特定のデータ・カラムのデータ値および関連する属性を表します。この構造体は、プロデューサが結果セットのデータを生成するときに使用します。*data*、*piece_len*、および *is_null* フラグの記憶領域もプロデューサが作成することになっています。

is_null、*null_mask*、*null_value* の各データ・メンバは、カラム内の NULL の扱いを表し、8 カラム分の NULL ビットを 1 バイトにコード化するという状況や、各カラムに対して 1 バイトを使用するという状況に対処できます。

次の例は、NULL を表す 3 つのフィールドである *is_null*、*null_mask*、*null_value* を解釈する方法を示します。

```
is_value_null()
   return( (*is_null & null_mask) == null_value )

set_value_null()
   *is_null = ( *is_null & ~null_mask) | null_value

set_value_not_null()
   *is_null = *is_null & ~null_mask | (~null_value & null_mask)
```

参照：

- get_blob (345 ページ)

カラム・リスト (a_v4_extfn_column_list)

a_v4_extfn_column_list 構造体を使用して、describe で **PARTITION BY** を使用するときのカラムのリスト、または **TABLE_UNUSED_COLUMNS** を使用するときのカラムのリストを表します。

実装

```
typedef struct a_v4_extfn_column_list {
    a_sql_int32         number_of_columns;
    a_sql_uint32        column_indexes[1];      // there are
number_of_columns entries
} a_v4_extfn_column_list;
```

データ・メンバとデータ型の概要

データ・メンバ	データ型	説明
number_of_columns	a_sql_uint32	リスト内のカラム数。

データ・メンバ	データ型	説明
column_indexes	a_sql_uint32 *	カラム・インデックス (1 ベース) を使用した、サイズ number_of_columns の連続した配列。

説明
カラム・リストの中身の意味は、**TABLE_PARTITIONBY** と **TABLE_UNUSED_COLUMNS** のどちらで使用するリストかによって変わります。

参照：
- V4 API の describe_parameter と EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY (157 ページ)
- EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY を使用した並列処理 TPF の PARTITION BY の例 (160 ページ)
- EXTFNAPIV4_DESCRIBE_PARM_TABLE_UNUSED_COLUMNS 属性 (get) (284 ページ)
- EXTFNAPIV4_DESCRIBE_PARM_TABLE_UNUSED_COLUMNS 属性 (set) (301 ページ)

カラム順序 (a_v4_extfn_order_el)

`a_v4_extfn_order_el` 構造体を使用して、カラム内の要素の順序を表します。

実装
```
typedef struct a_v4_extfn_order_el {
   a_sql_uint32       column_index;    // Index of the column in the
table (1-based)
   a_sql_byte       ascending;             // Nonzero if the column
is ordered "ascending".
} a_v4_extfn_order_el;
```

データ・メンバとデータ型の概要

データ・メンバ	データ型	説明
column_index	a_sql_uint32	テーブル内のカラムのインデックス (1 ベース)。
ascending	a_sql_byte	カラム順序が昇順の場合は 0 以外。

説明
`a_v4_extfn_order_el` 構造体は、カラムについて表し、昇順と降順のどちらであるかを示します。この構造体の配列を `a_v4_extfn_orderby_list` 構造体が

保持します。a_v4_extfn_order_el 構造体は **ORDERBY** 句の各カラムについて 1 つずつあります。

参照：
- ORDER BY リスト (a_v4_extfn_orderby_list) (333 ページ)

カラム・サブセット (a_v4_extfn_col_subset_of_input)

a_v4_extfn_col_subset_of_input 構造体を使用して、出力カラムの値を常に UDF の特定の入力カラムから取得することを宣言します。

実装
```
typedef struct a_v4_extfn_col_subset_of_input {
    a_sql_uint32    source_table_parameter_arg_num;    // arg_num of
the source table parameter
    a_sql_uint32    source_column_number;        // source column of
the source table
} a_v4_extfn_col_subset_of_input;
```

データ・メンバとデータ型の概要

データ・メンバ	データ型	説明
source_table_parameter_arg_num	a_sql_uint32 *	取得元テーブルのパラメータの *arg_num*。
source_column_number	a_sql_uint32 *	取得元テーブルの取得元カラム。

説明
クエリ・オプティマイザは、入力のサブセットを使用して、出力カラムの値の論理プロパティを推測します。 たとえば、入力カラム内の重複しない値の数は出力カラム内の重複しない値の上限であり、入力カラムのローカル述部は出力カラムでも保持されます。

参照：
- カラム・タイプの記述 (a_v4_extfn_describe_col_type) (307 ページ)

describe の API

_describe_extfn 関数は a_v4_extfn_proc のメンバです。UDF は、a_v4_extfn_proc_context オブジェクトの describe_column、describe_parameter、describe_udf の各プロパティを使用して、論理プロパティを取得および設定します。

_describe_extfn の宣言

```
void (UDF_CALLBACK *_describe_extfn)(a_v4_extfn_proc_context
*cntxt );
)
```

使用法

_describe_extfn 関数はプロシージャの評価をサーバに対して伝えます。

describe_column、describe_parameter、describe_udf の各プロパティには、対応する get メソッドと set メソッド、属性タイプのセット、および各属性に対応するデータ型があります。get メソッドではサーバから情報を取得します。set メソッドでは UDF の論理プロパティ (出力カラム数や、出力カラム内の重複しない値の数) をサーバに伝えます。

参照：

- *describe_column_get (228 ページ)
- *describe_column_set (246 ページ)
- *describe_parameter_get (264 ページ)
- *describe_parameter_set (286 ページ)
- *describe_udf_get (303 ページ)
- *describe_udf_set (305 ページ)
- 外部関数 (a_v4_extfn_proc) (314 ページ)

*describe_column_get

v4 API メソッド describe_column_get は、テーブル UDF がテーブル・パラメータの個別のカラムについてのプロパティを取得するために使用します。

宣言

```
a_sql_int32 (SQL_CALLBACK *describe_column_get)(
    a_v4_extfn_proc_context       *cntxt,
    a_sql_uint32                     arg_num,
    a_sql_uint32                        column_num,
    a_v4_extfn_describe_parm_type describe_type,
    void                                *describe_buffer,
    size_t                              describe_buffer_len );
```

パラメータ

パラメータ	説明
cntxt	この UDF のプロシージャ・コンテキスト・オブジェクト。

パラメータ	説明
arg_num	テーブル・パラメータの順序数 (0 は結果テーブル、1 は最初の入力引数)。
column_num	カラムの順序数 (開始値は 1)。
describe_type	取得するプロパティを示すセレクタ。
describe_buffer	サーバから取得する指定のプロパティの describe 情報を保持する構造体。具体的な構造体またはデータ型は **describe_type** パラメータで示されます。
describe_buffer_length	**describe_buffer** のバイト単位の長さ。

戻り値
成功した場合は、**describe_buffer** に書き込まれたバイト数を返します。エラーが発生した場合や、プロパティが取得されなかった場合は、汎用の `describe_column` エラーのいずれかを返します。

参照：

- *describe_column_set (246 ページ)

***describe_column_get の属性**

v4 API メソッド `describe_column_get` の属性を示すコードです。

```
typedef enum a_v4_extfn_describe_col_type {
    EXTFNAPIV4_DESCRIBE_COL_NAME,
    EXTFNAPIV4_DESCRIBE_COL_TYPE,
    EXTFNAPIV4_DESCRIBE_COL_WIDTH,
    EXTFNAPIV4_DESCRIBE_COL_SCALE,
    EXTFNAPIV4_DESCRIBE_COL_CAN_BE_NULL,
    EXTFNAPIV4_DESCRIBE_COL_DISTINCT_VALUES,
    EXTFNAPIV4_DESCRIBE_COL_IS_UNIQUE,
    EXTFNAPIV4_DESCRIBE_COL_IS_CONSTANT,
    EXTFNAPIV4_DESCRIBE_COL_CONSTANT_VALUE,
    EXTFNAPIV4_DESCRIBE_COL_IS_USED_BY_CONSUMER,
    EXTFNAPIV4_DESCRIBE_COL_MINIMUM_VALUE,
    EXTFNAPIV4_DESCRIBE_COL_MAXIMUM_VALUE,
    EXTFNAPIV4_DESCRIBE_COL_VALUES_SUBSET_OF_INPUT,
    } a_v4_extfn_describe_col_type;
```

EXTFNAPIV4_DESCRIBE_COL_NAME (get)
EXTFNAPIV4_DESCRIBE_COL_NAME 属性はカラム名を示します。
`describe_column_get` のシナリオで使用します。

データ型
`char[]`

説明
カラム名。このプロパティはテーブル引数に対してのみ有効です。

使用法
UDF がこのプロパティを取得すると、指定のカラムの名前が返ります。

戻り値
成功した場合は、カラム名の長さを返します。

失敗した場合は、汎用の `describe_column` エラーのいずれかか、次のいずれかを返します。

- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` — 状態が初期状態より後でない場合に返る get エラー。
- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` — バッファの長さが不十分な場合や長さが 0 の場合に返る get エラー。
- `EXTFNAPIV4_DESCRIBE_NON_TABLE_PARAMETER` — パラメータが TABLE パラメータでない場合に返る get エラー。

クエリ処理の状態
以下のときに有効です。

- 注釈状態
- クエリ最適化状態
- プラン構築状態
- 実行状態

参照：
- EXTFNAPIV4_DESCRIBE_COL_NAME (set) (248 ページ)
- EXTFNAPIV4_DESCRIBE_COL_TYPE (set) (249 ページ)
- EXTFNAPIV4_DESCRIBE_COL_TYPE (get) (231 ページ)
- describe_column の一般的なエラー (353 ページ)
- クエリ処理の状態 (135 ページ)

EXTFNAPIV4_DESCRIBE_COL_TYPE (get)
EXTFNAPIV4_DESCRIBE_COL_TYPE 属性は、カラムのデータ型を示します。
describe_column_get のシナリオで使用します。

データ型
a_sql_data_type

説明
カラムのデータ型。このプロパティはテーブル引数に対してのみ有効です。

使用法
UDF がこのプロパティを取得すると、指定のカラムのデータ型が返ります。

戻り値
成功した場合は、sizeof(a_sql_data_type) を返します。

失敗した場合は、汎用の describe_column エラーのいずれかか、次のいずれかを返します。

- EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH — describe バッファが a_sql_data_type のサイズでない場合に返る get エラー。
- EXTFNAPIV4_DESCRIBE_INVALID_STATE — 状態が初期状態より後でない場合に返る get エラー。

クエリ処理の状態

- 注釈状態
- クエリ最適化状態
- プラン構築状態
- 実行状態

参照：
- describe_column の一般的なエラー (353 ページ)
- EXTFNAPIV4_DESCRIBE_COL_TYPE (set) (249 ページ)
- クエリ処理の状態 (135 ページ)

EXTFNAPIV4_DESCRIBE_COL_WIDTH (get)
EXTFNAPIV4_DESCRIBE_COL_WIDTH 属性は、カラム幅を示します。
describe_column_get のシナリオで使用します。

データ型
a_sql_uint32

説明
カラムの幅。カラム幅は、対応するデータ型の値を格納するために必要な記憶領域のサイズのバイト数です。このプロパティはテーブル引数に対してのみ有効です。

使用法
UDF がこのプロパティを取得すると、**CREATE PROCEDURE** 文で定義されているカラム幅が返ります。

戻り値
成功した場合は、`sizeof(a_sql_uint32)` を返します。

失敗した場合は、汎用の `describe_column` エラーのいずれかか、次のいずれかを返します。

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` – describe バッファが `a_sql_uint32` のサイズでない場合に返る get エラー。
- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` – クエリ処理の状態が初期状態より後でない場合に返る get エラー。

クエリ処理の状態

- 注釈状態
- クエリ最適化状態
- プラン構築状態
- 実行状態

参照：
- EXTFNAPIV4_DESCRIBE_COL_WIDTH (set) (250 ページ)
- describe_column の一般的なエラー (353 ページ)
- クエリ処理の状態 (135 ページ)

EXTFNAPIV4_DESCRIBE_COL_SCALE (get)
EXTFNAPIV4_DESCRIBE_COL_SCALE 属性は、カラムの位取りを示します。`describe_column_get` のシナリオで使用します。

データ型
`a_sql_uint32`

説明
カラムの位取り。算術データ型については、パラメータの位取りは、数値の小数点の右側の桁数です。このプロパティはテーブル引数に対してのみ有効です。

使用法
UDF がこのプロパティを取得すると、**CREATE PROCEDURE** 文で定義されているカラムの位取りが返ります。このプロパティは算術データ型に対してのみ有効です。

戻り値
成功した場合、値が返されたときには `sizeof(a_sql_uint32)` を返します。それ以外のときには次を返します。

- `EXTFNAPIV4_DESCRIBE_NOT_AVAILABLE` – 指定のカラムのデータ型で位取りを取得できない場合に返る get エラー。

失敗した場合は、汎用の `describe_column` エラーのいずれかか、次のいずれかを返します。

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` – describe バッファが `a_sql_uint32` のサイズでない場合に返る get エラー。
- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` – クエリ処理の状態が初期状態より後でない場合に返る get エラー。

クエリ処理の状態

- 注釈状態
- クエリ最適化状態
- プラン構築状態
- 実行状態

参照：
- EXTFNAPIV4_DESCRIBE_COL_SCALE (set) (251 ページ)
- describe_column の一般的なエラー (353 ページ)
- クエリ処理の状態 (135 ページ)

EXTFNAPIV4_DESCRIBE_COL_CAN_BE_NULL (get)
EXTFNAPIV4_DESCRIBE_COL_CAN_BE_NULL 属性は、カラムが NULL になり得るかどうかを示します。`describe_column_get` のシナリオで使用します。

データ型
`a_sql_byte`

説明
カラムが NULL になり得る場合は true。このプロパティはテーブル引数に対してのみ有効です。このプロパティは引数 0 に対してのみ有効です。

使用法
UDF がこのプロパティを取得すると、カラムが NULL になり得る場合には 1 が、その他の場合には 0 が返ります。

戻り値
成功した場合、この属性を取得できたときには `sizeof(a_sql_byte)` を返します。それ以外のときには次を返します。

- `EXTFNAPIV4_DESCRIBE_NOT_AVAILABLE` – 属性を取得できなかった場合に返る。 これは、対象のカラムがクエリに含まれていない場合に起こり得る。

失敗した場合は、汎用の `describe_column` エラーのいずれかか、次のいずれかを返します。

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` — describe バッファが `a_sql_byte` のサイズでない場合に返る get エラー。
- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` — 指定の引数が入力テーブルで、クエリ処理の状態がプラン構築状態より後でない場合に返る get エラー。

クエリ処理の状態
以下のときに有効です。

- 実行状態

参照：
- EXTFNAPIV4_DESCRIBE_COL_CAN_BE_NULL (set) (252 ページ)
- describe_column の一般的なエラー (353 ページ)
- クエリ処理の状態 (135 ページ)

EXTFNAPIV4_DESCRIBE_COL_DISTINCT_VALUES (get)
EXTFNAPIV4_DESCRIBE_COL_DISTINCT_VALUES 属性は、カラム内の重複しない値を表します。`describe_column_get` のシナリオで使用します。

データ型
`a_v4_extfn_estimate`

説明
カラムの重複しない値の推定数。このプロパティはテーブル引数に対してのみ有効です。

使用法
UDF がこのプロパティを取得すると、カラム内の重複しない値の推定数が返ります。

戻り値
成功した場合、値を返したときには `sizeof(a_v4_extfn_estimate)` を返します。それ以外のときには次を返します。

- `EXTFNAPIV4_DESCRIBE_NOT_AVAILABLE` – 属性を取得できなかった場合に返る。 これは、対象のカラムがクエリに含まれていない場合に起こり得る。

失敗した場合は、汎用の `describe_column` エラーのいずれかか、次のいずれかを返します。

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` – describe バッファが `a_v4_extfn_estimate` のサイズでない場合に返る get エラー。
- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` – 指定の引数が入力テーブルで、クエリ処理の状態が最適化状態より後でない場合に返る get エラー。

クエリ処理の状態
以下のときに有効です。

- プラン構築状態
- 実行状態

例
`_describe_extfn` API 関数を使用する、次のようなプロシージャ定義とコードを考えます。

```
CREATE PROCEDURE my_tpf( col_char char(10), col_table TABLE( c1 INT,
c2 INT ) )
RESULTS ( r1 INT, r2 INT, r3 INT )
EXTERNAL ‘my_tpf_proc@mylibrary’;
```

```
CREATE TABLE T( x INT, y INT, z INT );
```

```
select * from my_tpf( 'test', TABLE( select x,y from T ) )
```

この例は、入力テーブルのカラム 1 の重複しない値の数を TPF が取得する方法を示しています。 TPF では、適切な処理アルゴリズムの選択に役立つ場合に、この値を取得できます。

```
my_tpf_describe(a_v4_extfn_proc_context *cntxt)
{
    if( cntxt->current_state == EXTFNAPIV4_STATE_PLAN_BUILDING ) {
           a_v4_extfn_estimate num_distinct;

           a_sql_int32 ret = 0;
```

```
         // Get the number of distinct values expected from the first
column
        // of the table input parameter 'col_table'
           ret = cntxt->describe_column_get( cntxt, 2, 1
                     EXTFNAPIV4_DESCRIBE_COL_DISTINCT_VALUES,
                     &num_distinct,
                     sizeof(a_v4_extfn_estimate) );

           // default algorithm is 1
           _algorithm = 1;

           if( ret > 0 ) {
               // choose the best algorithm for sample size.

               if ( num_distinct.value < 100 ) {
                   // use faster algorithm for small distinct values.
                       _algorithm = 2;
               }
           }
           else {
               if ( ret < 0 ) {
                       // Handle the error
                       //   or continue with default algorithm
               } else {
                       // Attribute was unavailable
                       // We will use the default algorithm.
               }
           }
   }
}
```

参照：

- EXTFNAPIV4_DESCRIBE_COL_DISTINCT_VALUES (set) (254 ページ)
- describe_column の一般的なエラー (353 ページ)
- クエリ処理の状態 (135 ページ)

EXTFNAPIV4_DESCRIBE_COL_IS_UNIQUE (get)

EXTFNAPIV4_DESCRIBE_COL_IS_UNIQUE 属性は、カラムがテーブル内でユニークかどうかを示します。`describe_column_get` のシナリオで使用します。

データ型
`a_sql_byte`

説明
カラムがテーブル内でユニークな場合には true。このプロパティはテーブル引数に対してのみ有効です。

使用法
UDF がこのプロパティを取得すると、カラムがユニークな場合には 1 が、それ以外の場合には 0 が返ります。

戻り値
成功した場合は、`sizeof(a_sql_byte)` を返すか、または次を返します。

- `EXTFNAPIV4_DESCRIBE_NOT_AVAILABLE` – 属性を取得できなかった場合に返る。 これは、対象のカラムがクエリに含まれていない場合に起こり得る。

失敗した場合は、汎用の `describe_column` エラーのいずれかか、次のいずれかを返します。

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` – describe バッファが `a_sql_byte` のサイズでない場合に返る get エラー。
- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` – クエリ処理の状態が初期状態より後でない場合に返る get エラー。

クエリ処理の状態
以下のときに有効です。

- 注釈状態
- クエリ最適化状態
- プラン構築状態
- 実行状態

参照：
- describe_column の一般的なエラー (353 ページ)
- EXTFNAPIV4_DESCRIBE_COL_IS_UNIQUE (set) (255 ページ)
- クエリ処理の状態 (135 ページ)

EXTFNAPIV4_DESCRIBE_COL_IS_CONSTANT (get)
EXTFNAPIV4_DESCRIBE_COL_IS_CONSTANT 属性は、カラムが定数かどうかを示します。`describe_column_get` のシナリオで使用します。

データ型
`a_sql_byte`

説明
文の有効期間全体にわたってカラムが定数の場合には true。このプロパティは入力のテーブル引数に対してのみ有効です。

使用法
UDF がこのプロパティを取得すると、文の有効期間全体にわたってカラムが定数の場合には戻り値は 1、その他の場合には戻り値は 0 です。入力テーブルのカラムが定数となるのは、その入力テーブルの select リストでそのカラムが定数式または NULL の場合です。

戻り値
成功した場合、値を返したときには `sizeof(a_sql_byte)` を返します。それ以外のときには次を返します。

- `EXTFNAPIV4_DESCRIBE_NOT_AVAILABLE` – 属性を取得できなかった。 これは、対象のカラムがクエリに含まれていない場合に返る。

失敗した場合は、汎用の `describe_column` エラーのいずれかか、次のいずれかを返します。

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` – describe バッファが `a_sql_byte` のサイズでない場合に返る get エラー。
- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` – 状態が初期状態より後でない場合に返る get エラー。
- `EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER` – 指定の引数が入力テーブルでない場合に返る get エラー。

クエリ処理の状態
以下のときに有効です。

- 注釈状態
- クエリ最適化状態
- プラン構築状態
- 実行状態

参照：
- EXTFNAPIV4_DESCRIBE_COL_IS_CONSTANT (set) (256 ページ)
- describe_column の一般的なエラー (353 ページ)
- クエリ処理の状態 (135 ページ)

EXTFNAPIV4_DESCRIBE_COL_CONSTANT_VALUE (get)
EXTFNAPIV4_DESCRIBE_COL_CONSTANT_VALUE 属性は、カラムの定数を示します。`describe_column_get` のシナリオで使用します。

データ型
`an_extfn_value`

説明
文の有効期間全体で定数の場合のカラムの値。このカラムの EXTFNAPIV4_DESCRIBE_COL_IS_CONSTANT が true を返す場合、この値を取得できます。このプロパティはテーブル引数に対してのみ有効です。

使用法
定数値を持つ入力テーブルのカラムについては、その値を返します。値を取得できない場合は NULL を返します。

戻り値
成功した場合、値を返したときには sizeof(a_sql_byte) を返します。その他のときには次を返します。

- EXTFNAPIV4_DESCRIBE_NOT_AVAILABLE – 属性を取得できない。 カラムがクエリに含まれていない場合か、値が定数とみなされない場合に返る。

失敗した場合は、汎用の describe_column エラーのいずれかか、次のいずれかを返します。

- EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH – describe バッファが a_sql_byte のサイズでない場合に返る get エラー。
- EXTFNAPIV4_DESCRIBE_INVALID_STATE – 状態が初期状態より後でない場合に返る get エラー。
- EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER – 指定の引数が入力テーブルでない場合に返る get エラー。

クエリ処理の状態
以下のときに有効です。

- 注釈状態
- クエリ最適化状態
- プラン構築状態
- 実行状態

参照：
- describe_column の一般的なエラー (353 ページ)
- EXTFNAPIV4_DESCRIBE_COL_CONSTANT_VALUE (set) (256 ページ)
- クエリ処理の状態 (135 ページ)

EXTFNAPIV4_DESCRIBE_COL_IS_USED_BY_CONSUMER (get)
EXTFNAPIV4_DESCRIBE_COL_IS_USED_BY_CONSUMER 属性は、結果テーブルのカラムをコンシューマが使用しているかどうかを示します。
`describe_column_get` のシナリオで使用します。

データ型
`a_sql_byte`

説明
結果テーブルのカラムをコンシューマが使用するかどうかを判断するため、または入力内のカラムが不要であることを示すために使用します。テーブル引数に対して有効です。単一のカラムについての情報を設定または取得できます。類似の属性 `EXTFNAPIV4_DESCRIBE_PARM_TABLE_UNUSED_COLUMNS` では、1 回の呼び出しですべてのカラムの情報を設定および取得できます。

使用法
UDF は、このプロパティを使用して、結果テーブルのカラムをコンシューマが必要としているかどうかを判断します。これは、使用しないカラムに対する不要な処理を UDF が回避するのに役立ちます。

戻り値
成功した場合は、`sizeof(a_sql_byte)` を返すか、または次を返します。

- `EXTFNAPIV4_DESCRIBE_NOT_AVAILABLE` – 属性を取得できなかった場合に返る。 これは、対象のカラムがクエリに含まれていない場合に起こり得る。

失敗した場合は、汎用の `describe_column` エラーのいずれかか、次のいずれかを返します。

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` – describe バッファが `a_v4_extfn_estimate` のサイズでない場合に返る get エラー。
- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` – 状態が初期状態より後でない場合に返る get エラー。
- `EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER` – 指定の引数が引数 0 でない場合に返る get エラー。

クエリ処理の状態
以下のときに有効です。

- 注釈状態
- クエリ最適化状態
- プラン構築状態

- 実行状態

_describe_extfn API 関数を使用する PROCEDURE 定義とコードです。

```
CREATE PROCEDURE my_tpf( col_char char(10), col_table TABLE( c1 INT,
c2
INT ) )
      RESULTS ( r1 INT, r2 INT, r3 INT )
      EXTERNAL ‘my_tpf_proc@mylibrary’;

CREATE TABLE T( x INT, y INT, z INT );

select r2,r3 from my_tpf( 'test', TABLE( select x,y from T ) )
```

このTPFを実行するときには、結果セットのカラム r1 をユーザが選択したかどうかを把握すると便利です。 ユーザが r1 を必要としない場合には、r1 を計算してサーバ向けに生成することが不要な可能性があります。

```
my_tpf_describe(a_v4_extfn_proc_context *cntxt)
{
    if( cntxt->current_state > EXTFNAPIV4_STATE_INITIAL ) {
      a_sql_byte  col_is_used = 0;
      a_sql_int32 ret = 0;

      ret = cntxt->describe_column_get( cntxt, 0, 1,
            EXTFNAPIV4_DESCRIBE_COL_IS_USED_BY_CONSUMER,
            &col_is_used,
            sizeof(a_sql_byte) );

      if( ret < 0 ) {
          // Handle the error.
      }

    }
```

参照：

- EXTFNAPIV4_DESCRIBE_COL_IS_USED_BY_CONSUMER (set) (257 ページ)
- describe_column の一般的なエラー (353 ページ)
- クエリ処理の状態 (135 ページ)

EXTFNAPIV4_DESCRIBE_COL_MINIMUM_VALUE (get)

EXTFNAPIV4_DESCRIBE_COL_MINIMUM_VALUE 属性は、カラムの最小値を示します。describe_column_get のシナリオで使用します。

データ型
an_extfn_value

説明
カラムの最小値 (取得可能な場合)。引数 0 およびテーブル引数に対してのみ有効です。

使用法
UDF が **EXTFNAPIV4_DESCRIBE_COL_MINIMUM_VALUE** プロパティを取得すると、カラム・データの最小値が **describe_buffer** に返ります。入力テーブルがベース・テーブルの場合には、最小値はテーブル内のすべてのカラム・データに基づいて決まり、対象のテーブル・カラムにインデックスがある場合にのみ使用できます。入力テーブルが別の UDF の結果の場合には、最小値はその UDF が設定した `EXTFNAPIV4_DESCRIBE_COL_TYPE` です。

このプロパティのデータ型はカラムごとに異なります。UDF は `EXTFNAPIV4_DESCRIBE_COL_TYPE` を使用してカラムのデータ型を判断できます。さらに UDF は、`EXTFNAPIV4_DESCRIBE_COL_WIDTH` を使用して、カラムに必要な記憶領域の要件を判断し、過不足のないサイズで値を保持するバッファを提供できます。

サーバは、バッファが有効かどうかを、**describe_buffer_length** を使用して判断できます。

EXTFNAPIV4_DESCRIBE_COL_MINIMUM_VALUE プロパティが取得できない場合、`describe_buffer` は NULL です。

戻り値
成功した場合は、`describe_buffer_length` を返すか、または次を返します。

- `EXTFNAPIV4_DESCRIBE_NOT_AVAILABLE` – 属性を取得できなかった。 対象のカラムがクエリに含まれていない場合か、対象のカラムの最小値を取得できない場合に返る。

失敗した場合は、汎用の `describe_column` エラーのいずれかか、次のいずれかを返します。

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` – describe バッファの大きさが、最小値を保持できる十分なサイズでない場合に返る get エラー。
- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` – 状態が初期状態より後でない場合に返る get エラー。

クエリ処理の状態
初期状態を除く、以下の状態のときに有効です。

- 注釈状態

- クエリ最適化状態
- プラン構築状態
- 実行状態

例
_describe_extfn API 関数を使用するプロシージャ定義とコードです。

```
CREATE PROCEDURE my_tpf( col_char char(10), col_table TABLE( c1 INT,
c2 INT ) )
    RESULTS ( r1 INT, r2 INT, r3 INT )
    EXTERNAL ‘my_tpf_proc@mylibrary’;

CREATE TABLE T( x INT, y INT, z INT );

select * from my_tpf( 'test', TABLE( select x,y from T ) )
```

この例は、TPF が内部的な最適化を目的として入力テーブルのカラム 2 の最小値を取得する方法を示します。

```
my_tpf_describe(a_v4_extfn_proc_context *cntxt)
{
      if( cntxt->current_state > EXTFNAPIV4_STATE_INITIAL ) {
           a_sql_int32 min_value = 0;
           a_sql_int32 ret = 0;

        // Get the minimum value of the second column of the
        // table input parameter 'col_table'

        ret = cntxt->describe_column_get( cntxt, 2, 2
                  EXTFNAPIV4_DESCRIBE_COL_MINIMUM_VALUE,
                  &min_value,
                  sizeof(a_sql_int32) );

     if( ret < 0 ) {
               // Handle the error.
        }

      }
}
```

参照：
- クエリ処理の状態 (135 ページ)
- EXTFNAPIV4_DESCRIBE_COL_MINIMUM_VALUE (set) (259 ページ)
- EXTFNAPIV4_DESCRIBE_COL_TYPE (get) (231 ページ)
- EXTFNAPIV4_DESCRIBE_COL_TYPE (set) (249 ページ)
- EXTFNAPIV4_DESCRIBE_COL_WIDTH (get) (231 ページ)
- EXTFNAPIV4_DESCRIBE_COL_WIDTH (set) (250 ページ)
- describe_column の一般的なエラー (353 ページ)

EXTFNAPIV4_DESCRIBE_COL_MAXIMUM_VALUE (get)
EXTFNAPIV4_DESCRIBE_COL_MAXIMUM_VALUE 属性は、カラムの最大値を示します。`describe_column_get` のシナリオで使用します。

データ型
`an_extfn_value`

説明
カラムの最大値。このプロパティは引数 0 およびテーブル引数に対してのみ有効です。

使用法
UDF が `EXTFNAPIV4_DESCRIBE_COL_MAXIMUM_VALUE` プロパティを取得すると、カラム・データの最大値が **describe_buffer** に返ります。入力テーブルがベース・テーブルの場合には、最大値はテーブル内のすべてのカラム・データに基づいて決まり、対象のテーブル・カラムにインデックスがある場合にのみ使用できます。入力テーブルが別の UDF の結果の場合には、最大値はその UDF が設定した `COL_MAXIMUM_VALUE` です。

このプロパティのデータ型はカラムごとに異なります。UDF は `EXTFNAPIV4_DESCRIBE_COL_TYPE` を使用してカラムのデータ型を判断できます。さらに UDF は、`EXTFNAPIV4_DESCRIBE_COL_WIDTH` を使用して、カラムに必要な記憶領域の要件を判断し、過不足のないサイズで値を保持するバッファを提供できます。

サーバは、バッファが有効かどうかを、**describe_buffer_length** を使用して判断できます。

`EXTFNAPIV4_DESCRIBE_COL_MAXIMUM_VALUE` が取得できない場合、`describe_buffer` は NULL です。

戻り値
成功した場合は、`describe_buffer_length` を返すか、または次を返します。

- `EXTFNAPIV4_DESCRIBE_NOT_AVAILABLE` – 属性を取得できなかった場合に返る。 これは、対象のカラムがクエリに含まれていない場合か、対象のカラムの最大値を取得できない場合に起こり得る。

失敗した場合は、汎用の `describe_column` エラーのいずれかか、次のいずれかを返します。

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` – describe バッファの大きさが、最大値を保持できる十分なサイズでない場合に返る get エラー。

- EXTFNAPIV4_DESCRIBE_INVALID_STATE – 状態が初期状態より後でない場合に返る get エラー。

クエリ処理の状態
初期状態を除く、以下の状態のときに有効です。

- 注釈状態
- クエリ最適化状態
- プラン構築状態
- 実行状態

例
_describe_extfn API 関数を使用する **PROCEDURE** 定義とコードです。

```
CREATE PROCEDURE my_tpf( col_char char(10), col_table TABLE( c1 INT,
c2 INT ) )
         RESULTS ( r1 INT, r2 INT, r3 INT )
         EXTERNAL ‘my_tpf_proc@mylibrary’;

CREATE TABLE T( x INT, y INT, z INT );

select * from my_tpf( 'test', TABLE( select x,y from T ) )
```

この例は、TPF が内部的な最適化を目的として入力テーブルのカラム 2 の最大値を取得する方法を示します。

```
my_tpf_describe(a_v4_extfn_proc_context *cntxt)
{
    if( cntxt->current_state > EXTFNAPIV4_STATE_INITIAL ) {
         a_sql_int32 max_value = 0;
         a_sql_int32 ret = 0;

         // Get the maximum value of the second column of the
         // table input parameter  'col_table'
         ret = cntxt->describe_column_get( cntxt, 2, 2
                  EXTFNAPIV4_DESCRIBE_COL_MAXIMUM_VALUE,
                  &max_value,
                  sizeof(a_sql_int32) );

         if( ret < 0 ) {
             // Handle the error.
            }
       }
}
```

参照：
- クエリ処理の状態 (135 ページ)
- EXTFNAPIV4_DESCRIBE_COL_MAXIMUM_VALUE (set) (261 ページ)
- EXTFNAPIV4_DESCRIBE_COL_TYPE (get) (231 ページ)
- EXTFNAPIV4_DESCRIBE_COL_TYPE (set) (249 ページ)

- EXTFNAPIV4_DESCRIBE_COL_WIDTH (get) (231 ページ)
- EXTFNAPIV4_DESCRIBE_COL_WIDTH (set) (250 ページ)
- describe_column の一般的なエラー (353 ページ)

EXTFNAPIV4_DESCRIBE_COL_VALUES_SUBSET_OF_INPUT (get)
EXTFNAPIV4_DESCRIBE_COL_VALUES_SUBSET_OF_INPUT 属性では、入力カラムが示す値のサブセットを設定します。この属性を `describe_column_get` のシナリオで使用するとエラーが返ります。

データ型
`a_v4_extfn_col_subset_of_input`

説明
カラム値は入力カラムが示す値のサブセットです。

使用法
この属性は設定のみが可能です。

戻り値
エラー `EXTFNAPIV4_DESCRIBE_INVALID_ATTRIBUTE` が返ります。

クエリ処理の状態
どの状態でも、エラー `EXTFNAPIV4_DESCRIBE_INVALID_ATTRIBUTE` が返ります。

参照：
- EXTFNAPIV4_DESCRIBE_COL_VALUES_SUBSET_OF_INPUT (set) (263 ページ)
- describe_column の一般的なエラー (353 ページ)
- クエリ処理の状態 (135 ページ)

*describe_column_set

v4 API メソッド `describe_column_set` では、UDF のカラムレベルのプロパティをサーバに対して設定します。

説明
カラムレベルのプロパティでは、結果セットまたは TPF の入力テーブルのカラムについてのさまざまな特性を記述します。たとえば、UDF からサーバに対して、結果セットのカラムは重複しない値を 10 個のみ持つということを伝えることができます。

宣言

```
a_sql_int32 (SQL_CALLBACK *describe_column_set)(
    a_v4_extfn_proc_context        *cntxt,
    a_sql_uint32        arg_num,
    a_sql_uint32        column_num,
    a_v4_extfn_describe_udf_type        describe_type,
    const void        *describe_buffer,
    size_t        describe_buffer_len );
```

パラメータ

パラメータ	説明
cntxt	この UDF のプロシージャ・コンテキスト・オブジェクト。
arg_num	テーブル・パラメータの順序数 (0 は結果テーブル、1 は最初の入力引数)。
column_num	カラムの順序数 (開始値は 1)。
describe_type	設定するプロパティを示すセレクタ。
describe_buffer	サーバで設定する、指定のプロパティの describe 情報を保持する構造体。具体的な構造体またはデータ型は **describe_type** パラメータで示されます。
describe_buffer_length	**describe_buffer** のバイト単位の長さ。

戻り値

成功した場合は、**describe_buffer** に書き込まれたバイト数を返します。エラーが発生した場合や、プロパティが取得されなかった場合は、汎用の `describe_column` エラーのいずれかを返します。

参照：

- *describe_column_get (228 ページ)

***describe_column_set の属性**

`describe_column_set` の属性を示すコードです。

```
typedef enum a_v4_extfn_describe_col_type {
    EXTFNAPIV4_DESCRIBE_COL_NAME,
    EXTFNAPIV4_DESCRIBE_COL_TYPE,
    EXTFNAPIV4_DESCRIBE_COL_WIDTH,
    EXTFNAPIV4_DESCRIBE_COL_SCALE,
    EXTFNAPIV4_DESCRIBE_COL_CAN_BE_NULL,
    EXTFNAPIV4_DESCRIBE_COL_DISTINCT_VALUES,
    EXTFNAPIV4_DESCRIBE_COL_IS_UNIQUE,
```

```
    EXTFNAPIV4_DESCRIBE_COL_IS_CONSTANT,
    EXTFNAPIV4_DESCRIBE_COL_CONSTANT_VALUE,
    EXTFNAPIV4_DESCRIBE_COL_IS_USED_BY_CONSUMER,
    EXTFNAPIV4_DESCRIBE_COL_MINIMUM_VALUE,
    EXTFNAPIV4_DESCRIBE_COL_MAXIMUM_VALUE,
    EXTFNAPIV4_DESCRIBE_COL_VALUES_SUBSET_OF_INPUT,
    } a_v4_extfn_describe_col_type;
```

EXTFNAPIV4_DESCRIBE_COL_NAME (set)

EXTFNAPIV4_DESCRIBE_COL_NAME 属性はカラム名を示します。`describe_column_set` のシナリオで使用します。

データ型
`char[]`

説明
カラム名。このプロパティはテーブル引数に対してのみ有効です。

使用法
引数 0 について、UDF がこのプロパティを設定すると、サーバはその値を、**CREATE PROCEDURE** 文で指定されたカラム名と比較します。この比較によって、**CREATE PROCEDURE** 文のカラム名が UDF が予期するものと同じであることを確認できます。

戻り値
成功した場合は、カラム名の長さを返します。

失敗した場合は、汎用の `describe_column` エラーのいずれかか、次のいずれかを返します。

- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` – 状態が注釈状態でない場合に返る set エラー。
- `EXTFNAPIV4_DESCRIBE_NON_TABLE_PARAMETER` – パラメータがテーブル・パラメータでない場合に返る set エラー。
- `EXTFNAPIV4_DESCRIBE_ INVALID_ATTRIBUTE_VALUE` – 入力カラム名の長さが 128 文字を超える場合、またはカタログに格納されているカラム名と入力カラム名が一致しない場合に返る set エラー。

クエリ処理の状態
- 注釈状態

例

```
short desc_rc = 0;
     char name[7] = ‘column1’;
```

```
    // Verify that the procedure was created with the second column
of the result table as an int
     if( ctx->current_state == EXTFNAPIV4_STATE_ANNOTATION ) {
          desc_rc = ctx->describe_column_set( ctx, 0, 2,
EXTFNAPIV4_DESCRIBE_COL_NAME,
                                                                name,
                                                          sizeof(name) );
          if( desc_rc < 0 ) {
          // handle the error.
          }
     }
```

参照：

- EXTFNAPIV4_DESCRIBE_COL_NAME (get) (230 ページ)
- EXTFNAPIV4_DESCRIBE_COL_TYPE (set) (249 ページ)
- EXTFNAPIV4_DESCRIBE_COL_TYPE (get) (231 ページ)
- describe_column の一般的なエラー (353 ページ)
- クエリ処理の状態 (135 ページ)

EXTFNAPIV4_DESCRIBE_COL_TYPE (set)

`EXTFNAPIV4_DESCRIBE_COL_TYPE` 属性は、カラムのデータ型を示します。`describe_column_set` のシナリオで使用します。

データ型
`a_sql_data_type`

説明
カラムのデータ型。このプロパティはテーブル引数に対してのみ有効です。

使用法
引数 0 について、UDF がこのプロパティを設定すると、サーバはその値を、**CREATE PROCEDURE** 文で指定されたカラムのデータ型と比較します。これによって UDF は、**CREATE PROCEDURE** 文のデータ型が UDF が予期するものと同じであることを確認できます。

戻り値
成功した場合は、`a_sql_data_type` を返します。

失敗した場合は、汎用の `describe_column` エラーのいずれかか、次のいずれかを返します。

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` – describe バッファが `a_sql_data_type` のサイズでない場合に返る set エラー。

- EXTFNAPIV4_DESCRIBE_INVALID_STATE – 状態が注釈状態でない場合に返る set エラー。
- EXTFNAPIV4_DESCRIBE_ INVALID_ATTRIBUTE_VALUE – カタログに格納されているデータ型と入力データ型が一致しない場合に返る set エラー。

クエリ処理の状態

- 注釈状態

例

```
short desc_rc = 0;
a_sql_data_type type = DT_INT;

   // Verify that the procedure was created with the second column of
the result table as an int
    if( ctx->current_state == EXTFNAPIV4_STATE_ANNOTATION ) {
        desc_rc = ctx->describe_column_set( ctx, 0, 2,
EXTFNAPIV4_DESCRIBE_COL_TYPE,
        &type,
                sizeof(a_sql_data_type) );
        if( desc_rc < 0 ) {
            // handle the error.
        }
    }
```

参照：

- describe_column の一般的なエラー (353 ページ)
- EXTFNAPIV4_DESCRIBE_COL_TYPE (get) (231 ページ)
- クエリ処理の状態 (135 ページ)

EXTFNAPIV4_DESCRIBE_COL_WIDTH (set)

EXTFNAPIV4_DESCRIBE_COL_WIDTH 属性は、カラム幅を示します。 `describe_column_set` のシナリオで使用します。

データ型

`a_sql_uint32`

説明

カラムの幅。カラム幅は、対応するデータ型の値を格納するために必要な記憶領域のサイズのバイト数です。このプロパティはテーブル引数に対してのみ有効です。

使用法

UDF がこのプロパティを設定すると、サーバはその値を、**CREATE PROCEDURE** 文で指定されたカラム幅と比較します。これによって UDF は、**CREATE PROCEDURE** 文のカラム幅が UDF が予期するものと同じであることを確認できます。

戻り値
成功した場合は、`sizeof(a_sql_uint32)` を返します。

失敗した場合は、汎用の `describe_column` エラーのいずれかか、次のいずれかを返します。

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` – describe バッファが `a_sql_uint32` のサイズでない場合に返る set エラー。
- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` – クエリ処理の状態が注釈状態でない場合に返る set エラー。
- `EXTFNAPIV4_DESCRIBE_INVALID_ATTRIBUTE_VALUE` – カタログに格納されている幅と入力幅が一致しない場合に返る set エラー。

クエリ処理の状態
以下のときに有効です。

- 注釈状態

参照：
- EXTFNAPIV4_DESCRIBE_COL_WIDTH (get) (231 ページ)
- describe_column の一般的なエラー (353 ページ)
- クエリ処理の状態 (135 ページ)

EXTFNAPIV4_DESCRIBE_COL_SCALE (set)
EXTFNAPIV4_DESCRIBE_COL_SCALE 属性は、カラムの位取りを示します。`describe_column_set` のシナリオで使用します。

データ型
`a_sql_uint32`

説明
カラムの位取り。算術データ型については、パラメータの位取りは、数値の小数点の右側の桁数です。このプロパティはテーブル引数に対してのみ有効です。

使用法
UDF がこのプロパティを設定すると、サーバはその値を、**CREATE PROCEDURE** 文で指定された位取りと比較します。これによって UDF は、**CREATE PROCEDURE** 文のカラム幅が UDF が予期するものと同じであることを確認できます。このプロパティは算術データ型に対してのみ有効です。

戻り値
成功した場合は、`sizeof(a_sql_uint32)` を返すか、または次を返します。

- EXTFNAPIV4_DESCRIBE_NOT_AVAILABLE – 指定のカラムのデータ型で位取りを取得できない場合に返る set エラー。

失敗した場合は、汎用の describe_column エラーのいずれかか、次のいずれかを返します。

- EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH – describe バッファが a_sql_uint32 のサイズでない場合に返る set エラー。
- EXTFNAPIV4_DESCRIBE_INVALID_STATE – クエリ処理の状態が注釈状態でない場合に返る set エラー。
- EXTFNAPIV4_DESCRIBE_INVALID_ATTRIBUTE_VALUE – カタログに格納されている位取りと入力の位取りが一致しない場合に返る set エラー。

クエリ処理の状態
以下のときに有効です。

- 注釈状態

例

```
    short desc_rc = 0;
    a_sql_uint32 scale = 0;

            // Verify that the procedure has a scale of zero for the
second result table column.
        if( ctx->current_state == EXTFNAPIV4_STATE_ANNOTATION ) {
        desc_rc = ctx->describe_column_set( ctx, 0, 2,
EXTFNAPIV4_DESCRIBE_COL_SCALE,
            &scale,
            sizeof(a_sql_data_type) );
        if( desc_rc < 0 ) {
            // handle the error.
        }
    }
```

参照：

- EXTFNAPIV4_DESCRIBE_COL_SCALE (get) (232 ページ)
- describe_column の一般的なエラー (353 ページ)
- クエリ処理の状態 (135 ページ)

EXTFNAPIV4_DESCRIBE_COL_CAN_BE_NULL (set)
EXTFNAPIV4_DESCRIBE_COL_CAN_BE_NULL 属性は、カラムが NULL になり得るかどうかを示します。describe_column_set のシナリオで使用します。

データ型
a_sql_byte

説明
カラムが NULL になり得る場合は true。このプロパティはテーブル引数に対してのみ有効です。このプロパティは引数 0 に対してのみ有効です。

使用法
UDF は、NULL になり得る結果テーブル・カラムに対してこのプロパティを設定できます。UDF がこのプロパティを明示的に設定しない場合、カラムは NULL になり得るものと見なされます。サーバはこの情報を、最適化状態のときに使用できます。

戻り値
成功した場合、属性が設定されたときには `sizeof(a_sql_byte)` を返します。それ以外のときには次を返します。

- `EXTFNAPIV4_DESCRIBE_NOT_AVAILABLE` – 属性を設定できなかった場合に返る。これは、対象のカラムがクエリに含まれていない場合に起こり得る。

失敗した場合は、汎用の `describe_column` エラーのいずれかか、次のいずれかを返します。

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` – describe バッファが `a_sql_byte` のサイズでない場合に返る set エラー。
- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` – 状態が最適化状態でない場合に返る set エラー。
- `EXTFNAPIV4_DESCRIBE_INVALID_ATTRIBUTE_VALUE` – UDF がこの属性を 0 と 1 以外の値に設定しようとした場合に返る set エラー。

クエリ処理の状態
以下のときに有効です。

- 最適化状態

参照：
- EXTFNAPIV4_DESCRIBE_COL_CAN_BE_NULL (get) (233 ページ)
- describe_column の一般的なエラー (353 ページ)
- クエリ処理の状態 (135 ページ)

EXTFNAPIV4_DESCRIBE_COL_DISTINCT_VALUES (set)
EXTFNAPIV4_DESCRIBE_COL_DISTINCT_VALUES 属性は、カラム内の重複しない値を表します。`describe_column_set` のシナリオで使用します。

データ型
`a_v4_extfn_estimate`

説明
カラムの重複しない値の推定数。このプロパティはテーブル引数に対してのみ有効です。

使用法
UDF は、結果テーブル内のカラムが持つことのできる重複しない値の数を把握している場合にこのプロパティを設定できます。サーバはこの情報を、最適化状態のときに使用します。

戻り値
成功した場合、値を設定したときには `sizeof(a_v4_extfn_estimate)` を返します。それ以外のときには次を返します。

- `EXTFNAPIV4_DESCRIBE_NOT_AVAILABLE` – 属性を設定できなかった場合に返る。 これは、対象のカラムがクエリに含まれていない場合に起こり得る。

失敗した場合は、次のいずれかを返します。

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` – describe バッファが `a_v4_extfn_estimate` のサイズでない場合に返る set エラー。
- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` – 状態が最適化状態でない場合に返る set エラー。

クエリ処理の状態
以下のときに有効です。

- 最適化状態

参照：
- EXTFNAPIV4_DESCRIBE_COL_DISTINCT_VALUES (get) (234 ページ)
- describe_column の一般的なエラー (353 ページ)
- クエリ処理の状態 (135 ページ)

EXTFNAPIV4_DESCRIBE_COL_IS_UNIQUE (set)
EXTFNAPIV4_DESCRIBE_COL_IS_UNIQUE 属性は、カラムがテーブル内でユニークかどうかを示します。`describe_column_set` のシナリオで使用します。

データ型
`a_sql_byte`

説明
カラムがテーブル内でユニークな場合には true。このプロパティはテーブル引数に対してのみ有効です。

使用法
UDF は、結果テーブルのカラム値がユニークかどうかを把握している場合にこのプロパティを設定できます。サーバはこの情報を、最適化状態のときに使用します。UDF はこのプロパティを引数 0 に対してのみ設定できます。

戻り値
成功した場合は、`sizeof(a_sql_byte)` を返すか、または次を返します。

- `EXTFNAPIV4_DESCRIBE_NOT_AVAILABLE` – 属性を設定できなかった場合に返る。 これは、対象のカラムがクエリに含まれていない場合に起こり得る。

失敗した場合は、汎用の `describe_column` エラーのいずれかか、次のいずれかを返します。

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` – describe バッファが `a_sql_byte` のサイズでない場合に返る set エラー。
- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` – クエリ処理の状態が最適化状態でない場合に返る set エラー。
- `EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER` – **arg_num** が 0 でない場合に返る set エラー。
- `EXTFNAPIV4_DESCRIBE_INVALID_ATTRIBUTE_VALUE` – UDF がこの属性を 0 と 1 以外の値に設定しようとした場合に返る set エラー。

クエリ処理の状態
以下のときに有効です。

- 最適化状態

参照：
- describe_column の一般的なエラー (353 ページ)
- EXTFNAPIV4_DESCRIBE_COL_IS_UNIQUE (get) (236 ページ)

- クエリ処理の状態 (135 ページ)

EXTFNAPIV4_DESCRIBE_COL_IS_CONSTANT (set)
EXTFNAPIV4_DESCRIBE_COL_IS_CONSTANT 属性は、カラムが定数かどうかを示します。`describe_column_set` のシナリオで使用します。

データ型
`a_sql_byte`

説明
文の有効期間全体にわたってカラムが定数の場合には true。このプロパティは入力のテーブル引数に対してのみ有効です。

使用法
これは読み込み専用プロパティです。設定しようとすると、必ず `EXTFNAPIV4_DESCRIBE_INVALID_ATTRIBUTE` が返ります。

戻り値

- `EXTFNAPIV4_DESCRIBE_INVALID ATTRIBUTE` – これは読み込み専用プロパティです。設定しようとすると必ずこのエラーが返ります。
- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` – 状態が最適化状態でない場合に返る set エラー。
- `EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER` – **arg_num** が 0 でない場合に返る set エラー。
- `EXTFNAPIV4_DESCRIBE_INVALID_ATTRIBUTE_VALUE` – UDF がこの属性を 0 と 1 以外の値に設定しようとした場合に返る set エラー。

クエリ処理の状態
適用されません。

参照：

- EXTFNAPIV4_DESCRIBE_COL_IS_CONSTANT (get) (237 ページ)
- describe_column の一般的なエラー (353 ページ)
- クエリ処理の状態 (135 ページ)

EXTFNAPIV4_DESCRIBE_COL_CONSTANT_VALUE (set)
EXTFNAPIV4_DESCRIBE_COL_CONSTANT_VALUE 属性は、カラムの定数を示します。`describe_column_set` のシナリオで使用します。

データ型
`an_extfn_value`

説明
文の有効期間全体で定数の場合のカラムの値。このカラムの EXTFNAPIV4_DESCRIBE_COL_IS_CONSTANT が true を返す場合、この値を取得できます。このプロパティはテーブル引数に対してのみ有効です。

使用法
このプロパティは読み込み専用です。

戻り値

- EXTFNAPIV4_DESCRIBE_INVALID_ATTRIBUTE – これは読み込み専用プロパティです。設定しようとすると必ずこのエラーが返ります。

クエリ処理の状態
適用されません。

参照：
- describe_column の一般的なエラー (353 ページ)
- EXTFNAPIV4_DESCRIBE_COL_CONSTANT_VALUE (get) (238 ページ)
- クエリ処理の状態 (135 ページ)

EXTFNAPIV4_DESCRIBE_COL_IS_USED_BY_CONSUMER (set)
EXTFNAPIV4_DESCRIBE_COL_IS_USED_BY_CONSUMER 属性は、結果テーブルのカラムをコンシューマが使用しているかどうかを示します。
describe_column_set のシナリオで使用します。

データ型
a_sql_byte

説明
結果テーブルのカラムをコンシューマが使用するかどうかを判断するため、または入力内のカラムが不要であることを示すために使用します。テーブル引数に対して有効です。単一のカラムについての情報を設定または取得できます。類似の属性 EXTFNAPIV4_DESCRIBE_PARM_TABLE_UNUSED_COLUMNS では、1 回の呼び出しですべてのカラムの情報を設定および取得できます。

使用法
UDF は、入力テーブルのカラムに **EXTFNAPIV4_DESCRIBE_COL_IS_USED_BY_CONSUMER** を設定することによって、そのカラムの値が必要ないことをプロデューサに伝えます。

戻り値
成功した場合は、`sizeof(a_sql_byte)` を返すか、または次を返します。

- `EXTFNAPIV4_DESCRIBE_NOT_AVAILABLE` – 属性を設定できなかった場合に返る。 これは、対象のカラムがクエリに含まれていない場合に起こり得る。

失敗した場合は、汎用の `describe_column` エラーのいずれかか、次のいずれかを返します。

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` – describe バッファが `a_v4_extfn_estimate` のサイズでない場合に返る set エラー。
- `EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER` – 指定の引数が引数 0 の場合に返る set エラー。
- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` – 状態が最適化状態でない場合に返る set エラー。
- `EXTFNAPIV4_DESCRIBE_INVALID_ATTRIBUTE_VALUE` – UDF が設定している値が 0 と 1 以外の場合に返る set エラー。

クエリ処理の状態
以下のときに有効です。

- 最適化状態

`_describe_extfn` API 関数を使用する PROCEDURE 定義とコードです。

```
CREATE PROCEDURE my_tpf( col_char char(10), col_table TABLE( c1 INT,
c2
INT ) )
      RESULTS ( r1 INT, r2 INT, r3 INT )
      EXTERNAL ‘my_tpf_proc@mylibrary’;

CREATE TABLE T( x INT, y INT, z INT );

select r2,r3 from my_tpf( 'test', TABLE( select x,y from T ) )
```

TPF を実行するときに、この TPF がカラム y を使用するかどうかを把握できるとサーバにとって便利です。TPF が y を必要としない場合には、サーバはそのことを最適化に活用でき、このカラムの情報を TPF に送信しません。

```
my_tpf_describe(a_v4_extfn_proc_context *cntxt)
{
    if( cntxt->current_state == EXTFNAPIV4_STATE_OPTIMIZATION ) {
      a_sql_byte  col_is_used = 0;
      a_sql_int32 ret = 0;

      ret = cntxt->describe_column_get( cntxt, 2, 2,
            EXTFNAPIV4_DESCRIBE_COL_IS_USED_BY_CONSUMER,
            &col_is_used,
```

```
            sizeof(a_sql_byte) );

      if( ret < 0 ) {
          // Handle the error.
      }

    }
}
```

参照：

- EXTFNAPIV4_DESCRIBE_COL_IS_USED_BY_CONSUMER (get) (240 ページ)
- describe_column の一般的なエラー (353 ページ)
- クエリ処理の状態 (135 ページ)

EXTFNAPIV4_DESCRIBE_COL_MINIMUM_VALUE (set)

EXTFNAPIV4_DESCRIBE_COL_MINIMUM_VALUE 属性は、カラムの最小値を示します。`describe_column_set` のシナリオで使用します。

データ型
`an_extfn_value`

説明
カラムが持つことのできる最小値です (使用可能な場合)。引数 0 に対してのみ有効です。

使用法
UDF は、カラムのデータの最小値を把握している場合には、`EXTFNAPIV4_DESCRIBE_COL_MINIMUM_VALUE` を設定できます。サーバはこの情報を、最適化のときに使用できます。

UDF は、`EXTFNAPIV4_DESCRIBE_COL_TYPE` を使用してカラムのデータ型を判断することと、`EXTFNAPIV4_DESCRIBE_COL_WIDTH` を使用してカラムの記憶領域の要件を判断することによって、設定する値を過不足のないサイズで保持するバッファを提供できます。

戻り値
成功した場合は、`describe_buffer_length` を返すか、または次を返します。

- `EXTFNAPIV4_DESCRIBE_NOT_AVAILABLE` – 属性を設定できない。 対象のカラムがクエリに含まれていない場合か、対象のカラムの最小値を使用できない場合に返る。

失敗した場合は、汎用の `describe_column` エラーのいずれかか、次のいずれかを返します。

- EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH – describe バッファの大きさが、最小値を保持できる十分なサイズでない場合に返る set エラー。
- EXTFNAPIV4_DESCRIBE_INVALID_STATE – 状態が最適化状態でない場合に返る set エラー。
- EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER – **arg_num** が 0 でない場合に返る set エラー。

クエリ処理の状態
以下のときに有効です。

- 最適化状態

例
_describe_extfn コールバック API 関数を実装する **PROCEDURE** 定義と UDF コードです。

```
CREATE PROCEDURE my_tpf( col_char char(10), col_table TABLE( c1 INT,
c2 INT ) )
    RESULTS ( r1 INT, r2 INT, r3 INT )
    EXTERNAL ‘my_tpf_proc@mylibrary’;

CREATE TABLE T( x INT, y INT, z INT );

select * from my_tpf( 'test', TABLE( select x,y from T ) )
```

この例は、結果セットのカラム 1 の最小値を把握することがサーバにとって役に立つ (または、この TPF の結果を入力として使用する別の TPF にとって役に立つ) TPF の例です。この例では、カラム 1 の最小の出力値は 27 です。

```
my_tpf_describe(a_v4_extfn_proc_context *cntxt)
{
    if( cntxt->current_state == EXTFNAPIV4_STATE_OPTIMIZATION ) {
    a_sql_int32 min_value = 27;
    a_sql_int32 ret = 0;

// Tell the server what the minimum value of the first column
// of our result set will be.

    ret = cntxt->describe_column_set( cntxt, 0, 1
        EXTFNAPIV4_DESCRIBE_COL_MINIMUM_VALUE,
        &min_value,
        sizeof(a_sql_int32) );

    if( ret < 0 ) {
        // Handle the error.
    }
}
}
```

参照：
- クエリ処理の状態 (135 ページ)
- EXTFNAPIV4_DESCRIBE_COL_MINIMUM_VALUE (get) (241 ページ)
- EXTFNAPIV4_DESCRIBE_COL_TYPE (set) (249 ページ)
- EXTFNAPIV4_DESCRIBE_COL_TYPE (get) (231 ページ)
- EXTFNAPIV4_DESCRIBE_COL_WIDTH (set) (250 ページ)
- EXTFNAPIV4_DESCRIBE_COL_WIDTH (get) (231 ページ)
- describe_column の一般的なエラー (353 ページ)

EXTFNAPIV4_DESCRIBE_COL_MAXIMUM_VALUE (set)

EXTFNAPIV4_DESCRIBE_COL_MAXIMUM_VALUE 属性は、カラムの最大値を示します。`describe_column_set` のシナリオで使用します。

データ型
`an_extfn_value`

説明
カラムの最大値。このプロパティは引数 0 およびテーブル引数に対してのみ有効です。

使用法
UDF は、カラムのデータの最大値を把握している場合には、`EXTFNAPIV4_DESCRIBE_COL_MAXIMUM_VALUE` を設定できます。サーバはこの情報を、最適化のときに使用できます。

UDF は、`EXTFNAPIV4_DESCRIBE_COL_TYPE` を使用してカラムのデータ型を判断することと、`EXTFNAPIV4_DESCRIBE_COL_WIDTH` を使用してカラムの記憶領域の要件を判断することによって、設定する値を過不足のないサイズで保持するバッファを提供できます。

`describe_buffer_length` はこのバッファの `sizeof()` です。

戻り値
成功した場合、値を設定したときには `describe_buffer_length` を返します。それ以外のときには次を返します。

- `EXTFNAPIV4_DESCRIBE_NOT_AVAILABLE` – 属性を設定できなかった。 対象のカラムがクエリに含まれていない場合か、対象のカラムの最大値を使用できない場合に返る。

失敗した場合は、汎用の `describe_column` エラーのいずれかか、次のいずれかを返します。

- EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH – describe バッファの大きさが、最大値を保持できる十分なサイズでない場合に返る set エラー。
- EXTFNAPIV4_DESCRIBE_INVALID_STATE – クエリ処理の状態が最適化状態でない場合に返る set エラー。
- EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER – **arg_num** が 0 でない場合に返る set エラー。

クエリ処理の状態
以下のときに有効です。

- 最適化状態

例
_describe_extfn コールバック API 関数を実装する PROCEDURE 定義と UDF コードです。

```
CREATE PROCEDURE my_tpf( col_char char(10), col_table TABLE( c1 INT,
c2 INT ) )
         RESULTS ( r1 INT, r2 INT, r3 INT )
         EXTERNAL ‘my_tpf_proc@mylibrary’;

CREATE TABLE T( x INT, y INT, z INT );

select * from my_tpf( 'test', TABLE( select x,y from T ) )
```

この例は、結果セットのカラム 1 の最大値を把握することがサーバにとって役に立つ (または、この TPF の結果を入力として使用する別の TPF にとって役に立つ) TPF の例です。この例では、カラム 1 の最大の出力値は 500000 です。

```
my_tpf_describe(a_v4_extfn_proc_context *cntxt)
{
    if( cntxt->current_state == EXTFNAPIV4_STATE_OPTIMIZATION ) {
         a_sql_int32 max_value = 500000;
         a_sql_int32 ret = 0;

        // Tell the server what the maximum value of the first column
         // of our result set will be.

      ret = cntxt->describe_column_set( cntxt, 0, 1
                  EXTFNAPIV4_DESCRIBE_COL_MAXIMUM_VALUE,
                  &max_value,
                  sizeof(a_sql_int32) );

         if( ret < 0 ) {
             // Handle the error.
         }

   }
}
```

参照：
- クエリ処理の状態 (135 ページ)
- EXTFNAPIV4_DESCRIBE_COL_MAXIMUM_VALUE (get) (244 ページ)
- EXTFNAPIV4_DESCRIBE_COL_TYPE (get) (231 ページ)
- EXTFNAPIV4_DESCRIBE_COL_TYPE (set) (249 ページ)
- EXTFNAPIV4_DESCRIBE_COL_WIDTH (get) (231 ページ)
- EXTFNAPIV4_DESCRIBE_COL_WIDTH (set) (250 ページ)
- describe_column の一般的なエラー (353 ページ)

EXTFNAPIV4_DESCRIBE_COL_VALUES_SUBSET_OF_INPUT (set)

EXTFNAPIV4_DESCRIBE_COL_VALUES_SUBSET_OF_INPUT 属性では、入力カラムが示す値のサブセットを設定します。`describe_column_set` のシナリオで使用します。

データ型
`a_v4_extfn_col_subset_of_input`

説明
カラム値は入力カラムが示す値のサブセットです。

使用法
この describe 属性を設定すると、指定のカラムの値が入力カラムで指定された値のサブセットであることをクエリ・オプティマイザに伝えることができます。たとえば、テーブルを利用し、関数に基づいてローをフィルタリングするフィルタ TPF があるとします。この場合、出力テーブルは入力テーブルのサブセットです。このフィルタ TPF で **EXTFNAPIV4_DESCRIBE_COL_VALUES_SUBSET_OF_INPUT** を設定するとクエリを最適化できます。

戻り値
成功した場合は、`sizeof(a_v4_extfn_col_subset_of_input)` を返します。

失敗した場合は、汎用の `describe_column` エラーのいずれかか、次のいずれかを返します。

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` – バッファの長さが `sizeof (a_v4_extfn_col_subset_of_input)` 未満の場合に返る set エラー。
- `EXTFNAPIV4_DESCRIBE_INVALID_ATTRIBUTE_VALUE` – ソース・テーブルのカラムのインデックスが範囲外の場合に返る set エラー。

- EXTFNAPIV4_DESCRIBE_NOT_AVAILABLE – subset_of_input をオンに設定したカラムが使用できない場合 (たとえば、そのカラムが select リストに含まれていない場合) に返る set エラー。
- EXTFNAPIV4_DESCRIBE_INVAILD_STATE – クエリ処理の状態が最適化状態でない場合に返る set エラー。
- EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH – バッファの長さが 0 の場合に返る set エラー。
- EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER – 出力テーブル以外のパラメータに対して呼び出した場合に返る set エラー。

クエリ処理の状態
以下のときに有効です。

- 最適化状態

例

```
a_v4_extfn_col_subset_of_input colMap;

    colMap.source_table_parameter_arg_num = 4;
    colMap.source_column_number = i;

    desc_rc = ctx->describe_column_set( ctx,
        0, i,
        EXTFNAPIV4_DESCRIBE_COL_VALUES_SUBSET_OF_INPUT,
        &colMap, sizeof(a_v4_extfn_col_subset_of_input) );
```

参照：

- EXTFNAPIV4_DESCRIBE_COL_VALUES_SUBSET_OF_INPUT (get) (246 ページ)
- describe_column の一般的なエラー (353 ページ)
- クエリ処理の状態 (135 ページ)

*describe_parameter_get

v4 API メソッド describe_parameter_get では、UDF のパラメータのプロパティをサーバから取得します。

宣言

```
a_sql_int32 (SQL_CALLBACK *describe_parameter_get)(
            a_v4_extfn_proc_context             *cntxt,
            a_sql_uint32                        arg_num,
            a_v4_extfn_describe_udf_type        describe_type,
            const void                          *describe_buffer,
           size_t                             describe_buffer_len );
```

パラメータ

パラメータ	説明
cntxt	プロシージャ・コンテキスト・オブジェクト。
arg_num	テーブル・パラメータの順序数 (0 は結果テーブル、1 は最初の入力引数)。
describe_type	設定するプロパティを示すセレクタ。
describe_buffer	サーバで設定する、指定のプロパティの describe 情報を保持する構造体。具体的な構造体またはデータ型は **describe_type** パラメータで示されます。
describe_buffer_length	**describe_buffer** のバイト単位の長さ。

戻り値
成功した場合は、0 または **describe_buffer** に書き込まれたバイト数を返します。値 0 の場合、サーバは属性を取得できなかったが、エラー状態は発生しなかったことを示します。エラーが発生した場合や、プロパティが取得されなかった場合は、汎用の **describe_parameter** エラーのいずれかを返します。

***describe_parameter_get の属性**
`describe_parameter_get` の属性を示すコードです。

```
typedef enum a_v4_extfn_describe_parm_type {
    EXTFNAPIV4_DESCRIBE_PARM_NAME,
    EXTFNAPIV4_DESCRIBE_PARM_TYPE,
    EXTFNAPIV4_DESCRIBE_PARM_WIDTH,
    EXTFNAPIV4_DESCRIBE_PARM_SCALE,
    EXTFNAPIV4_DESCRIBE_PARM_CAN_BE_NULL,
    EXTFNAPIV4_DESCRIBE_PARM_DISTINCT_VALUES,
    EXTFNAPIV4_DESCRIBE_PARM_IS_CONSTANT,
    EXTFNAPIV4_DESCRIBE_PARM_CONSTANT_VALUE,

    EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_COLUMNS,
    EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_ROWS,
    EXTFNAPIV4_DESCRIBE_PARM_TABLE_ORDERBY,
    EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY,
    EXTFNAPIV4_DESCRIBE_PARM_TABLE_REQUEST_REWIND,
    EXTFNAPIV4_DESCRIBE_PARM_TABLE_HAS_REWIND,
    EXTFNAPIV4_DESCRIBE_PARM_TABLE_UNUSED_COLUMNS,

} a_v4_extfn_describe_parm_type;
```

EXTFNAPIV4_DESCRIBE_PARM_NAME 属性 (get)
EXTFNAPIV4_DESCRIBE_PARM_NAME 属性は、パラメータ名を示します。describe_parameter_get のシナリオで使用します。

データ型
char[]

説明
UDF に対するパラメータの名前。

使用法
CREATE PROCEDURE 文で定義されているパラメータ名を取得します。パラメータ 0 に対しては無効です。

戻り値
成功した場合は、パラメータ名の長さを返します。

失敗した場合は、汎用の describe_parameter エラーのいずれかか、次のいずれかを返します。

- EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH – **describe_buffer** が名前を保持できる十分な大きさでない場合に返る get エラー。
- EXTFNAPIV4_DESCRIBE_INVALID_STATE – 状態が初期状態より後でない場合に返る get エラー。
- EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER – パラメータが結果テーブルの場合に返る get エラー。

クエリ処理の状態
以下のときに有効です。

- 注釈状態
- クエリ最適化状態
- プラン構築状態
- 実行状態

参照：
- EXTFNAPIV4_DESCRIBE_PARM_NAME 属性 (set) (287 ページ)
- describe_parameter の一般的なエラー (354 ページ)
- クエリ処理の状態 (135 ページ)

EXTFNAPIV4_DESCRIBE_PARM_TYPE 属性 (get)

EXTFNAPIV4_DESCRIBE_PARM_TYPE 属性は、`describe_parameter_get` のシナリオでデータ型を返します。

データ型
`a_sql_data_type`

説明
UDF に対するパラメータのデータ型。

使用法
CREATE PROCEDURE 文で定義されているパラメータのデータ型を取得します。

戻り値
成功した場合は、`sizeof(a_sql_data_type)` を返します。

失敗した場合は、汎用の `describe_parameter` エラーのいずれかか、次のいずれかを返します。

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` – **describe_buffer** が `sizeof(a_sql_data_type)` でない場合に返る get エラー。
- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` – 状態が初期状態より後でない場合に返る get エラー。

クエリ処理の状態
以下のときに有効です。

- 注釈状態
- クエリ最適化状態
- プラン構築状態
- 実行状態

参照：

- EXTFNAPIV4_DESCRIBE_PARM_TYPE 属性 (set) (288 ページ)
- describe_parameter の一般的なエラー (354 ページ)
- クエリ処理の状態 (135 ページ)

EXTFNAPIV4_DESCRIBE_PARM_WIDTH 属性 (get)
EXTFNAPIV4_DESCRIBE_PARM_WIDTH 属性は、パラメータの幅を示します。 `describe_parameter_get` のシナリオで使用します。

データ型
`a_sql_uint32`

説明
UDF に対するパラメータの幅。`EXTFNAPIV4_DESCRIBE_PARM_WIDTH` はスカラ・パラメータにのみ該当します。パラメータ幅は、対応するデータ型のパラメータを格納するために必要な記憶領域のサイズのバイト数です。

- **固定長のデータ型** – データの格納に必要なバイト数
- **可変長のデータ型** – 最大長
- **LOB データ型** – データへのハンドルの格納に必要な記憶領域のサイズ
- **TIME データ型** – コード化した時間の格納に必要な記憶領域のサイズ

使用法
CREATE PROCEDURE 文で定義されているパラメータの幅を取得します。

戻り値
成功した場合は、`sizeof(a_sql_uint32)` を返します。

失敗した場合は、汎用の `describe_parameter` エラーのいずれかか、次のいずれかを返します。

- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` – クエリの処理状態が初期状態より後でない場合に返る get エラー。
- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` – **describe_buffer** が `a_sql_uint32` のサイズでない場合に返る get エラー。
- `EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER` – 指定のパラメータがテーブル・パラメータの場合に返る get エラー。 これには、パラメータ 0 やパラメータ *n* (*n* が入力テーブルの場合) が含まれます。

クエリ処理の状態
以下のときに有効です。

- 注釈状態
- クエリ最適化状態
- プラン構築状態
- 実行状態

例
プロシージャ定義の例です。

```
CREATE PROCEDURE my_udf(IN p1 INT, IN p2 char(100))
RESULT (x INT)
EXTERNAL NAME ‘my_udf@myudflib’;
```

_describe_extfn API 関数のコードの例です。

```
my_udf_describe(a_v4_extfn_proc_context *cntxt)
{
    if( cntxt->current_state == EXTFNAPIV4_STATE_OPTIMIZATION ) {
    a_sql_uint32 width = 0;
    a_sql_int32 ret = 0;

    // Get the width of parameter 1
    ret = cntxt->describe_parameter_get( cntxt, 1,
    EXTFNAPIV4_DESCRIBE_PARM_WIDTH,
    &width,
    sizeof(a_sql_uint32) );

    if( ret < 0 ) {
        // Handle the error.
    }

    //Allocate some storage based on parameter width
    a_sql_byte *p = (a_sql_byte *)cntxt->alloc( cntxt, width )

    // Get the width of parameter 2
    ret = cntxt->describe_parameter_get( cntxt, 2,
        EXTFNAPIV4_DESCRIBE_PARM_WIDTH,
        &width,
        sizeof(a_sql_uint32) );
    if( ret <= 0 ) {
        // Handle the error.
    }

    // Allocate some storage based on parameter width
    char *c = (char *)cntxt->alloc( cntxt, width )

    …

}
}
```

参照：

- EXTFNAPIV4_DESCRIBE_PARM_WIDTH 属性 (set) (289 ページ)
- describe_parameter の一般的なエラー (354 ページ)
- クエリ処理の状態 (135 ページ)

EXTFNAPIV4_DESCRIBE_PARM_SCALE 属性 (get)
EXTFNAPIV4_DESCRIBE_PARM_SCALE 属性は、パラメータの位取りを示します。`describe_parameter_get` のシナリオで使用します。

データ型
`a_sql_uint32`

説明
UDF に対するパラメータの位取り。算術データ型については、パラメータの位取りは、数値の小数点の右側の桁数です。

この属性は以下に対しては無効です。

- 算術データ型以外のデータ型
- テーブル・パラメータ

使用法
CREATE PROCEDURE 文で定義されているパラメータの位取りを取得します。

戻り値
成功した場合は、`(a_sql_uint32)` のサイズを返します。

失敗した場合は、汎用の `describe_parameter` エラーのいずれかか、次のいずれかを返します。

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` – **describe_buffer** が `a_sql_uint32` のサイズでない場合に返る get エラー。
- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` – 状態が初期状態より後でない場合に返る get エラー。
- `EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER` – 指定のパラメータがテーブル・パラメータの場合に返る get エラー。 これには、パラメータ 0 やパラメータ *n* (*n* が入力テーブルの場合) が含まれます。

クエリ処理の状態
以下のときに有効です。

- 注釈状態
- クエリ最適化状態
- プラン構築状態
- 実行状態

例
パラメータ 1 の位取りを取得する `_describe_extfn` API 関数のコードの例です。

```
if( cntxt->current_state > EXTFNAPIV4_STATE_ANNOTATION ) {
    a_sql_uint32 scale = 0;
    a_sql_int32 ret = 0;

    ret = ctx->describe_parameter_get( ctx, 1,
EXTFNAPIV4_DESCRIBE_PARM_SCALE,
    &scale, sizeof(a_sql_uint32) );

    if( ret <= 0 ) {
    // Handle the error.
    }
}
```

参照：

- EXTFNAPIV4_DESCRIBE_PARM_SCALE 属性 (set) (290 ページ)
- describe_parameter の一般的なエラー (354 ページ)
- クエリ処理の状態 (135 ページ)

EXTFNAPIV4_DESCRIBE_PARM_CAN_BE_NULL 属性 (get)

EXTFNAPIV4_DESCRIBE_PARM_CAN_BE_NULL 属性は、パラメータが NULL かどうかを示します。`describe_parameter_get` のシナリオで使用します。

データ型
`a_sql_byte`

説明
パラメータの値が実行時に NULL になり得る場合は true。テーブル・パラメータまたはパラメータ 0 については、値は false です。

使用法
クエリの実行中に、指定のパラメータが NULL の可能性があるかどうかを取得します。

戻り値
成功した場合は、`sizeof(a_sql_byte)` を返します。

失敗した場合は、汎用の `describe_parameter` エラーのいずれかか、次のいずれかを返します。

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` – **describe_buffer** が `a_sql_byte` のサイズでない場合に返る get エラー。
- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` – 状態が初期状態より後でない場合に返る get エラー。

クエリ処理の状態
以下のときに有効です。

- 実行状態

例： EXTFNAPIV4_DESCRIBE_PARM_CAN_BE_NULL (get)
プロシージャ定義、_describe_extfn API 関数のコード、**EXTFNAPIV4_DESCRIBE_PARM_CAN_BE_NULL** の値を取得する SQL クエリの例です。

プロシージャ定義
この項のサンプル・クエリで使用するプロシージャ定義の例を示します。

```
CREATE PROCEDURE my_udf(IN p INT)
RESULT (x INT)
EXTERNAL NAME ‘my_udf@myudflib’;
```

API 関数のコード
この項のサンプル・クエリで使用する _describe_extfn API 関数のコードの例を示します。

```
my_udf_describe(a_v4_extfn_proc_context *cntxt)
{

    if( cntxt->current_state > EXTFNAPIV4_STATE_OPTIMIZATION ) {
    a_sql_byte can_be_null = 0;
    a_sql_int32 ret = 0;

    ret = cntxt->describe_parameter_get( cntxt, 1,
    EXTFNAPIV4_DESCRIBE_PARM_CAN_BE_NULL,
    &can_be_null,
    sizeof(a_sql_byte) );

    if( ret <= 0 ) {
        // Handle the error.
    }
}
}
```

例 1： NOT NULL なしの場合
この例では、単一の整数カラムを持つテーブルを、**NOT NULL** 変更子を指定せずに作成しています。相関サブクエリではテーブル has_nulls のカラム *c1* を渡しています。実行状態のときにプロシージャ **my_udf_describe** が呼び出されると、describe_parameter_get の呼び出しで **can_be_null** は値 1 に設定されます。

```
CREATE TABLE has_nulls ( c1 INT );
INSERT INTO has_nulls VALUES(1);
INSERT INTO has_nulls VALUES(NULL);
SELECT * from has_nulls WHERE (SELECT sum(my_udf.x) FROM
my_udf(has_nulls.c1)) > 0;
```

例 2： NOT NULL ありの場合
この例では、単一の整数カラムを持つテーブルを、**NOT NULL** 変更子を指定して作成しています。相関サブクエリではテーブル no_nulls のカラム *c1* を渡しています。実行状態のときにプロシージャ **my_udf_describe** が呼び出されると、describe_parameter_get の呼び出しで **can_be_null** は値 0 に設定されます。

```
CREATE TABLE no_nulls ( c1 INT NOT NULL);
INSERT INTO no_nulls VALUES(1);
INSERT INTO no_nulls VALUES(2);
SELECT * from no_nulls WHERE (SELECT sum(my_udf.x) FROM
my_udf(no_nulls.c1)) > 0;
```

例 3： 定数の場合
この例では、プロシージャ **my_udf** を定数で呼び出しています。実行状態のときにプロシージャ **my_udf_describe** が呼び出されると、describe_parameter_get の呼び出しで **can_be_null** は値 0 に設定されます。

```
SELECT * from my_udf(5);
```

例 4： NULL の場合
この例では、プロシージャ **my_udf** を NULL で呼び出しています。実行状態のときにプロシージャ **my_udf_describe** が呼び出されると、describe_parameter_get の呼び出しで **can_be_null** は値 1 に設定されます。

```
SELECT * from my_udf(NULL);
```

EXTFNAPIV4_DESCRIBE_PARM_DISTINCT_VALUES 属性 (get)
EXTFNAPIV4_DESCRIBE_PARM_DISTINCT_VALUES 属性は、重複しない値の数を返します。describe_parameter_get のシナリオで使用します。

データ型
a_v4_extfn_estimate

説明
すべての呼び出し全体での重複しない値の推定数を返します。スカラ・パラメータに対してのみ有効です。

使用法
この情報を使用できる場合、UDF は重複しない値の推定数を 100% の確度で返します。この情報を使用できない場合、UDF は推定数 0 を 0% の確度で返します。

戻り値
成功した場合は、sizeof(a_v4_extfn_estimate) を返します。

失敗した場合は、汎用の describe_parameter エラーのいずれかか、次のいずれかを返します。

- EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH – **describe_buffer** が a_v4_extfn_estimate のサイズでない場合に返る get エラー。
- EXTFNAPIV4_DESCRIBE_INVALID_STATE – 状態が初期状態より後でない場合に返る get エラー。
- EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER – パラメータがテーブル・パラメータの場合に返る get エラー。

クエリ処理の状態
以下のときに有効です。

- 注釈状態
- クエリ最適化状態
- プラン構築状態
- 実行状態

例
_describe_extfn API 関数のコードの例です。

```
if( ctx->current_state >= EXTFNAPIV4_STATE_ANNOTATION ) {
    desc_est.value = 0.0;
    desc_est.confidence = 0.0;

    desc_rc = ctx->describe_parameter_get( ctx,
        1,
        EXTFNAPIV4_DESCRIBE_PARM_DISTINCT_VALUES,
        &desc_est, sizeof(a_v4_extfn_estimate) );
}
```

参照：

- EXTFNAPIV4_DESCRIBE_PARM_DISTINCT_VALUES 属性 (set) (291 ページ)
- EXTFNAPIV4_DESCRIBE_PARM_TYPE 属性 (get) (267 ページ)
- describe_parameter の一般的なエラー (354 ページ)
- クエリ処理の状態 (135 ページ)

EXTFNAPIV4_DESCRIBE_PARM_IS_CONSTANT 属性 (get)

EXTFNAPIV4_DESCRIBE_PARM_DISTINCT_VALUES 属性は、パラメータが定数かどうかを返します。describe_parameter_get のシナリオで使用します。

データ型
a_sql_byte

説明
文のパラメータが定数の場合は true。スカラ・パラメータに対してのみ有効です。

使用法
指定のパラメータの値が定数でない場合は 0 を返します。指定のパラメータの値が定数の場合は 1 を返します。

戻り値
成功した場合は、`sizeof(a_sql_byte)` を返します。

失敗した場合は、汎用の `describe_parameter` エラーのいずれかか、次のいずれかを返します。

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` – **describe_buffer** が `a_sql_byte` のサイズでない場合に返る get エラー。
- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` – 状態が初期状態より後でない場合に返る get エラー。
- `EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER` – パラメータがテーブル・パラメータの場合に返る get エラー。

クエリ処理の状態
以下のときに有効です。

- 注釈状態
- クエリ最適化状態
- プラン構築状態
- 実行状態

例
`_describe_extfn` API 関数のコードの例です。

```
if( ctx->current_state >= EXTFNAPIV4_STATE_ANNOTATION ) {
    desc_rc = ctx->describe_parameter_get( ctx,
        1,
        EXTFNAPIV4_DESCRIBE_PARM_IS_CONSTANT,
        &desc_byte, sizeof( a_sql_byte ) );
}
```

参照：

- EXTFNAPIV4_DESCRIBE_PARM_IS_CONSTANT 属性 (set) (292 ページ)
- EXTFNAPIV4_DESCRIBE_PARM_TYPE 属性 (set) (288 ページ)
- describe_parameter の一般的なエラー (354 ページ)
- クエリ処理の状態 (135 ページ)

EXTFNAPIV4_DESCRIBE_PARM_CONSTANT_VALUE 属性 (get)
EXTFNAPIV4_DESCRIBE_PARM_CONSTANT_VALUE 属性は、パラメータの値を示します。`describe_parameter_get` のシナリオで使用します。

データ型
`an_extfn_value`

説明
パラメータの値 (describe の時点で把握している場合)。スカラ・パラメータに対してのみ有効です。

使用法
パラメータの値を返します。

戻り値
成功した場合、値を使用できるときには `sizeof(an_extfn_value)` を返します。その他のときには次を返します。

- `EXTFNAPIV4_DESCRIBE_NOT_AVILABLE` – 値が定数でない場合に返る値。

失敗した場合は、汎用の `describe_parameter` エラーのいずれかか、次のいずれかを返します。

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` – **describe_buffer** が `an_extfn_value` のサイズでない場合に返る get エラー。
- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` – 状態が初期状態より後でない場合に返る get エラー。
- `EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER` – パラメータがテーブル・パラメータの場合に返る get エラー。

クエリ処理の状態
以下のときに有効です。

- 注釈状態
- クエリ最適化状態
- プラン構築状態
- 実行状態

例
`_describe_extfn` API 関数のコードの例です。

```
if( ctx->current_state >= EXTFNAPIV4_STATE_ANNOTATION ) {
    a_sql_int32 desc_rc;
    desc_rc = ctx->describe_parameter_get( ctx,
```

```
        1,
        EXTFNAPIV4_DESCRIBE_PARM_CONSTANT_VALUE,
        &arg,
        sizeof( an_extfn_value ) );
}
```

参照：

- EXTFNAPIV4_DESCRIBE_PARM_IS_CONSTANT 属性 (set) (292 ページ)
- EXTFNAPIV4_DESCRIBE_PARM_TYPE 属性 (get) (267 ページ)
- describe_parameter の一般的なエラー (354 ページ)
- クエリ処理の状態 (135 ページ)

EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_COLUMNS 属性 (get)

EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_COLUMNS 属性は、テーブル内のカラム数を示します。describe_parameter_get のシナリオで使用します。

データ型
a_sql_uint32

説明
テーブル内のカラム数。引数 0 およびテーブル引数に対してのみ有効です。

使用法
指定のテーブル引数に含まれるカラム数を返します。引数 0 は、結果テーブルに含まれるカラム数を返します。

戻り値
成功した場合は、sizeof(a_sql_uint32) を返します。

失敗した場合は、汎用の describe_parameter エラーのいずれかか、次のいずれかを返します。

- EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH – **describe_buffer** が sizeof a_sql_uint32 のサイズでない場合に返る get エラー。
- EXTFNAPIV4_DESCRIBE_INVALID_STATE – 状態が初期状態より後でない場合に返る get エラー。
- EXTFNAPIV4_DESCRIBE_NON_TABLE_PARAMETER – パラメータがテーブル・パラメータでない場合に返る get エラー。

クエリ処理の状態
以下のときに有効です。

- 注釈状態

- クエリ最適化状態
- プラン構築状態
- 実行状態

参照：

- EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_COLUMNS 属性 (set) (293 ページ)
- クエリ処理の状態 (135 ページ)

EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_ROWS 属性 (get)

`EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_ROWS` 属性は、テーブル内のロー数を示します。`describe_parameter_get` のシナリオで使用します。

データ型
`a_v4_extfn_estimate`

説明
テーブルの推定ロー数。引数 0 およびテーブル引数に対してのみ有効です。

使用法
指定のテーブル引数または結果セットに含まれるローの推定数を 100% の確度で返します。

戻り値
成功した場合は、`a_v4_extfn_estimate` のサイズを返します。

失敗した場合は、汎用の `describe_parameter` エラーのいずれかか、次のいずれかを返します。

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` – **describe_buffer** が `a_v4_extfn_estimate` のサイズでない場合に返る get エラー。
- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` – 状態が初期状態より後でない場合に返る get エラー。
- `EXTFNAPIV4_DESCRIBE_NON_TABLE_PARAMETER` – パラメータがテーブル・パラメータでない場合に返る get エラー。

クエリ処理の状態
以下のときに有効です。

- 注釈状態
- クエリ最適化状態
- プラン構築状態

- 実行状態

参照：
- EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_ROWS 属性 (set) (294 ページ)
- クエリ処理の状態 (135 ページ)

EXTFNAPIV4_DESCRIBE_PARM_TABLE_ORDERBY 属性 (get)
`EXTFNAPIV4_DESCRIBE_PARM_TABLE_ORDERBY` 属性は、テーブル内のローの順序を示します。`describe_parameter_get` のシナリオで使用します。

データ型
`a_v4_extfn_orderby_list`

説明
テーブル内のローの順序。このプロパティは引数 0 およびテーブル引数に対してのみ有効です。

使用法
この属性を使用することによって、UDF コードは以下が可能になります。

- 入力の **TABLE** パラメータが順序付けされているかどうかを判断する。
- 結果セットが順序付けされていることを宣言する。

パラメータ番号が 0 の場合、この属性は出力の結果セットを表します。パラメータが 0 より大きく、パラメータの種類がテーブルの場合、この属性は入力の **TABLE** パラメータを表します。

順序付けは `a_v4_extfn_orderby_list` で示されます。これは、カラムの順序数と、それに対応する昇順または降順のプロパティのリストを保持する構造体です。UDF が出力の結果セットに対してこの順序付けのプロパティを設定した場合、サーバは順序付けの最適化を実行できます。たとえば、UDF が結果セットの最初のカラムを昇順で生成した場合、サーバは同じカラムに対する重複した順序付けの要求を省略できます。

UDF が出力の結果セットに対して順序付けのプロパティを設定しない場合、サーバはデータが順序付けされていないものと見なします。

UDF が入力の **TABLE** パラメータに順序付けのプロパティを設定する場合、その入力データに対する順序付けはサーバが保証します。このシナリオでは、入力データの順序付けが必要であることを UDF がサーバに伝達します。サーバは、実行時に競合を検出した場合は SQL 例外を発生させます。たとえば、入力の **TABLE** パラメータの最初のカラムは昇順である必要があると UDF が伝達したのに対し、SQL 文に降順を指定する句が含まれている場合、サーバは SQL 例外を発生させます。

SQL に順序付けの句が含まれていない場合、サーバは順序付けを自動的に追加します。これにより、入力の **TABLE** パラメータが要件どおりに順序付けされます。

戻り値
成功した場合は、`a_v4_extfn_orderby_list` からコピーされたバイト数を返します。

クエリ処理の状態
以下のときに有効です。

- 注釈状態
- クエリ最適化状態

参照：

- EXTFNAPIV4_DESCRIBE_PARM_TABLE_ORDERBY 属性 (set) (295 ページ)
- クエリ処理の状態 (135 ページ)

EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY (get)
`EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY` 属性は、UDF がパーティション分割を必要としていることを示します。`describe_parameter_get` のシナリオで使用します。

データ型
`a_v4_extfn_column_list`

説明
UDF 開発者は、**EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY** を使用して、呼び出しの前にパーティション分割が必要であることを UDF のコードで宣言します。

使用法
UDF は、パーティションの適用またはパーティション分割の動的な実施のための確認を実行できます。**a_v4_extfn_column_list** を確保することは UDF の役割です。そのときは、入力テーブルに含まれるカラムの総数を考慮に入れ、そのデータをサーバに送信します。

戻り値
成功した場合は、`a_v4_extfn_column_list` のサイズを返します。この値は次と同じです。

```
sizeof(a_v4_extfn_column_list) + sizeof(a_sql_uint32) *
number_of_partition_columns
```

失敗した場合は、汎用の `describe_parameter` エラーのいずれかか、次のいずれかを返します。

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` – バッファの長さが予期したサイズ未満の場合に返る get エラー。

クエリ処理の状態
以下のときに有効です。

- クエリ最適化状態
- プラン構築状態
- 実行状態

例

```
void UDF_CALLBACK my_tpf_proc_describe( a_v4_extfn_proc_context
*ctx )
{
    if( ctx->current_state == EXTFNAPIV4_STATE_OPTIMIZATION ) {
        a_sql_uint32    col_count   = 0;
        a_sql_uint32    buffer_size = 0;
        a_v4_extfn_column_list    *clist = NULL;

        col_count = 3;    // Set to the max number of possible pby
columns

       buffer_size = sizeof( a_v4_extfn_column_list ) + (col_count -
1) * sizeof( a_sql_uint32 );

        clist = (a_v4_extfn_column_list *)ctx->alloc( ctx,
buffer_size );

        clist->number_of_columns = 0;
        clist->column_indexes[0] = 0;
        clist->column_indexes[1] = 0;
        clist->column_indexes[2] = 0;

    args->r_api_rc = ctx->describe_parameter_get( ctx,
        args->p3_arg_num,
    EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY,
    clist,
    buffer_size );
}
}
```

参照：

- EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY (set) (297 ページ)
- describe_parameter の一般的なエラー (354 ページ)
- V4 API の describe_parameter と EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY (157 ページ)

- EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY を使用した並列処理 TPF の PARTITION BY の例 (160 ページ)
- クエリ処理の状態 (135 ページ)
- 入力データのパーティション分割 (157 ページ)

EXTFNAPIV4_DESCRIBE_PARM_TABLE_REQUEST_REWIND 属性 (get)
EXTFNAPIV4_DESCRIBE_PARM_TABLE_REQUEST_REWIND 属性は、コンシューマが入力テーブルのリワインドを要求していることを示します。describe_parameter_get のシナリオで使用します。

データ型
a_sql_byte

説明
コンシューマが入力テーブルのリワインドを希望していることを示します。テーブル入力引数に対してのみ有効です。デフォルトでは、このプロパティは false です。

使用法
UDF はこのプロパティを確認して true/false 値を取得します。

戻り値
成功した場合は、sizeof(a_sql_byte) を返します。

失敗した場合は、汎用の describe_parameter エラーのいずれかか、次のいずれかを返します。

- EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH – **describe_buffer** が a_sql_byte のサイズでない場合に返る get エラー。
- EXTFNAPIV4_DESCRIBE_INVALID_STATE – 状態が最適化状態とプラン構築状態のいずれでもない場合に返る get エラー。
- EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER – UDF がこの属性をパラメータ 0 に対して取得しようとした場合に返る get エラー。
- EXTFNAPIV4_DESCRIBE_NON_TABLE_PARAMETER – UDF がこの属性をテーブルでないパラメータに対して取得しようとした場合に返る get エラー。

クエリ処理の状態
以下のときに有効です。

- 最適化状態
- プラン構築状態

参照：

- EXTFNAPIV4_DESCRIBE_PARM_TABLE_REQUEST_REWIND 属性 (Set) (298 ページ)
- EXTFNAPIV4_DESCRIBE_PARM_TABLE_HAS_REWIND 属性 (set) (300 ページ)
- EXTFNAPIV4_DESCRIBE_PARM_TABLE_HAS_REWIND 属性 (get) (283 ページ)
- _rewind_extfn (350 ページ)
- クエリ処理の状態 (135 ページ)

EXTFNAPIV4_DESCRIBE_PARM_TABLE_HAS_REWIND 属性 (get)

`EXTFNAPIV4_DESCRIBE_PARM_TABLE_HAS_REWIND` 属性は、パラメータがリワインドをサポートしていることを示します。`describe_parameter_get` のシナリオで使用します。

データ型
`a_sql_byte`

説明
プロデューサがリワインドをサポートできるかどうかを示します。テーブル引数に対してのみ有効です。

`DESCRIBE_PARM_TABLE_HAS_REWIND` を `true` に設定する予定の場合は、リワインド・テーブル・コールバック (**_rewind_extfn()**) の実装も提供する必要があります。このコールバック・メソッドが提供されていない場合、サーバは UDF の実行に失敗します。

使用法
UDF は、テーブルの入力引数がリワインドをサポートしているかどうかを確認します。このプロパティを使用する前に、前提条件として、UDF は **DESCRIBE_PARM_TABLE_REQUEST_REWIND** を使用してリワインドを要求する必要があります。

戻り値
成功した場合は、`sizeof(a_sql_byte)` を返します。

失敗した場合は、汎用の `describe_parameter` エラーのいずれかか、次のいずれかを返します。

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` – **describe_buffer** が `a_sql_byte` のサイズでない場合に返る get エラー。

- EXTFNAPIV4_DESCRIBE_INVALID_STATE – 状態が注釈状態より後でない場合に返る get エラー。
- EXTFNAPIV4_DESCRIBE_NON_TABLE_PARAMETER – UDF がこの属性をテーブルでないパラメータに対して取得しようとした場合に返る get エラー。
- EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER – UDF がこの属性を結果テーブルに対して取得しようとした場合に返る get エラー。

クエリ処理の状態
以下のときに有効です。

- 最適化状態
- プラン構築状態
- 実行状態

参照：

- EXTFNAPIV4_DESCRIBE_PARM_TABLE_REQUEST_REWIND 属性 (get) (282 ページ)
- EXTFNAPIV4_DESCRIBE_PARM_TABLE_REQUEST_REWIND 属性 (Set) (298 ページ)
- EXTFNAPIV4_DESCRIBE_PARM_TABLE_HAS_REWIND 属性 (set) (300 ページ)
- _rewind_extfn (350 ページ)
- クエリ処理の状態 (135 ページ)

EXTFNAPIV4_DESCRIBE_PARM_TABLE_UNUSED_COLUMNS 属性 (get)

EXTFNAPIV4_DESCRIBE_PARM_TABLE_UNUSED_COLUMNS 属性は、利用しないカラムをリストします。describe_parameter_get のシナリオで使用します。

データ型
a_v4_extfn_column_list

説明
サーバまたは UDF が利用しない出力テーブル・カラムのリスト。

出力テーブル・パラメータについては、UDF は通常すべてのカラムに対してデータを生成し、サーバはすべてのカラムを利用します。入力テーブル・パラメータについても同様で、サーバは通常すべてのカラムに対してデータを生成し、UDF はすべてのカラムを利用します。

しかし場合によっては、サーバまたは UDF が一部のカラムを利用しないこともあります。そのような場合の最善の方法は、UDF が describe 属性 **EXTFNAPIV4_DESCRIBE_PARM_TABLE_UNUSED_COLUMNS** を使用して、出力テーブルに対する **GET** を実行することです。この処理は、出力テーブルのうちでサー

バが利用しないカラムのリストをサーバに問い合わせる処理です。UDF はこのリストを、出力テーブルのカラム・データを格納するときに利用できます。つまり、使用しないカラムに対するデータの格納を省略できます。

要約すると、出力テーブルについては、使用しないカラムのリストを UDF が問い合わせます。入力テーブルについては、使用しないカラムのリストを UDF がプッシュします。

使用法
UDF は、出力テーブルのすべてのカラムを利用するかどうかをサーバに問い合わせます。UDF は出力テーブルのすべてのカラムを含めた `a_v4_extfn_column_list` を確保してサーバに渡す必要があります。これを受けてサーバは、使用しないすべてのカラムの位置を 1 とマークします。サーバから返されたリストはデータを生成するときに使用できます。

戻り値
成功した場合は、カラム・リストのサイズを返します。
`sizeof(a_v4_extfn_column_list) + sizeof(a_sql_uint32) * number result columns`.

失敗した場合は、汎用の `describe_parameter` エラーのいずれかか、次のいずれかを返します。

- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` – 状態が初期状態より後でない場合に返る get エラー。
- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` – **describe_buffer** の大きさが、返されるリストを保持できる十分なサイズでない場合に返る get エラー。
- `EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER` – UDF がこの属性を入力テーブルに対して取得しようとした場合に返る get エラー。
- `EXTFNAPIV4_DESCRIBE_NON_TABLE_PARAMETER` – UDF がこの属性をテーブルでないパラメータに対して取得しようとした場合に返る get エラー。

クエリ処理の状態
以下のときに有効です。

- 実行状態

参照：

- EXTFNAPIV4_DESCRIBE_PARM_TABLE_UNUSED_COLUMNS 属性 (set) (301 ページ)

*describe_parameter_set

v4 API メソッド describe_parameter_set では、単一のパラメータについてのプロパティを UDF に対して設定します。

宣言

```
a_sql_int32 (SQL_CALLBACK *describe_parameter_set)(
            a_v4_extfn_proc_context             *cntxt,
            a_sql_uint32                        arg_num,
            a_v4_extfn_describe_udf_type        describe_type,
            const void                          *describe_buffer,
           size_t                             describe_buffer_len );
```

パラメータ

パラメータ	説明
cntxt	プロシージャ・コンテキスト・オブジェクト。
arg_num	テーブル・パラメータの順序数 (0 は結果テーブル、1 は最初の入力引数)。
describe_type	設定するプロパティを示すセレクタ。
describe_buffer	サーバで設定する、指定のプロパティの describe 情報を保持する構造体。具体的な構造体またはデータ型は **describe_type** パラメータで示されます。
describe_buffer_length	**describe_buffer** のバイト単位の長さ。

戻り値

成功した場合は、0 または **describe_buffer** に書き込まれたバイト数を返します。値 0 の場合、サーバは属性を設定できなかったが、エラー状態は発生しなかったことを示します。エラーが発生した場合や、プロパティが取得されなかった場合は、汎用の **describe_parameter** エラーのいずれかを返します。

*describe_parameter_set の属性

describe_parameter_set の属性を示すコードです。

```
typedef enum a_v4_extfn_describe_parm_type {
    EXTFNAPIV4_DESCRIBE_PARM_NAME,
    EXTFNAPIV4_DESCRIBE_PARM_TYPE,
    EXTFNAPIV4_DESCRIBE_PARM_WIDTH,
    EXTFNAPIV4_DESCRIBE_PARM_SCALE,
    EXTFNAPIV4_DESCRIBE_PARM_CAN_BE_NULL,
    EXTFNAPIV4_DESCRIBE_PARM_DISTINCT_VALUES,
```

```
    EXTFNAPIV4_DESCRIBE_PARM_IS_CONSTANT,
    EXTFNAPIV4_DESCRIBE_PARM_CONSTANT_VALUE,

    EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_COLUMNS,
    EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_ROWS,
    EXTFNAPIV4_DESCRIBE_PARM_TABLE_ORDERBY,
    EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY,
    EXTFNAPIV4_DESCRIBE_PARM_TABLE_REQUEST_REWIND,
    EXTFNAPIV4_DESCRIBE_PARM_TABLE_HAS_REWIND,
    EXTFNAPIV4_DESCRIBE_PARM_TABLE_UNUSED_COLUMNS,

    } a_v4_extfn_describe_parm_type;
```

EXTFNAPIV4_DESCRIBE_PARM_NAME 属性 (set)
EXTFNAPIV4_DESCRIBE_PARM_NAME 属性は、パラメータ名を示します。describe_parameter_set のシナリオで使用します。

データ型
char[]

説明
UDF に対するパラメータの名前。

使用法
UDF がこのプロパティを設定すると、サーバはその値を、**CREATE PROCEDURE** 文で指定されたパラメータ名と比較します。2 つの値が一致しない場合、サーバはエラーを返します。これによって UDF は、**CREATE PROCEDURE** 文のパラメータ名が UDF が予期するものと同じであることを確認できます。

戻り値
成功した場合は、パラメータ名の長さを返します。

失敗した場合は、汎用の describe_parameter エラーのいずれかか、次のいずれかを返します。

- EXTFNAPIV4_DESCRIBE_INVALID_STATE – 状態が注釈状態でない場合に返る set エラー。
- EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER – パラメータが結果テーブルの場合に返る set エラー。
- EXTFNAPI4_DESCRIBE_INVALID_ATTRIBUTE_VALUE – UDF が名前を再設定しようとした場合に返る set エラー。

クエリ処理の状態
以下のときに有効です。

- 注釈状態

参照：
- EXTFNAPIV4_DESCRIBE_PARM_NAME 属性 (get) (266 ページ)
- describe_parameter の一般的なエラー (354 ページ)
- クエリ処理の状態 (135 ページ)

EXTFNAPIV4_DESCRIBE_PARM_TYPE 属性 (set)
EXTFNAPIV4_DESCRIBE_PARM_TYPE 属性は、パラメータのデータ型を示します。describe_parameter_set のシナリオで使用します。

データ型
a_sql_data_type

説明
UDF に対するパラメータのデータ型。

使用法
UDF がこのプロパティを設定すると、サーバはその値を、**CREATE PROCEDURE** 文で指定されたパラメータの型と比較します。2 つの値が一致しない場合、サーバはエラーを返します。このチェックによって、**CREATE PROCEDURE** 文のパラメータのデータ型が UDF が予期するものと同じであることを確認できます。

戻り値
成功した場合は、sizeof(a_sql_data_type) を返します。

失敗した場合は、汎用の describe_parameter エラーのいずれかか、次のいずれかを返します。

- EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH – **describe_buffer** が sizeof(a_sql_data_type) でない場合に返る set エラー。
- EXTFNAPIV4_DESCRIBE_INVALID_STATE – クエリ処理の状態が注釈状態でない場合に返る set エラー。
- EXTFNAPI4_DESCRIBE_INVALID_ATTRIBUTE_VALUE – UDF がパラメータのデータ型を既に定義済み以外のものに設定しようとした場合に返る set エラー。

クエリ処理の状態
以下のときに有効です。

- 注釈状態

参照：
- EXTFNAPIV4_DESCRIBE_PARM_TYPE 属性 (get) (267 ページ)
- describe_parameter の一般的なエラー (354 ページ)
- クエリ処理の状態 (135 ページ)

EXTFNAPIV4_DESCRIBE_PARM_WIDTH 属性 (set)

EXTFNAPIV4_DESCRIBE_PARM_WIDTH 属性は、パラメータの幅を示します。`describe_parameter_set` のシナリオで使用します。

データ型
`a_sql_uint32`

説明
UDF に対するパラメータの幅。`EXTFNAPIV4_DESCRIBE_PARM_WIDTH` はスカラ・パラメータにのみ該当します。パラメータ幅は、対応するデータ型のパラメータを格納するために必要な記憶領域のサイズのバイト数です。

- **固定長のデータ型** – データの格納に必要なバイト数
- **可変長のデータ型** – 最大長
- **LOB データ型** – データへのハンドルの格納に必要な記憶領域のサイズ
- **TIME データ型** – コード化した時間の格納に必要な記憶領域のサイズ

使用法
これは読み込み専用プロパティです。幅は、対応するカラムのデータ型から取得されます。データ型を設定した後で幅を変更することはできません。

戻り値
成功した場合は、`sizeof(a_sql_uint32)` を返します。

失敗した場合は、汎用の `describe_parameter` エラーのいずれかか、次のいずれかを返します。

- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` – クエリの処理状態が注釈状態でない場合に返る set エラー。
- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` – **describe_buffer** が `a_sql_uint32` のサイズでない場合に返る set エラー。
- `EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER` – 指定のパラメータがテーブル・パラメータの場合に返る set エラー。 これには、パラメータ 0 やパラメータ *n* (*n* が入力テーブルの場合) が含まれる。
- `EXTFNAPI4_DESCRIBE_INVALID_ATTRIBUTE_VALUE` – UDF がパラメータの幅を再設定しようとした場合に返る set エラー。

クエリ処理の状態
以下のときに有効です。

- 注釈状態

参照：
- EXTFNAPIV4_DESCRIBE_PARM_WIDTH 属性 (get) (268 ページ)
- describe_parameter の一般的なエラー (354 ページ)
- クエリ処理の状態 (135 ページ)

EXTFNAPIV4_DESCRIBE_PARM_SCALE 属性 (set)
EXTFNAPIV4_DESCRIBE_PARM_SCALE 属性は、パラメータの位取りを示します。`describe_parameter_set` のシナリオで使用します。

データ型
`a_sql_uint32`

説明
UDF に対するパラメータの位取り。算術データ型については、パラメータの位取りは、数値の小数点の右側の桁数です。

この属性は以下に対しては無効です。

- 算術データ型以外のデータ型
- テーブル・パラメータ

使用法
これは読み込み専用プロパティです。位取りは、対応するカラムのデータ型から取得されます。データ型を設定した後で位取りを変更することはできません。

戻り値
成功した場合は、`sizeof(a_sql_uint32)` を返します。

失敗した場合は、汎用の `describe_parameter` エラーのいずれかか、次のいずれかを返します。

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` – **describe_buffer** が `a_sql_uint32` のサイズでない場合に返る set エラー。
- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` – 状態が注釈状態でない場合に返る set エラー。

- EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER – 指定のパラメータがテーブル・パラメータの場合に返る set エラー。 これには、パラメータ 0 やパラメータ *n* (*n* が入力テーブルの場合) が含まれます。

クエリ処理の状態
以下のときに有効です。

- 注釈状態

参照：

- EXTFNAPIV4_DESCRIBE_PARM_SCALE 属性 (get) (270 ページ)
- describe_parameter の一般的なエラー (354 ページ)
- クエリ処理の状態 (135 ページ)

EXTFNAPIV4_DESCRIBE_PARM_CAN_BE_NULL 属性 (set)

EXTFNAPIV4_DESCRIBE_PARM_CAN_BE_NULL 属性は、パラメータが NULL かどうかを返します。この属性を describe_parameter_set のシナリオで使用するとエラーが返ります。

データ型
a_sql_byte

説明
パラメータの値が実行時に NULL になり得る場合は true。テーブル・パラメータまたはパラメータ 0 については、値は false です。

使用法
これは読み込み専用プロパティです。

戻り値
これは読み込み専用プロパティです。設定しようとすると必ず EXTFNAPIV4_DESCRIBE_INVALID_ATTRIBUTE エラーが返ります。

クエリ処理の状態
適用されません。

EXTFNAPIV4_DESCRIBE_PARM_DISTINCT_VALUES 属性 (set)

EXTFNAPIV4_DESCRIBE_PARM_DISTINCT_VALUES 属性は、重複しない値の数を返します。この属性を describe_parameter_set のシナリオで使用するとエラーが返ります。

データ型
a_v4_extfn_estimate

説明
すべての呼び出し全体での重複しない値の推定数を返します。スカラ・パラメータに対してのみ有効です。

使用法
これは読み込み専用プロパティです。

戻り値
これは読み込み専用プロパティです。設定しようとすると必ず `EXTFNAPIV4_DESCRIBE_INVALID_ATTRIBUTE` エラーが返ります。

クエリ処理の状態
適用されません。

参照：
- EXTFNAPIV4_DESCRIBE_PARM_DISTINCT_VALUES 属性 (get) (273 ページ)
- EXTFNAPIV4_DESCRIBE_PARM_TYPE 属性 (get) (267 ページ)
- describe_parameter の一般的なエラー (354 ページ)
- クエリ処理の状態 (135 ページ)

EXTFNAPIV4_DESCRIBE_PARM_IS_CONSTANT 属性 (set)
`EXTFNAPIV4_DESCRIBE_PARM_DISTINCT_VALUES` 属性は、パラメータが定数かどうかを返します。この属性を `describe_parameter_set` のシナリオで使用するとエラーが返ります。

データ型
`a_sql_byte`

説明
文のパラメータが定数の場合は true。スカラ・パラメータに対してのみ有効です。

使用法
これは読み込み専用プロパティです。

戻り値
これは読み込み専用プロパティです。設定しようとすると必ず `EXTFNAPIV4_DESCRIBE_INVALID_ATTRIBUTE` エラーが返ります。

クエリ処理の状態
適用されません。

参照：
- EXTFNAPIV4_DESCRIBE_PARM_IS_CONSTANT 属性 (get) (274 ページ)
- EXTFNAPIV4_DESCRIBE_PARM_TYPE 属性 (set) (288 ページ)
- describe_parameter の一般的なエラー (354 ページ)
- クエリ処理の状態 (135 ページ)
- EXTFNAPIV4_DESCRIBE_PARM_CONSTANT_VALUE 属性 (get) (276 ページ)
- EXTFNAPIV4_DESCRIBE_PARM_TYPE 属性 (get) (267 ページ)

EXTFNAPIV4_DESCRIBE_PARM_CONSTANT_VALUE 属性 (set)
EXTFNAPIV4_DESCRIBE_PARM_CONSTANT_VALUE 属性は、パラメータの値を示します。`describe_parameter_set` のシナリオで使用します。

データ型
`an_extfn_value`

説明
パラメータの値 (describe の時点で把握している場合)。スカラ・パラメータに対してのみ有効です。

使用法
これは読み込み専用プロパティです。

戻り値
これは読み込み専用プロパティです。設定しようとすると必ず `EXTFNAPIV4_DESCRIBE_INVALID_ATTRIBUTE` エラーが返ります。

クエリ処理の状態
適用されません。

EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_COLUMNS 属性 (set)
`EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_COLUMNS` 属性は、テーブル内のカラム数を示します。`describe_parameter_set` のシナリオで使用します。

データ型
`a_sql_uint32`

説明
テーブル内のカラム数。引数 0 およびテーブル引数に対してのみ有効です。

使用法
UDF がこのプロパティを設定すると、サーバはその値を、**CREATE PROCEDURE** 文で指定されたパラメータ名と比較します。2 つの値が一致しない場合、サーバは

エラーを返します。これによって UDF は、**CREATE PROCEDURE** 文のパラメータ名が UDF が予期するものと同じであることを確認できます。

戻り値
成功した場合は、`sizeof(a_sql_uint32)` を返します。

失敗した場合は、汎用の `describe_parameter` エラーのいずれかか、次のいずれかを返します。

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` – **describe_buffer** が `sizeof a_sql_uint32` のサイズでない場合に返る set エラー。
- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` – 状態が注釈状態でない場合に返る set エラー。
- `EXTFNAPIV4_DESCRIBE_NON_TABLE_PARAMETER` – パラメータがテーブル・パラメータでない場合に返る set エラー。
- `EXTFNAPI4_DESCRIBE_INVALID_ATTRIBUTE_VALUE` – UDF が指定のテーブルのカラム数を再設定しようとした場合に返る set エラー。

クエリ処理の状態
以下のときに有効です。

- 注釈状態

参照：
- EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_COLUMNS 属性 (get) (277 ページ)
- クエリ処理の状態 (135 ページ)

EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_ROWS 属性 (set)
`EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_ROWS` 属性は、テーブル内のロー数を示します。`describe_parameter_set` のシナリオで使用します。

データ型
`a_sql_a_v4_extfn_estimate`

説明
テーブルの推定ロー数。引数 0 およびテーブル引数に対してのみ有効です。

使用法
UDF は、結果セットに含まれるロー数を推定した場合、引数 0 に対してこのプロパティを設定します。サーバは、最適化のときにこの推定を使用してクエリ処理の判断を下します。この値を入力テーブルに対して設定することはできません。

この値を設定しない場合、サーバは **DEFAULT_TABLE_UDF_ROW_COUNT** オプションで指定されたロー数をデフォルトとして使用します。

戻り値
成功した場合は、`a_v4_extfn_estimate` を返します。

失敗した場合は、汎用の `describe_parameter` エラーのいずれかか、次のいずれかを返します。

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` – **describe_buffer** が `a_v4_extfn_estimate` のサイズでない場合に返る set エラー。
- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` – 状態が最適化状態でない場合に返る set エラー。
- `EXTFNAPIV4_DESCRIBE_NON_TABLE_PARAMETER` – パラメータがテーブル・パラメータでない場合に返る get エラー。
- `EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER` – テーブル・パラメータが結果テーブルでない場合に返る get エラー。
- `EXTFNAPI4_DESCRIBE_INVALID_ATTRIBUTE_VALUE` – UDF が指定のテーブルのカラム数を再設定しようとした場合に返る get エラー。

クエリ処理の状態
以下のときに有効です。

- クエリ最適化状態

参照：
- EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_ROWS 属性 (get) (278 ページ)
- クエリ処理の状態 (135 ページ)

EXTFNAPIV4_DESCRIBE_PARM_TABLE_ORDERBY 属性 (set)
`EXTFNAPIV4_DESCRIBE_PARM_TABLE_ORDERBY` 属性は、テーブル内のローの順序を示します。`describe_parameter_set` のシナリオで使用します。

データ型
`a_v4_extfn_orderby_list`

説明
テーブル内のローの順序。このプロパティは引数 0 およびテーブル引数に対してのみ有効です。

使用法
この属性を使用することによって、UDF コードは以下が可能になります。

- 入力の **TABLE** パラメータが順序付けされているかどうかを判断する。
- 結果セットが順序付けされていることを宣言する。

パラメータ番号が 0 の場合、この属性は出力の結果セットを表します。パラメータが 0 より大きく、パラメータの種類がテーブルの場合、この属性は入力の **TABLE** パラメータを表します。

順序付けは `a_v4_extfn_orderby_list` で示されます。これは、カラムの順序数と、それに対応する昇順または降順のプロパティのリストを保持する構造体です。UDF が出力の結果セットに対してこの順序付けのプロパティを設定した場合、サーバは順序付けの最適化を実行できます。たとえば、UDF が結果セットの最初のカラムを昇順で生成した場合、サーバは同じカラムに対する重複した順序付けの要求を省略できます。

UDF が出力の結果セットに対して順序付けのプロパティを設定しない場合、サーバはデータが順序付けされていないものと見なします。

UDF が入力の **TABLE** パラメータに順序付けのプロパティを設定する場合、その入力データに対する順序付けはサーバが保証します。このシナリオでは、入力データの順序付けが必要であることを UDF がサーバに伝達します。サーバは、実行時に競合を検出した場合は SQL 例外を発生させます。たとえば、入力の **TABLE** パラメータの最初のカラムは昇順である必要があると UDF が伝達したのに対し、SQL 文に降順を指定する句が含まれている場合、サーバは SQL 例外を発生させます。

SQL に順序付けの句が含まれていない場合、サーバは順序付けを自動的に追加します。これにより、入力の **TABLE** パラメータが要件どおりに順序付けされます。

戻り値
成功した場合は、`a_v4_extfn_orderby_list` からコピーされたバイト数を返します。

クエリ処理の状態
以下のときに有効です。

- 注釈状態
- クエリ最適化状態

参照：
- EXTFNAPIV4_DESCRIBE_PARM_TABLE_ORDERBY 属性 (get) (279 ページ)
- クエリ処理の状態 (135 ページ)

EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY (set)
EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY 属性は、UDF がパーティション分割を必要としていることを示します。describe_parameter_set のシナリオで使用します。

データ型
a_v4_extfn_column_list

説明
UDF 開発者は、**EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY** を使用して、呼び出しの前にパーティション分割が必要であることを UDF のコードで宣言します。

使用法
UDF は、パーティションの適用またはパーティション分割の動的な実施のための確認を実行できます。UDF は **a_v4_extfn_column_list** を確保する必要があります。そのときは、入力テーブルに含まれるカラムの総数を考慮に入れ、そのデータをサーバに送信します。

戻り値
成功した場合は、a_v4_extfn_column_list のサイズを返します。この値は次と同じです。

```
sizeof(a_v4_extfn_column_list) + sizeof(a_sql_uint32) *
number_of_partition_columns
```

失敗した場合は、汎用の describe_parameter エラーのいずれかか、次のいずれかを返します。

- EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH – バッファの長さが予期したサイズ未満の場合に返る set エラー。

クエリ処理の状態
以下のときに有効です。

- 注釈状態
- クエリ最適化状態

例

```
void UDF_CALLBACK my_tpf_proc_describe( a_v4_extfn_proc_context
*ctx )
{
    if( ctx->current_state == EXTFNAPIV4_STATE_ANNOTATION ) {
        a_sql_int32    rc = 0;
        a_v4_extfn_column_list pbcol =
        { 1,        // 1 column in the partition by list
```

```
        2 };    // column index 2 requires partitioning

        // Describe partitioning for argument 1 (the table)
        rc = ctx->describe_parameter_set(
        ctx, 1,
        EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY,
        &pbcol,
        sizeof(pbcol) );

        if( rc == 0 ) {
        ctx->set_error( ctx, 17000,
           “Runtime error, unable set partitioning requirements for
column.” );
        }
    }
}
```

参照：

- EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY (get) (280 ページ)
- describe_parameter の一般的なエラー (354 ページ)
- V4 API の describe_parameter と EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY (157 ページ)
- EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY を使用した並列処理 TPF の PARTITION BY の例 (160 ページ)
- クエリ処理の状態 (135 ページ)
- 入力データのパーティション分割 (157 ページ)

EXTFNAPIV4_DESCRIBE_PARM_TABLE_REQUEST_REWIND 属性 (Set)

`EXTFNAPIV4_DESCRIBE_PARM_TABLE_REQUEST_REWIND` 属性は、コンシューマが入力テーブルのリワインドを要求していることを示します。`describe_parameter_set` のシナリオで使用します。

データ型
`a_sql_byte`

説明
コンシューマが入力テーブルのリワインドを希望していることを示します。テーブル入力引数に対してのみ有効です。デフォルトでは、このプロパティは false です。

使用法
UDF は、入力テーブルのリワインド機能を必要とする場合には、最適化状態のときにこのプロパティを設定する必要があります。

戻り値
成功した場合は、`sizeof(a_sql_byte)` を返します。

失敗した場合は、汎用の `describe_parameter` エラーのいずれかか、次のいずれかを返します。

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` – **describe_buffer** が `a_sql_byte` のサイズでない場合に返る set エラー。
- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` – 状態が最適化状態でない場合に返る set エラー。
- `EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER` – UDF がこの属性をパラメータ 0 に対して設定しようとした場合に返る set エラー。
- `EXTFNAPIV4_DESCRIBE_NON_TABLE_PARAMETER` – UDF がこの属性をテーブルでないパラメータに対して設定しようとした場合に返る set エラー。
- `EXTFNAPIV4_DESCRIBE_INVALID_ATTRIBUTE_VALUE` – UDF がこの属性を 0 と 1 以外の値に設定しようとした場合に返る set エラー。

クエリ処理の状態
以下のときに有効です。

- 最適化状態

例
この例では、最適化状態のときに関数 **my_udf_describe** が呼び出されると、`describe_parameter_set` の呼び出しによって、リワインドが必要な可能性があることがテーブル入力パラメータ 1 のプロデューサに通知されます。

プロシージャ定義の例です。

```
CREATE PROCEDURE my_udf(IN t TABLE(c1 INT))
RESULT (x INT)
EXTERNAL NAME ‘my_udf@myudflib’;
```

`_describe_extfn` API 関数のコードの例です。

```
my_udf_describe(a_v4_extfn_proc_context *cntxt)
{
if( cntxt->current_state == EXTFNAPIV4_STATE_OPTIMIZATION ) {
a_sql_byte rewind_required = 1;
a_sql_int32 ret = 0;

ret = cntxt->describe_parameter_set( cntxt, 1,
EXTFNAPIV4_DESCRIBE_PARM_TABLE_REQUEST_REWIND,
&rewind_required,
sizeof(a_sql_byte) );

if( ret <= 0 ) {
    // Handle the error.
```

```
}
}
}
```

参照：

- EXTFNAPIV4_DESCRIBE_PARM_TABLE_REQUEST_REWIND 属性 (get) (282 ページ)
- EXTFNAPIV4_DESCRIBE_PARM_TABLE_HAS_REWIND 属性 (set) (300 ページ)
- EXTFNAPIV4_DESCRIBE_PARM_TABLE_HAS_REWIND 属性 (get) (283 ページ)
- _rewind_extfn (350 ページ)
- クエリ処理の状態 (135 ページ)

EXTFNAPIV4_DESCRIBE_PARM_TABLE_HAS_REWIND 属性 (set)

`EXTFNAPIV4_DESCRIBE_PARM_TABLE_HAS_REWIND` 属性は、パラメータがリワインドをサポートしていることを示します。`describe_parameter_set` のシナリオで使用します。

データ型
`a_sql_byte`

説明
プロデューサがリワインドをサポートできるかどうかを示します。テーブル引数に対してのみ有効です。

`DESCRIBE_PARM_TABLE_HAS_REWIND` を true に設定する予定の場合は、リワインド・テーブル・コールバック (**_rewind_extfn()**) の実装も提供する必要があります。コールバック・メソッドを提供しないと、サーバが UDF を実行できません。

使用法
UDF は、結果テーブルに対するリワインド機能をコストなしで提供できる場合には、最適化状態のときにこのプロパティを設定します。UDF がリワインドを提供するとコストがかかる場合には、このプロパティを設定しないか、または 0 に設定します。0 に設定した場合は、サーバがリワインドのサポートを提供します。

戻り値
成功した場合は、`sizeof(a_sql_byte)` を返します。

失敗した場合は、汎用の `describe_parameter` エラーのいずれかか、次のいずれかを返します。

- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` – **describe_buffer** が `a_sql_byte` のサイズでない場合に返る set エラー。

- EXTFNAPIV4_DESCRIBE_INVALID_STATE – 状態が最適化状態でない場合に返る set エラー。
- EXTFNAPIV4_DESCRIBE_NON_TABLE_PARAMETER – UDF がこの属性をテーブルでないパラメータに対して設定しようとした場合に返る set エラー。
- EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER – 指定の引数が結果テーブルでない場合に返る set エラー。
- EXTFNAPIV4_DESCRIBE_INVALID_ATTRIBUTE_VALUE – UDF がこの属性を 0 と 1 以外の値に設定しようとした場合に返る set エラー。

クエリ処理の状態
以下のときに有効です。

- 最適化状態

参照：

- EXTFNAPIV4_DESCRIBE_PARM_TABLE_REQUEST_REWIND 属性 (get) (282 ページ)
- EXTFNAPIV4_DESCRIBE_PARM_TABLE_REQUEST_REWIND 属性 (Set) (298 ページ)
- EXTFNAPIV4_DESCRIBE_PARM_TABLE_HAS_REWIND 属性 (get) (283 ページ)
- _rewind_extfn (350 ページ)
- クエリ処理の状態 (135 ページ)

EXTFNAPIV4_DESCRIBE_PARM_TABLE_UNUSED_COLUMNS 属性 (set)
EXTFNAPIV4_DESCRIBE_PARM_TABLE_UNUSED_COLUMNS 属性は、利用しないカラムをリストします。describe_parameter_set のシナリオで使用します。

データ型
a_v4_extfn_column_list

説明
サーバまたは UDF が利用しない出力テーブル・カラムのリスト。

出力テーブル・パラメータについては、UDF は通常すべてのカラムに対してデータを生成し、サーバはすべてのカラムを利用します。入力テーブル・パラメータについても同様で、サーバは通常すべてのカラムに対してデータを生成し、UDF はすべてのカラムを利用します。

しかし場合によっては、サーバまたは UDF が一部のカラムを利用しないこともあります。そのような場合の最善の方法は、UDF が describe 属性 **EXTFNAPIV4_DESCRIBE_PARM_TABLE_UNUSED_COLUMNS** を使用して、出力テーブルに対する **GET** を実行することです。この処理は、出力テーブルのうちでサー

バが利用しないカラムのリストをサーバに問い合わせる処理です。UDF はこのリストを、出力テーブルのカラム・データを格納するときに利用できます。つまり、使用しないカラムに対するデータの格納を省略できます。

要約すると、出力テーブルについては、使用しないカラムのリストを UDF が問い合わせます。入力テーブルについては、使用しないカラムのリストを UDF がプッシュします。

使用法
UDF は、入力テーブル・パラメータの一部のカラムを使用しない場合には、最適化状態のときにこのプロパティを設定します。UDF は出力テーブルのすべてのカラムを含めた `a_v4_extfn_column_list` を確保してサーバに渡す必要があります。これを受けてサーバは、使用しないすべてのカラムの位置を 1 とマークします。サーバは、内部のデータ構造体にリストをコピーします。

戻り値
成功した場合は、カラム・リストのサイズを返します。
`sizeof(a_v4_extfn_column_list) + sizeof(a_sql_uint32) * number result columns`.

失敗した場合は、汎用の `describe_parameter` エラーのいずれかか、次のいずれかを返します。

- `EXTFNAPIV4_DESCRIBE_INVALID_STATE` – 状態が最適化状態でない場合に返る set エラー。
- `EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER` – UDF がこの属性を入力テーブルに対して取得しようとした場合に返る set エラー。
- `EXTFNAPIV4_DESCRIBE_NON_TABLE_PARAMETER` – UDF がこの属性をテーブルでないパラメータに対して設定しようとした場合に返る set エラー。

クエリ処理の状態
以下のときに有効です。

- 最適化状態

参照：

- EXTFNAPIV4_DESCRIBE_PARM_TABLE_UNUSED_COLUMNS 属性 (get) (284 ページ)

*describe_udf_get

v4 API メソッド describe_udf_get では、UDF のプロパティをサーバから取得します。

宣言

```
a_sql_int32 (SQL_CALLBACK *describe_udf_get)(
    a_v4_extfn_proc_context    *cntxt,
    a_v4_extfn_describe_udf_type    describe_type,
    void    *describe_buffer,
    size_t    describe_buffer_len );
```

パラメータ

パラメータ	説明
cntxt	この UDF のプロシージャ・コンテキスト・オブジェクト。
describe_type	取得するプロパティを示すセレクタ。
describe_buffer	サーバで設定する、指定のプロパティの describe 情報を保持する構造体。具体的な構造体またはデータ型は **describe_type** パラメータで示されます。
describe_buffer_length	**describe_buffer** のバイト単位の長さ。

戻り値

成功した場合は、0 または **describe_buffer** に書き込まれたバイト数を返します。値 0 の場合、サーバは属性を取得できなかったが、エラー状態は発生しなかったことを示します。エラーが発生した場合や、プロパティが取得されなかった場合は、汎用の describe_udf エラーのいずれかを返します。

参照：

- *describe_udf_set (305 ページ)
- describe_udf の一般的なエラー (354 ページ)

*describe_udf_get の属性

describe_udf_get の属性を示すコードです。

```
typedef enum a_v4_extfn_describe_udf_type {
    EXTFNAPIV4_DESCRIBE_UDF_NUM_PARMS,
    EXTFNAPIV4_DESCRIBE_UDF_LAST
} a_v4_extGetfn_describe_udf_type;
```

EXTFNAPIV4_DESCRIBE_UDF_NUM_PARMS 属性 (get)
EXTFNAPIV4_DESCRIBE_UDF_NUM_PARMS 属性は、パラメータの数を示します。describe_udf_get のシナリオで使用します。

データ型
a_sql_uint32

説明
UDF に渡されるパラメータの数。

使用法
CREATE PROCEDURE 文で定義されているパラメータの数を取得します。

戻り値
成功した場合は、sizeof(a_sql_uint32) を返します。

失敗した場合は、汎用の describe_udf エラーのいずれかか、次のいずれかを返します。

- EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH – describe バッファが a_sql_uint32 のサイズでない場合に返る get エラー。
- EXTFNAPIV4_DESCRIBE_INVALID_STATE – 状態が初期状態より後でない場合に返る get エラー。

クエリ処理の状態

- 注釈状態
- クエリ最適化状態
- プラン構築状態
- 実行状態

参照：
- EXTFNAPIV4_DESCRIBE_UDF_NUM_PARMS 属性 (set) (306 ページ)
- describe_udf の一般的なエラー (354 ページ)
- クエリ処理の状態 (135 ページ)

*describe_udf_set

v4 API メソッド `describe_udf_set` では、UDF のプロパティをサーバに対して設定します。

宣言

```
a_sql_int32 (SQL_CALLBACK *describe_udf_set)(
    a_v4_extfn_proc_context    *cntxt,
    a_v4_extfn_describe_udf_type    describe_type,
    const void    *describe_buffer,
    size_t    describe_buffer_len );
```

パラメータ

パラメータ	説明
cntxt	この UDF のプロシージャ・コンテキスト・オブジェクト。
describe_type	設定するプロパティを示すセレクタ。
describe_buffer	サーバで設定する、指定のプロパティの describe 情報を保持する構造体。具体的な構造体またはデータ型は **describe_type** パラメータで示されます。
describe_buffer_length	**describe_buffer** のバイト単位の長さ。

戻り値

成功した場合は、**describe_buffer** に書き込まれたバイト数を返します。エラーが発生した場合や、プロパティが取得されなかった場合は、汎用の `describe_udf` エラーのいずれかを返します。

エラーが発生した場合や、プロパティが取得されなかった場合は、汎用の `describe_udf` エラーのいずれかか、次のいずれかを返します。

- `EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER` – **cntxt** 引数か **describe_buffer** 引数が NULL の場合、または **describe_buffer_length** が 0 の場合に返る set エラー。
- `EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH` – 要求された属性のサイズと指定された **describe_buffer_length** に不一致がある場合に返る set エラー。

参照：

- *describe_udf_get (303 ページ)
- describe_udf の一般的なエラー (354 ページ)

***describe_udf_set の属性**

describe_udf_set の属性を示すコードです。

```
typedef enum a_v4_extfn_describe_udf_type {
    EXTFNAPIV4_DESCRIBE_UDF_NUM_PARMS,
    EXTFNAPIV4_DESCRIBE_UDF_LAST
} a_v4_extGetfn_describe_udf_type;
```

EXTFNAPIV4_DESCRIBE_UDF_NUM_PARMS 属性 (set)

EXTFNAPIV4_DESCRIBE_UDF_NUM_PARMS 属性は、パラメータの数を示します。describe_udf_set のシナリオで使用します。

データ型
a_sql_uint32

説明
UDF に渡されるパラメータの数。

使用法
UDF がこのプロパティを設定すると、サーバはその値を、**CREATE PROCEDURE** 文で指定されたパラメータの数と比較します。2 つの値が一致しない場合、サーバは SQL エラーを返します。これによって UDF は、**CREATE PROCEDURE** 文のパラメータ数が UDF が予期するものと同じであることを確認できます。

戻り値
成功した場合は、sizeof(a_sql_uint32) を返します。

失敗した場合は、汎用の describe_udf エラーのいずれかか、次のいずれかを返します。

- EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH – describe バッファが a_sql_uint32 のサイズでない場合に返る set エラー。
- EXTFNAPIV4_DESCRIBE_INVALID_STATE – 状態が注釈状態でない場合に返る set エラー。
- EXTFNAPIV4_DESCRIBE_INVALID_ATTRIBUTE_VALUE – UDF がパラメータのデータ型を再設定しようとした場合に返る set エラー。

クエリ処理の状態

- 注釈状態

参照：

- EXTFNAPIV4_DESCRIBE_UDF_NUM_PARMS 属性 (get) (304 ページ)

- describe_udf の一般的なエラー (354 ページ)
- クエリ処理の状態 (135 ページ)

カラム・タイプの記述 (a_v4_extfn_describe_col_type)

a_v4_extfn_describe_col_type 列挙型では、UDF が取得または設定したカラム・プロパティを選択します。

実装

```
typedef enum a_v4_extfn_describe_col_type {
    EXTFNAPIV4_DESCRIBE_COL_NAME,
    EXTFNAPIV4_DESCRIBE_COL_TYPE,
    EXTFNAPIV4_DESCRIBE_COL_WIDTH,
    EXTFNAPIV4_DESCRIBE_COL_SCALE,
    EXTFNAPIV4_DESCRIBE_COL_CAN_BE_NULL,
    EXTFNAPIV4_DESCRIBE_COL_DISTINCT_VALUES,
    EXTFNAPIV4_DESCRIBE_COL_IS_UNIQUE,
    EXTFNAPIV4_DESCRIBE_COL_IS_CONSTANT,
    EXTFNAPIV4_DESCRIBE_COL_CONSTANT_VALUE,
    EXTFNAPIV4_DESCRIBE_COL_IS_USED_BY_CONSUMER,
    EXTFNAPIV4_DESCRIBE_COL_MINIMUM_VALUE,
    EXTFNAPIV4_DESCRIBE_COL_MAXIMUM_VALUE,
    EXTFNAPIV4_DESCRIBE_COL_VALUES_SUBSET_OF_INPUT,
    EXTFNAPIV4_DESCRIBE_COL_LAST
} a_v4_extfn_describe_col_type;
```

メンバの概要

メンバ	説明
EXTFNAPIV4_DESCRIBE_COL_NAME	カラム名 (有効な識別子)。
EXTFNAPIV4_DESCRIBE_COL_TYPE	カラムのデータ型。
EXTFNAPIV4_DESCRIBE_COL_WIDTH	文字列の幅 (NUMERIC の精度)。
EXTFNAPIV4_DESCRIBE_COL_SCALE	NUMERIC の位取り。
EXTFNAPIV4_DESCRIBE_COL_CAN_BE_NULL	カラムが NULL になり得る場合は true。
EXTFNAPIV4_DESCRIBE_COL_DISTINCT_VALUES	カラム内の重複しない値の推定数。

メンバ	説明
EXTFNAPIV4_DESCRIBE_COL_IS_UNIQUE	カラムがテーブル内でユニークな場合には true。
EXTFNAPIV4_DESCRIBE_COL_IS_CONSTANT	カラムが文の有効期間全体にわたって定数の場合には true。
EXTFNAPIV4_DESCRIBE_COL_CONSTANT_VALUE	パラメータの値 (describe の時点で把握している場合)。
EXTFNAPIV4_DESCRIBE_COL_IS_USED_BY_CONSUMER	テーブルのコンシューマがカラムを必要とする場合には true。
EXTFNAPIV4_DESCRIBE_COL_MINIMUM_VALUE	カラムの最小値 (把握している場合)。
EXTFNAPIV4_DESCRIBE_COL_MAXIMUM_VALUE	カラムの最大値 (把握している場合)。
EXTFNAPIV4_DESCRIBE_COL_VALUES_SUBSET_OF_INPUT	結果カラムの値は入力テーブルのカラムのサブセット。
EXTFNAPIV4_DESCRIBE_COL_LAST	v4 API に対する最初の不正な値。範囲外の値。

パラメータ・タイプの記述 (a_v4_extfn_describe_parm_type)

`a_v4_extfn_describe_parm_type` 列挙型では、UDF が取得または設定したパラメータ・プロパティを選択します。

実装

```
typedef enum a_v4_extfn_describe_parm_type {
    EXTFNAPIV4_DESCRIBE_PARM_NAME,
    EXTFNAPIV4_DESCRIBE_PARM_TYPE,
    EXTFNAPIV4_DESCRIBE_PARM_WIDTH,
    EXTFNAPIV4_DESCRIBE_PARM_SCALE,
    EXTFNAPIV4_DESCRIBE_PARM_CAN_BE_NULL,
    EXTFNAPIV4_DESCRIBE_PARM_DISTINCT_VALUES,
    EXTFNAPIV4_DESCRIBE_PARM_IS_CONSTANT,
    EXTFNAPIV4_DESCRIBE_PARM_CONSTANT_VALUE,

    EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_COLUMNS,
    EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_ROWS,
    EXTFNAPIV4_DESCRIBE_PARM_TABLE_ORDERBY,
    EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY,
```

```
    EXTFNAPIV4_DESCRIBE_PARM_TABLE_REQUEST_REWIND,
    EXTFNAPIV4_DESCRIBE_PARM_TABLE_HAS_REWIND,
    EXTFNAPIV4_DESCRIBE_PARM_TABLE_UNUSED_COLUMNS,

    EXTFNAPIV4_DESCRIBE_PARM_LAST
} a_v4_extfn_describe_parm_type;
```

メンバの概要

メンバ	説明
EXTFNAPIV4_DESCRIBE_PARM_NAME	パラメータ名 (有効な識別子)。
EXTFNAPIV4_DESCRIBE_PARM_TYPE	データ型。
EXTFNAPIV4_DESCRIBE_PARM_WIDTH	文字列の幅 (NUMERIC の精度)。
EXTFNAPIV4_DESCRIBE_PARM_SCALE	NUMERIC の位取り。
EXTFNAPIV4_DESCRIBE_PARM_CAN_BE_NULL	値が NULL になり得る場合は true。
EXTFNAPIV4_DESCRIBE_PARM_DISTINCT_VALUES	すべての呼び出し全体での重複しない値の推定数。
EXTFNAPIV4_DESCRIBE_PARM_IS_CONSTANT	文のパラメータが定数の場合は true。
EXTFNAPIV4_DESCRIBE_PARM_CONSTANT_VALUE	パラメータの値 (describe の時点で把握している場合)。
以下のセレクタでは、テーブル・パラメータのプロパティを取得または設定できます。これらの列挙子の値はスカラ・パラメータでは使用できません。	
EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_COLUMNS	テーブル内のカラム数。
EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_ROWS	テーブルの推定ロー数。
EXTFNAPIV4_DESCRIBE_PARM_TABLE_ORDERBY	テーブル内のローの順序。
EXTFNAPIV4_DESCRIBE_PARM_TABLE_PARTITIONBY	パーティション分割。ANY には *number_of_columns*=0 を使用。

メンバ	説明
EXTFNAPIV4_DESCRIBE_PARM_TABLE_REQUEST_REWIND	コンシューマが入力テーブルのリワインド機能を希望する場合は true。
EXTFNAPIV4_DESCRIBE_PARM_TABLE_HAS_REWIND	プロデューサがリワインドをサポートする場合は true を返します。
EXTFNAPIV4_DESCRIBE_PARM_TABLE_UNUSED_COLUMNS	サーバまたは UDF が利用しない出力テーブル・カラムのリスト。
EXTFNAPIV4_DESCRIBE_PARM_LAST	v4 API に対する最初の不正な値。範囲外の値。

戻り値の記述 (a_v4_extfn_describe_return)

`a_v4_extfn_describe_return` 列挙型は、`a_v4_extfn_proc_context.describe_xxx_get()` または `a_v4_extfn_proc_context.describe_xxx_set()` が成功しなかった場合の戻り値を表します。

実装

```
typedef enum a_v4_extfn_describe_return {
    EXTFNAPIV4_DESCRIBE_NOT_AVAILABLE          =  0,   // the specified operation has no
meaning either for this attribute or in

the current context.
    EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH        = -1,   // the provided buffer size
does not match the required length or the

length is insufficient.
    EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER       = -2,   // the provided parameter number
is invalid
    EXTFNAPIV4_DESCRIBE_INVALID_COLUMN           = -3,   // the column number is invalid
for this table parameter
    EXTFNAPIV4_DESCRIBE_INVALID_STATE          = -4,   // the describe method call is not
valid in the present state
    EXTFNAPIV4_DESCRIBE_INVALID_ATTRIBUTE       = -5,   // the attribute is known but not
appropriate for this object
    EXTFNAPIV4_DESCRIBE_UNKNOWN_ATTRIBUTE        = -6,   // the identified attribute is
not known to this server version
    EXTFNAPIV4_DESCRIBE_NON_TABLE_PARAMETER       = -7,   // the specified parameter is
not a table parameter (for describe_col_get()

or set())
    EXTFNAPIV4_DESCRIBE_INVALID_ATTRIBUTE_VALUE       = -8,   // the specified attribute
value is illegal
     EXTFNAPIV4_DESCRIBE_LAST                = -9
} a_v4_extfn_describe_return;
```

メンバの概要

メンバ	戻り値	説明
EXTFNAPIV4_DESCRIBE_NOT_AVAILABLE	0	指定の操作はこの属性に対して意味を持たないか、または現在のコンテキストで意味を持ちません。
EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH	-1	指定のバッファ・サイズが必要な長さと一致しないか、または長さが不十分です。
EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER	-2	指定のパラメータ番号が無効です。
EXTFNAPIV4_DESCRIBE_INVALID_COLUMN	-3	カラム番号がこのテーブル・パラメータに対して無効です。
EXTFNAPIV4_DESCRIBE_INVALID_STATE	-4	現在の状態では describe メソッド呼び出しは無効です。
EXTFNAPIV4_DESCRIBE_INVALID_ATTRIBUTE	-5	既知の属性ですが、このオブジェクトには該当しません。
EXTFNAPIV4_DESCRIBE_UNKNOWN_ATTRIBUTE	-6	指定の属性は、このサーバ・バージョンでは未知の属性です。
EXTFNAPIV4_DESCRIBE_NON_TABLE_PARAMETER	-7	指定のパラメータはテーブル・パラメータではありません (対象は `describe_col_get()` または `describe_col_set()`)。
EXTFNAPIV4_DESCRIBE_INVALID_ATTRIBUTE_VALUE	-8	指定の属性値は無効です。
EXTFNAPIV4_DESCRIBE_LAST	-9	v4 API に対する最初の不正な値。

説明

`a_v4_extfn_proc_context.describe_xxx_get()` と `a_v4_extfn_proc_context.describe_xxx_set()` の戻り値は符号付き整数です。 結果が正の値の場合は、操作は成功し、値はコピーされたバイト数を表します。 戻り値が 0 以下の場合は、操作は失敗し、戻り値は `a_v4_extfn_describe_return` のいずれかの値です。

UDF タイプの記述 (a_v4_extfn_describe_udf_type)

a_v4_extfn_describe_udf_type 列挙型では、UDF が取得または設定する論理プロパティを選択します。

実装

```
typedef enum a_v4_extfn_describe_udf_type {
    EXTFNAPIV4_DESCRIBE_UDF_NUM_PARMS,
    EXTFNAPIV4_DESCRIBE_UDF_LAST
} a_v4_extfn_describe_udf_type;
```

メンバの概要

メンバ	説明
EXTFNAPIV4_DESCRIBE_UDF_NUM_PARMS	UDF に渡されるパラメータの数。
EXTFNAPIV4_DESCRIBE_UDF_LAST	範囲外の値。

説明

UDF は、プロパティを取得するときに a_v4_extfn_proc_context.describe_udf_get() メソッドを使用し、UDF 全体についてのプロパティを設定するときに a_v4_extfn_proc_context.describe_udf_set() メソッドを使用します。a_v4_extfn_describe_udf_type 列挙子では、UDF が取得または設定する論理プロパティを選択します。

参照：

- 外部プロシージャ・コンテキスト (a_v4_extfn_proc_context) (317 ページ)

実行状態 (a_v4_extfn_state)

a_v4_extfn_state 列挙型は、UDF のクエリの処理状態を表します。

実装

```
typedef enum a_v4_extfn_state {
   EXTFNAPIV4_STATE_INITIAL,                // Server initial state,
not used by UDF
    EXTFNAPIV4_STATE_ANNOTATION,               // Annotating parse
tree with UDF reference
    EXTFNAPIV4_STATE_OPTIMIZATION,              // Optimizing
   EXTFNAPIV4_STATE_PLAN_BUILDING,            // Building execution
plan
```

```
    EXTFNAPIV4_STATE_EXECUTING,                  // Executing UDF and
fetching results from UDF
    EXTFNAPIV4_STATE_LAST
} a_v4_extfn_state;
```

メンバの概要

メンバ	説明
EXTFNAPIV4_STATE_INITIAL	サーバの初期状態。この状態のときに呼び出される UDF メソッドは `_start_extfn` のみです。
EXTFNAPIV4_STATE_ANNOTATION	UDF 参照を使用して解析ツリーに注釈を付けています。この状態のときには UDF は呼び出されません。
EXTFNAPIV4_STATE_OPTIMIZATION	最適化を実行しています。サーバは UDF の `_start_extfn` メソッドを呼び出し、続いて `_describe_extfn` 関数を呼び出します。
EXTFNAPIV4_STATE_PLAN_BUILDING	クエリ実行プランを構築しています。サーバは UDF の `_describe_extfn` 関数を呼び出します。
EXTFNAPIV4_STATE_EXECUTING	UDF を実行し、UDF から結果をフェッチしています。サーバは UDF からデータのフェッチを開始する前に `_describe_extfn` 関数を呼び出します。続いてサーバは `_evaluate_extfn` を呼び出してフェッチ・サイクルを開始します。フェッチ・サイクルの間に、サーバは `a_v4_extfn_table_func` で定義されている関数を呼び出します。フェッチが完了すると、サーバは UDF の `_close_extfn` 関数を呼び出します。
EXTFNAPIV4_STATE_LAST	v4 API に対する最初の不正な値。範囲外の値。

説明

`a_v4_extfn_state` 列挙型は、サーバの UDF 実行がどの段階にいるかを示します。サーバが状態間を遷移するときには、前の状態が終了することを UDF に通知するために、UDF の `_leave_state_extfn` 関数を呼び出します。また、新しい状態に入ることを UDF に通知するために、UDF の `_enter_state_extfn` 関数を呼び出します。

UDF のクエリ処理の状態に応じて、UDF が実行できる操作は制限されます。たとえば、注釈状態のときには、UDF は定数パラメータに対してのみデータ型を取得できます。

参照：
- クエリ処理の状態 (135 ページ)
- _start_extfn (314 ページ)
- _evaluate_extfn (315 ページ)
- _enter_state_extfn (316 ページ)
- _leave_state_extfn (317 ページ)
- テーブル関数 (a_v4_extfn_table_func) (346 ページ)

外部関数 (a_v4_extfn_proc)

サーバは a_v4_extfn_proc 構造体を使用して UDF のさまざまなエントリ・ポイントを呼び出します。サーバは各関数に a_v4_extfn_proc_context のインスタンスを渡します。

メソッドの概要

メソッド	説明
_start_extfn	構造体を確保し、そのアドレスを a_v4_extfn_proc_context の **_user_data** フィールドに格納します。
_finish_extfn	a_v4_extfn_proc_context の **_user_data** フィールドに格納されているアドレスの構造体を解放します。
_evaluate_extfn	必須の関数ポインタ。新しい引数値のセットによる関数呼び出しごとに呼び出されます。
_describe_extfn	describe の API (227 ページ) を参照してください。
_enter_state_extfn	UDF はこの関数を使用して構造体を確保できます。
_leave_state_extfn	UDF はこの関数を使用して、対象の状態で必要だったメモリまたはリソースを解放できます。

_start_extfn

v4 API メソッド _start_extfn は、初期化関数に対するオプションのポインタとして使用します。この関数の唯一の引数は a_v4_extfn_proc_context 構造体のポインタです。

宣言

```
_start_extfn(
a_v4_extfn_proc_context *
```

```
)
```

使用法

`_start_extfn` メソッドでは、構造体を確保し、そのアドレスを `a_v4_extfn_proc_context` の `_user_data` フィールドに格納します。初期化が不要な場合は、この関数ポインタを NULL ポインタに設定する必要があります。

パラメータ

パラメータ	説明
cntxt	プロシージャ・コンテキスト・オブジェクト。

_finish_extfn

v4 API メソッド `_finish_extfn` は、シャットダウン関数に対するオプションのポインタとして使用します。この関数の唯一の引数は `a_v4_extfn_proc_context` のポインタです。

宣言

```
_finish_extfn(
     a_v4_extfn_proc_context *cntxt,
)
```

使用法

`_finish_extfn` API では、`a_v4_extfn_proc_context` の `_user_data` フィールドに格納されているアドレスの構造体を解放します。クリーンアップが不要な場合は、この関数ポインタを NULL ポインタに設定する必要があります。

パラメータ

パラメータ	説明
cntxt	プロシージャ・コンテキスト・オブジェクト。

_evaluate_extfn

v4 API メソッド `_evaluate_extfn` は、必須の関数ポインタとして使用します。新しい引数値のセットによる関数呼び出しごとに呼び出されます。

宣言

```
_evaluate_extfn(
     a_v4_extfn_proc_context *cntxt,
     void *args_handle
)
```

使用法
_evaluate_extfn 関数では、a_v4_extfn_table 構造体の a_v4_extfn_table_func 部分を設定することによって結果のフェッチ方法をサーバに伝達する必要があり、コンテキストの set_value メソッドを引数 0 で呼び出してこの情報をサーバに送信します。またこの関数では、引数 0 の set_value を呼び出す前に、Ia_v4_extfn_table 構造体の a_v4_extfn_value_schema を設定することによって、出力スキーマについてサーバに通知する必要があります。入力引数値へのアクセスには get_value コールバック関数を使用します。この時点で、UDF は定数と非定数のどちらの引数も使用できます。

パラメータ

パラメータ	説明
cntxt	プロシージャ・コンテキスト・オブジェクト。
args_handle	サーバの引数へのハンドル。

_describe_extfn

_describe_extfn は、サーバが論理プロパティを取得および設定できるようにするために、それぞれの状態の最初の時点で呼び出されます。UDF はそのために、a_v4_proc_context object で 6 つの describe メソッド (describe_parameter_get、describe_parameter_set、describe_column_get、describe_column_set、describe_udf_get、describe_udf_set) を使用します。

詳細については、describe の API (227 ページ) を参照してください。

_enter_state_extfn

v4 API メソッド _enter_state_extfn は、UDF が新しい状態に入るときに通知するオプションのエントリ・ポイントとして UDF が実装できます。

宣言

```
_enter_state_extfn(
     a_v4_extfn_proc_context *cntxt,
)
```

使用法
UDF はこの通知を使用して構造体を確保できます。

パラメータ

パラメータ	説明
cntxt	プロシージャ・コンテキスト・オブジェクト。

_leave_state_extfn

v4 API メソッド `_leave_state_extfn` は、UDF のクエリ処理の状態を抜けるときに通知を受信するために UDF が実装できるオプションのエントリ・ポイントです。

宣言

```
_leave_state_extfn(
     a_v4_extfn_proc_context *cntxt,
)
```

使用法

UDF はこの通知を使用して、対象の状態で必要だったメモリまたはリソースを解放できます。

パラメータ

パラメータ	説明
cntxt	プロシージャ・コンテキスト・オブジェクト。

外部プロシージャ・コンテキスト (a_v4_extfn_proc_context)

`a_v4_extfn_proc_context` 構造体では、サーバおよび UDF からのコンテキスト情報を保持します。

実装

```
typedef struct a_v4_extfn_proc_context {
.
.
.
} a_v4_extfn_proc_context;
```

メソッドの概要

戻り値の型	メソッド	説明
short	**get_value**	UDF から入力引数を取得します。

戻り値の型	メソッド	説明
short	**get_value_is_constant**	指定された引数が定数かどうかを示す情報を UDF が要求できます。
short	**set_value**	UDF が `_evaluate_extfn` 関数または `_describe_extfn` 関数で使用し、UDF の出力の形式や、UDF からの結果をフェッチする方法についてサーバに伝達します。
a_sql_uint32	**get_is_cancelled**	**get_is_cancelled** コールバックを 1 ～ 2 秒ごとに呼び出して、現在の文をユーザが中断したかどうかを確認します。
short	**set_error**	現在の文をロールバックしてエラーを生成します。
void	**log_message**	メッセージ・ログにメッセージを書き込みます。
short	**convert_value**	データ型を別のものに変換します。
short	**get_option**	設定可能なオプションの値を取得します。
void	**alloc**	長さが "len" 以上のメモリ・ブロックを確保します。
void	**free**	**alloc()** により指定の存続期間で確保したメモリを解放します。
a_sql_uint32	**describe_column_get**	詳細については、*describe_column_get (228 ページ) を参照してください。
a_sql_uint32	**describe_column_set**	詳細については、*describe_column_set (246 ページ) を参照してください。
a_sql_uint32	**describe_parameter_get**	詳細については、*describe_parameter_get (264 ページ) を参照してください。
a_sql_uint32	**describe_parameter_set**	詳細については、*describe_parameter_set (286 ページ) を参照してください。
a_sql_uint32	**describe_udf_get**	詳細については、*describe_udf_get (303 ページ) を参照してください。
a_sql_uint32	**describe_udf_set**	詳細については、*describe_udf_set (305 ページ) を参照してください。
short	**open_result_set**	テーブル値の結果セットを開きます。
short	**close_result_set**	開いた結果セットを閉じます。

戻り値の型	メソッド	説明
short	**get_blob**	BLOB の入力パラメータを取得します。
short	**set_cannot_be_distributed**	ライブラリが分散可能な場合でも、UDF レベルで分散を無効にします。

データ・メンバとデータ型の概要

データ・メンバ	データ型	説明
_user_data	void *	このデータ・ポインタは、外部ルーチンが必要とする任意のコンテキスト・データの使用により設定できます。
_executionMode	a_sql_uint32	**External_UDF_Execution_Mode** オプションにより要求されたデバッグ／トレース・レベルを示します。これは読み込み専用フィールドです。
current_state	a_sql_uint32	*current_state* 属性は、コンテキストの現在の実行モードを反映しています。これは `_describe_extfn` などの関数から確認でき、次に行う処理の判断に使用できます。

説明

`a_v4_extfn_proc_context` 構造体では、サーバおよび UDF からのコンテキスト情報の保持に加えて、UDF からサーバへのコールバック呼び出しによる処理を実行できます。UDF はプライベート・データをこの構造体の `_user_data` メンバに格納できます。この構造体のインスタンスはサーバによって `a_v4_extfn_proc` メソッドの関数に渡されます。ユーザ・データは、サーバが注釈状態に達するまでは維持されません。

get_value

v4 API メソッド `get_value` を使用して、SQL クエリで UDF に渡された入力引数の値を取得します。

宣言

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

使用法
get_value API は、UDF が受け取ったそれぞれの入力引数の値を evaluate メソッドで取得するために使用します。引数のデータ型の範囲が狭い場合 (32K より大)、get_value の呼び出しで十分に引数値全体を取得できます。

get_value API は、arg_handle ポインタにアクセスできるどの API からでも呼び出すことができます。これには、a_v4_table_context をパラメータで受け取る API 関数が含まれます。a_v4_table_context のメンバ変数には args_handle があり、これを使用できます。

固定長データ型の場合は、どの型についても、返される値でデータを取得でき、すべてのデータを取得するために連続して呼び出す必要はありません。プロデューサは、get_value メソッドの呼び出しで返る全体の最大長を判断できます。固定長データ型はすべて、連続した単一のバッファに収まることが保証されます。可変長データの場合は、上限はプロデューサに応じて変わります。

固定長でないデータ型の場合、データの長さによっては、データを取得するために get_blob メソッドで BLOB を作成することが必要な場合があります。BLOB オブジェクトが必要かどうかは、get_value で戻る値に対してマクロ **EXTFN_IS_INCOMPLETE** を使用することで判断できます。**EXTFN_IS_INCOMPLETE** が true の場合、BLOB が必要です。

入力引数がテーブルの場合、その型は **AN_EXTFN_TABLE** です。この型の引数については、open_result_set メソッドを使用して結果セットを作成してテーブルから値を読み込む必要があります。

UDF で、_evaluate_extfn API の呼び出しより前に引数の値が必要な場合は、_describe_extfn API を実装する必要があります。_describe_extfn API からは、describe_parameter_get メソッドを使用して定数式の値を取得できます。

パラメータ

パラメータ	説明
arg_handle	コンシューマが提供するコンテキスト・ポインタ。
arg_num	値を取得する対象の引数のインデックス。引数のインデックスの先頭の値は 1 です。
value	指定の引数の値。

戻り値
成功の場合は 1、それ以外の場合は 0。

an_extfn_value 構造体

an_extfn_value 構造体は、get_value API が返す入力引数の値を表します。

次のコードは、**an_extfn_value** 構造体の宣言を示します。

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

次の表は、get_value メソッドを呼び出した後で **an_extfn_value** オブジェクトが返す値の内容を示します。

get_value API が返す値	EXTFN_IS_IN-COMPLETE	total_len	piece_len	data
NULL	FALSE	0	0	NULL
空の文字列	FALSE	0	0	非 NULL
Size < MAX_UINT32	FALSE	実際の長さ	実際の長さ	非 NULL
size < MAX_UINT32	TRUE	実際の長さ	0	非 NULL
size >= MAX_UINT32	TRUE	MAX_UINT32	0	非 NULL

an_extfn_value の type フィールドは、値のデータ型を表します。入力引数にテーブルがある UDF の場合、その引数のデータ型は **DT_EXTFN_TABLE** です。v4 のテーブル UDF では、remain_len フィールドは使用しません。

参照：

- _evaluate_extfn (315 ページ)
- テーブル・コンテキスト (a_v4_extfn_table_context) (337 ページ)
- _describe_extfn (316 ページ)
- *describe_parameter_get (264 ページ)

get_value_is_constant

v4 API メソッド `get_value_is_constant` を使用して、指定の入力引数が定数かどうかを判断します。

宣言

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

使用法

UDF は、指定された引数が定数かどうかを示す情報を要求できます。この関数は、たとえば、評価関数を呼び出すたびに行うのではなく、`_evaluate_extfn` 関数の最初の呼び出しで 1 回行うだけで済む場合など、UDF の最適化に便利です。

パラメータ

パラメータ	説明
arg_handle	サーバの引数のハンドル。
arg_num	取得する入力引数のインデックス値。インデックス値は 1 ～ N の間です。
value_is_constant	定数かどうかを格納する出力パラメータ。

戻り値

成功の場合は 1、それ以外の場合は 0。

参照：

- _evaluate_extfn (315 ページ)

set_value

v4 API メソッド `set_value` を使用して、結果セットに含まれるカラムの数とデータの読み込み方法をコンシューマに伝達します。

宣言

```
short set_value(
       void *           arg_handle,
       a_sql_uint32     arg_num,
       an_extfn_value *value
       )
```

使用法
このメソッドは、UDF が `_evaluate_extfn` API で使用します。UDF は、`set_value` メソッドを呼び出して、結果セットに含まれるカラムの数と、UDF がサポートする `a_v4_extfn_table_func` の関数のセットをコンシューマに伝える必要があります。

UDF は `set_value` API に対し、`_evaluate_extfn` API を通じた適切な **arg_handle** ポインタか、または `a_v4_extfn_table_context` 構造体の **args_handle** メンバを渡します。

v4 のテーブル UDF については、`set_value` メソッドの **value** 引数は型 *DT_EXTFN_TABLE* である必要があります。

パラメータ

パラメータ	説明
arg_handle	コンシューマが提供するコンテキスト・ポインタ。
arg_num	値を設定する対象の引数のインデックス。サポートされている引数は 0 のみです。
value	指定の引数の値。

戻り値
成功の場合は 1、それ以外の場合は 0。

参照：
- _evaluate_extfn (315 ページ)
- テーブル関数 (a_v4_extfn_table_func) (346 ページ)
- テーブル・コンテキスト (a_v4_extfn_table_context) (337 ページ)

get_is_cancelled

v4 API メソッド `get_is_cancelled` を使用して、文の実行がキャンセルされたかどうかを判断します。

宣言
```
short get_is_cancelled(
       a_v4_extfn_proc_context *          cntxt,

       )
```

使用法
UDF エントリ・ポイントが長期間 (かなりの秒数) にわたって動作を実行する場合、可能であれば、ユーザが現在の文を中断したかどうかを確認するために、1 秒または 2 秒ごとに `get_is_cancelled` コールバックを呼び出します。 文が中断されている場合、0 以外の値が返され、UDF エントリ・ポイントはただちに処理を戻す必要があります。`_finish_extfn` 関数を呼び出して、必要なクリーンアップを実行します。その後は、他の UDF エントリ・ポイントを呼び出さないでください。

パラメータ

パラメータ	説明
cntxt	プロシージャ・コンテキスト・オブジェクト。

戻り値
文が中断されている場合は 0 以外の値。

参照：
- スカラ UDF と集合 UDF のコールバック関数 (94 ページ)

set_error

v4 API メソッド `set_error` を使用して、サーバにエラーを返します。このエラーは最終的にはユーザに返されます。

宣言
```
void set_error(
       a_v4_extfn_proc_context *          cntxt,
       a_sql_uint32                       error_number,
       const char                         *error_desc_string
       )
```

使用法
UDF エントリ・ポイントでエラーが発生し、ユーザにエラー・メッセージを表示して現在の文を停止する必要がある場合には、`set_error` API を呼び出します。`set_error` API を呼び出すと、現在の文がロールバックされ、ユーザには `"Error raised by user-defined function: <error_desc_string>"` と表示されます。SQLCODE は指定の <error_number> の負数です。

既存のエラーとの競合を避けるため、UDF は 17000 ～ 99999 の間のエラー番号を生成する必要があります。この範囲にない値を指定した場合、文はロールバックされますが、エラー・メッセージは `"Invalid error raised by user-`

`defined function: (<error_number>) <error_desc_string>"` となり、SQLCODE は -1577 となります。**error_desc_string** の最大長は 140 文字です。

UDF エントリ・ポイントでは、`set_error` を呼び出した後は、直ちに処理を戻す必要があります。最終的には `_finish_extfn` 関数が呼び出され、必要なクリーンアップが実行されます。その後は、他の UDF エントリ・ポイントを呼び出さないでください。

パラメータ

パラメータ	説明
cntxt	プロシージャ・コンテキスト・オブジェクト。
error_number	設定するエラー番号。
error_desc_string	使用するメッセージ文字列。

参照：

- スカラ UDF と集合 UDF のコールバック関数 (94 ページ)

log_message

v4 API メソッド `log_message` を使用して、サーバのメッセージ・ログにメッセージを送信します。

宣言

```
short log_message(
       const char         *msg,
       short              msg_length
       )
```

使用法

`log_message` メソッドでは、メッセージ・ログにメッセージを書き込みます。メッセージ文字列は、表示可能な 255 バイト以下のテキスト文字列とする必要があります。これより長いメッセージはトランケートされます。

パラメータ

パラメータ	説明
msg	ログに記録するメッセージ文字列。
msg_length	メッセージ文字列の長さ。

参照：
- エラー・チェックと呼び出しトレーシングの制御 (33 ページ)

convert_value

v4 API メソッド convert_value を使用して、データ型を変換します。

宣言
```
short convert_value(
       an_extfn_value *input,
       an_extfn_value *output
       )
```

使用法
. convert_value API の主な用途は、DT_DATE、DT_TIME、および DT_TIMESTAMP と DT_TIMESTAMP_STRUCT の間の変換です。 入力と出力の an_extfn_value をこの関数に渡します。

入力パラメータ

パラメータ	説明
an_extfn_value.data	入力データのポインタ。
an_extfn_value.total_len	入力データの長さ。
an_extfn_value.type	入力の DT_ データ型。

出力パラメータ

パラメータ	説明
an_extfn_value.data	UDF が指定する出力データのポインタ。
an_extfn_value.piece_len	出力データの最大長。
an_extfn_value.total_len	サーバが設定する変換済みデータの長さ。
an_extfn_value.type	希望の出力の DT_ データ型。

戻り値
成功の場合は 1、それ以外の場合は 0。

参照：
- get_value (319 ページ)

get_option

v4 API メソッド `get_option` では、設定可能なオプションの値を取得します。

宣言

```
short get_option(
a_v4_extfn_proc_context * cntxt,
char *option_name,
an_extfn_value *output
)
```

パラメータ

パラメータ	説明
cntxt	プロシージャ・コンテキスト・オブジェクト。
option_name	取得するオプションの名前。
output	• `an_extfn_value.data` – UDF が指定する出力データのポインタ。 • `an_extfn_value.piece_len` – 出力データの最大長。 • `an_extfn_value.total_len` – サーバが設定する変換済み出力の長さ。 • `an_extfn_value.type` – サーバが設定する値のデータ型。

戻り値
成功の場合は 1、それ以外の場合は 0。

参照：
- 外部関数のプロトタイプ (105 ページ)
- 外部プロシージャ・コンテキスト (a_v4_extfn_proc_context) (317 ページ)

alloc

v4 API メソッド `alloc` では、メモリのブロックを確保します。

宣言

```
void*alloc(
     a_v4_extfn_proc_context *cntxt,
     size_t len
     )
```

使用法
長さが **len** 以上のメモリ・ブロックを確保します。返されるメモリは 8 バイト単位で揃えられています。メモリの確保には `alloc()` メソッドのみを使用することを

おすすめします。そうすると、外部ルーチンが使用しているメモリの量をサーバが把握できます。サーバでは、その他のメモリ使用の調整、メモリ・リークの追跡、高度な診断とモニタリングの提供を行うことができます。

メモリ・トラッキングは、**external_UDF_execution_mode** が値 1 または 2 (検証モードまたはトレース・モード) に設定されている場合にのみ有効になります。

パラメータ

パラメータ	説明
cntxt	プロシージャ・コンテキスト・オブジェクト。
len	確保するメモリのバイト単位の長さ。

参照：

- free (328 ページ)
- メモリ・トラッキングの有効化 (151 ページ)

free

v4 API メソッド `free` では、確保したメモリのブロックを解放します。

宣言

```
void free(
     a_v4_extfn_proc_context *cntxt,
     void *mem
)
```

使用法

`alloc()` により指定の存続期間で確保したメモリを解放します。

メモリ・トラッキングは、**external_UDF_execution_mode** が値 1 または 2 (検証モードまたはトレース・モード) に設定されている場合にのみ有効になります。

パラメータ

パラメータ	説明
cntxt	プロシージャ・コンテキスト・オブジェクト。
mem	`alloc` メソッドで確保したメモリのポインタ。

参照：

- alloc (327 ページ)
- メモリ・トラッキングの有効化 (151 ページ)

open_result_set

v4 API メソッド `open_result_set` では、テーブル値の結果セットを開きます。

宣言

```
short open_result_set(
a_v4_extfn_proc_context *cntxt,
a_v4_extfn_table *table,
a_v4_extfn_table_context **result_set
)
```

使用法

`open_result_set` では、テーブル値の結果セットを開きます。UDF は、`DT_EXTFN_TABLE` 型の入力パラメータからローを読み込むための結果セットを開くことができます。サーバ (または別の UDF) は、UDF からのローを読み込むための結果セットを開くことができます。

パラメータ

パラメータ	説明
cntxt	プロシージャ・コンテキスト・オブジェクト。
table	結果セットを開く対象のテーブル・オブジェクト。
result_set	開いた結果セットに設定される出力パラメータ。

戻り値

成功の場合は 1、それ以外の場合は 0。

`open_result_set` の使用例については、v4 API メソッド `fetch_block` および `fetch_into` の説明を参照してください。

参照：

- 外部プロシージャ・コンテキスト (a_v4_extfn_proc_context) (317 ページ)
- fetch_into (340 ページ)
- fetch_block (342 ページ)

close_result_set

v4 API メソッド `close_result_set` では、開いた結果セットを閉じます。

宣言

```
short close_result_set(
    a_v4_extfn_proc_context *cntxt,
```

```
    a_v4_extfn_table_context *result_set
)
```

使用法
close_result_set は、それぞれの結果セットに対して 1 回のみ呼び出すことができます。

パラメータ

パラメータ	説明
cntxt	プロシージャ・コンテキスト・オブジェクト。
result_set	閉じる結果セット。

戻り値
成功の場合は 1、それ以外の場合は 0。

get_blob

v4 API メソッド get_blob を使用して、入力の BLOB パラメータを取得します。

宣言

```
short get_blob(
       void             *arg_handle,
       a_sql_uint32     arg_num,
       a_v4_extfn_blob **blob
       )
```

使用法
get_blob は、get_value() を呼び出した後で使用し、BLOB 入力パラメータを取得します。get_value() で *piece_len* < *total_len* の場合に、返った値のデータの読み込みに BLOB オブジェクトが必要かどうかを判断するには、マクロ EXTFN_IS_INCOMPLETE を使用します。 BLOB オブジェクトは出力パラメータで返り、呼び出し元が所有します。

get_blob で取得する BLOB ハンドルを使用して、BLOB の内容を読み込むことができます。このメソッドは BLOB オブジェクトが格納されたカラムに対してのみ使用します。

パラメータ

パラメータ	説明
arg_handle	サーバの引数へのハンドル。

パラメータ	説明
arg_num	引数を表す番号 1 ～ N。
blob	BLOB オブジェクトが入る出力引数。

戻り値
成功の場合は 1、それ以外の場合は 0。

参照：
- 外部プロシージャ・コンテキスト (a_v4_extfn_proc_context) (317 ページ)
- get_value (319 ページ)

set_cannot_be_distributed

v4 API メソッド `set_cannot_be_distributed` では、ライブラリ・レベルで分散の基準を満たす場合でも、UDF レベルで分散を無効にできます。

宣言
```
void set_cannot_be_distributed( a_v4_extfn_proc_context *cntxt)
```

使用法
デフォルトの動作では、ライブラリが分散可能な場合、UDF も分散可能です。UDF では、`set_cannot_be_distributed` を使用して、分散を無効にするという判断をサーバにプッシュできます。

ライセンス情報 (a_v4_extfn_license_info)

Sybase デザイン・パートナーは、`a_v4_extfn_license_info` 構造体を使用して、会社名、ライブラリのバージョン情報、Sybase が提供するライセンス・キーなど、ライブラリレベルでの UDF のライセンス検証を定義します。

実装
```
typedef struct an_extfn_license_info {
    short     version;
} an_extfn_license_info;

typedef struct a_v4_extfn_license_info {
    an_extfn_license_info version;

    const char        name[255];
    const char        info[255];
    void *        key;
} a_v4_extfn_license_info;
```

データ・メンバの概要

データ・メンバ	説明
version	内部でのみ使用。1 に設定する必要があります。
name	UDF で会社名として設定する値。
info	UDF でライブラリ・バージョンやビルド番号などの追加的なライブラリ情報として設定する値。
key	(Sybase パートナーのみ) Sybase が提供するライセンス・キー。キーは 26 文字の配列です。

オプティマイザの推定 (a_v4_extfn_estimate)

`a_v4_extfn_estimate` 構造体は推定を表します。推定には値と信頼レベルが含まれます。

実装

```
typedef struct a_v4_extfn_estimate {
    double    value;
    double    confidence;
} a_v4_extfn_estimate;
```

データ・メンバとデータ型の概要

データ・メンバ	データ型	説明
value	double	推定値。
confidence	double	この推定に該当する信頼レベル。信頼レベルは 0.0 ～ 1.0 の間で変わります。0.0 は推定が無効であることを表し、1.0 は推定が確実であることを表します。

ORDER BY リスト (a_v4_extfn_orderby_list)

a_v4_extfn_orderby_list 構造体はテーブルの ORDER BY プロパティを表します。

実装

```
typedef struct a_v4_extfn_orderby_list {
    a_sql_uint32        number_of_elements;
    a_v4_extfn_order_el order_elements[1];      // there are
number_of_elements entries
} a_v4_extfn_orderby_list;
```

データ・メンバとデータ型の概要

データ・メンバ	データ型	説明
number_of_elements	a_sql_uint32	エントリの数。
order_elements[1]	a_v4_extfn_order_el	要素の順序。

説明

エントリは *number_of_elements* 個あり、それぞれのエントリには、要素が昇順と降順のどちらであるかを示すフラグと、該当するテーブル内の適切なカラムを示すカラム・インデックスがあります。

参照：

- カラム順序 (a_v4_extfn_order_el) (226 ページ)

PARTITION BY のカラム番号 (a_v4_extfn_partitionby_col_num)

a_v4_extfn_partitionby_col_num 列挙型はカラム番号を表し、UDF で SQL と同様の **PARTITION BY** のサポートを表明できます。

実装

```
typedef enum a_v4_extfn_partitionby_col_num {
    EXTFNAPIV4_PARTITION_BY_COLUMN_NONE = -1,        // NO PARTITION
BY
    EXTFNAPIV4_PARTITION_BY_COLUMN_ANY  = 0,         // PARTITION BY
ANY
                                // + INTEGER representing a specific
column ordinal
} a_v4_extfn_partitionby_col_num;
```

メンバの概要

a_v4_extfn_partitionby_col_num 列挙型のメンバ	値	説明
EXTFNAPIV4_PARTITION_BY_COLUMN_NONE	-1	NO PARTITION BY
EXTFNAPIV4_PARTITION_BY_COLUMN_ANY	0	PARTITION BY ANY。特定のカラム順序数を表す正の整数。
Column Ordinal Number	N > 0	パーティションの対象となるテーブル・カラム番号の順序数。

説明
UDF では、この構造体を使用して、パーティション分割とその対象カラムをコードで記述できます。

`a_v4_extfn_column_list number_of_columns` フィールドの設定には、この列挙型を使用します。UDF が PARTITION BY のサポートをサーバに伝達するときには、`number_of_columns` を、列挙値のいずれかか、リストのカラム順序数の番号を表す正の整数に設定します。たとえば、パーティション分割をサポートしないことをサーバに伝達する場合には、構造体を次のように作成します。

```
a_v4_extfn_column_list nopby = {
EXTFNAPIV4_PARTITION_BY_COLUMN_NONE,
0
};
```

メンバ値の *EXTFNAPIV4_PARTITION_BY_COLUMN_ANY* は、UDF が任意の形式のパーティション分割をサポートすることをサーバに伝えます。

パーティションの対象となる順序数のセットを記述するには、構造体を次のように作成します。

```
a_v4_extfn_column_list nopby = {
2,
3, 4
};
```

こうすると、順序数が 3 と 4 のカラム 2 つによるパーティション分割を表せます。

注意： この例は説明のみを目的としたものであり、正当なコードではありません。呼び出し元は整数 3 つ分の構造体を適切に確保する必要があります。

ロー (a_v4_extfn_row)

a_v4_extfn_row 構造体は、単一のローのデータを表します。

実装

```
/* a_v4_extfn_row - */
typedef struct a_v4_extfn_row {
    a_sql_uint32           *row_status;
    a_v4_extfn_column_data    *column_data;
} a_v4_extfn_row;
```

データ・メンバとデータ型の概要

データ・メンバ	データ型	説明
row_status	a_sql_uint32 *	ローのステータス。存在するローは 1、それ以外は 0 に設定します。
column_data	a_v4_extfn_column_data *	ローのカラム・データの配列。

説明
ローの構造体は、いくつかのカラムで構成される特定のローの情報を保持します。この構造体は個別のローのステータスを定め、そのローが持つ個別のカラムのポインタも保持しています。ローのステータスは、ローの存在を示すフラグです。ロー・ステータス・フラグはネストしたフェッチ呼び出しで変更でき、ロー・ブロック構造体の操作は必要ありません。

row_status フラグを 1 に設定すると、そのローが使用可能で結果セットに含めることができるということを表します。*row_status* を 0 に設定すると、そのローを無視する必要があることを表します。これは、TPF がフィルタの役割を果たしている場合に便利です。入力テーブル内のローを結果セットにパススルーするときに、ステータスを 0 に設定することによって一部のローをスキップできるからです。

参照：
- カラム・データ (a_v4_extfn_column_data) (224 ページ)

ロー・ブロック (a_v4_extfn_row_block)

a_v4_extfn_row_block 構造体は、複数のローで構成されるブロックのデータを表します。

実装

```
/* a_v4_extfn_row_block - */
typedef struct a_v4_extfn_row_block {
    a_sql_uint32        max_rows;
    a_sql_uint32        num_rows;
    a_v4_extfn_row  *row_data;
} a_v4_extfn_row_block;
```

データ・メンバとデータ型の概要

データ・メンバ	データ型	説明
max_rows	a_sql_uint32	このロー・ブロックが対応できるローの最大数。
num_rows	a_sql_uint32	ロー・ブロックが保持できるローの最大数以下である必要があります。
row_data	a_v4_extfn_row *	ロー・データのポインタ。

説明

ロー・ブロック構造体は、fetch_into メソッドと fetch_block メソッドで使用し、プロデューサでのデータの生成とコンシューマでの利用に対応しています。ローの最大数は、ロー・ブロックを確保した側が設定します。ローの数はプロデューサが正確に設定します。コンシューマは、プロデューサが生成したロー数を超えてデータを読み込まないようにする必要があります。

row_block 構造体の *max_rows* データ・メンバの値は、構造体を所有する側が決定します。たとえば、テーブル UDF が fetch_into を実装している場合、*max_rows* の値は、128K のメモリに収まるロー数としてサーバが決定します。一方、テーブル UDF が fetch_block を実装している場合、*max_rows* の値は、テーブル UDF が自ら決定します。

制約と制限事項

num_rows と *max_rows* は、どちらも 0 より大きな値です。*num_rows* は *max_rows* 以下である必要があります。有効なロー・ブロックについては、row_data フィールドは NULL 以外である必要があります。

テーブル (a_v4_extfn_table)

a_v4_extfn_table 構造体は、テーブルにデータを格納する方法と、コンシューマがそのデータをフェッチする方法を表します。

実装

```
typedef struct a_v4_extfn_table {
    a_v4_extfn_table_func       *func;
    a_sql_uint32                number_of_columns;
} a_v4_extfn_table;
```

データ・メンバとデータ型の概要

データ・メンバ	データ型	説明
func	a_v4_extfn_table_func *	このメンバは、コンシューマが結果データのフェッチに使用する一連の関数ポインタを保持します。
number_of_columns	a_sql_uint32 *	テーブル内のカラム数。

テーブル・コンテキスト (a_v4_extfn_table_context)

a_v4_extfn_table_context 構造体は、テーブルに対して開いた結果セットを表します。

実装

```
typedef struct a_v4_extfn_table_context {

//    size_t struct_size;

    /* fetch_into() - fetch into a specified row_block. This entry point
        is used when the consumer has a transfer area with a specific format.
       The fetch_into() function will write the fetched rows into the provided row block.
    */
    short (UDF_CALLBACK *fetch_into)(a_v4_extfn_table_context *cntxt,
a_v4_extfn_row_block *);

    /* fetch_block() - fetch a block of rows. This entry point is used
        when the consumer does not need the data in a particular format. For example,
        if the consumer is reading a result set and formatting it as HTML, the consumer
       does not care how the transfer area is layed out. The fetch_block() entry point is
        more efficient if the consumer does not need a specific layout.

        The row_block parameter is in/out. The first call should point to a NULL row
block.
        The fetch_block() call sets row_block to a block that can be consumed, and this
block
```

```
        should be passed on the next fetch_block() call.
    */
    short (UDF_CALLBACK *fetch_block)(a_v4_extfn_table_context *cntxt,
a_v4_extfn_row_block **row_block);

    /*  rewind() - this is an optional entry point. If NULL, rewind is not supported.
Otherwise,
        the rewind() entry point restarts the result set at the beginning of the table.
    */
    short (UDF_CALLBACK *rewind)(a_v4_extfn_table_context *);

    /* get_blob() - If the specified column has a blob object, return it.  The blob
         is returned as an out parameter and is owned by the caller.  This method should
         only be called on a column that contains a blob.  The helper macro
EXTFN_COL_IS_BLOB can
         be used to determine whether a column contains a blob.
    */
    short (UDF_CALLBACK *get_blob)(a_v4_extfn_table_context *cntxt,
                a_v4_extfn_column_data *col,
                a_v4_extfn_blob **blob);

   /* The following fields are reserved for future use and must be initialized to NULL.
*/
    void  *reserved1_must_be_null;
    void  *reserved2_must_be_null;
    void  *reserved3_must_be_null;
    void  *reserved4_must_be_null;
    void  *reserved5_must_be_null;

    a_v4_extfn_proc_context     *proc_context;
    void                        *args_handle;           // use in
a_v4_extfn_proc_context::get_value() etc.
    a_v4_extfn_table            *table;
    void                        *user_data;
    void                        *server_internal_use;

   /* The following fields are reserved for future use and must be initialized to NULL.
*/
    void  *reserved6_must_be_null;
    void  *reserved7_must_be_null;
    void  *reserved8_must_be_null;
    void  *reserved9_must_be_null;
    void  *reserved10_must_be_null;

} a_v4_extfn_table_context;
```

メソッドの概要

データ型	メソッド	説明
short	**fetch_into**	指定の `row_block` にフェッチします。
short	**fetch_block**	ローのブロックをフェッチします。
short	**rewind**	テーブルの先頭から結果セットを再度開始します。
short	**get_blob**	指定のカラムに BLOB オブジェクトが格納されている場合、BLOB オブジェクトを返します。

データ・メンバとデータ型の概要

データ・メンバ	データ型	説明
proc_context	a_v4_extfn_proc_context *	プロシージャ・コンテキスト・オブジェクトへのポインタ。UDF はこれを使用して、エラーの設定、ログ・メッセージの記録、キャンセルなどを実行できます。
args_handle	void *	サーバが指定する引数のハンドル。
table	a_v4_extfn_table *	開いた結果セット・テーブルのポインタ。a_v4_extfn_proc_context open_result_set の呼び出し後に設定されます。
_user_data	void *	このデータ・ポインタは、外部ルーチンが必要とする任意のコンテキスト・データの使用により設定できます。
server_internal_use	void *	内部でのみ使用。

説明

a_v4_extfn_table_context 構造体は、プロデューサとコンシューマの仲立ちとなる中間層の役割を果たし、コンシューマとプロデューサで別のフォーマットが必要な場合のデータの管理に役立ちます。

UDF は、a_v4_extfn_table_context を使用して、入力テーブル・パラメータからローを読み込むことができます。サーバまたは別の UDF は、a_v4_extfn_table_context を使用して、UDF の結果テーブルのローを読み込むことができます。

a_v4_extfn_table_context のメソッドはサーバが実装します。サーバはこれを生かすことで、障害となる不一致を解決できます。

参照：

- fetch_into (340 ページ)
- fetch_block (342 ページ)
- rewind (345 ページ)

fetch_into

v4 API メソッド fetch_into では、指定のロー・ブロックにデータをフェッチします。

宣言

```
short fetch_into(
a_v4_extfn_table_context *cntxt,
a_v4_extfn_row_block *)
```

使用法

fetch_into メソッドは、メモリ内でのデータの配置方法をプロデューサが関知していない場合に便利です。このメソッドは、コンシューマが転送領域を特定のフォーマットで保持している場合にエントリ・ポイントとして使用します。fetch_into() 関数はフェッチしたローを指定のロー・ブロックに書き込みます。このメソッドは a_v4_extfn_table_context 構造体に含まれています。

データ転送領域のメモリをコンシューマが保持し、その領域を使用するようプロデューサに要求する場合には、fetch_into を使用します。fetch_into を使用する場合、データ転送領域のセットアップはコンシューマが担い、その領域に対して必要なデータをコピーする処理はプロデューサが担います。

パラメータ

パラメータ	説明
cntxt	open_result_set API で取得したテーブル・コンテキスト・オブジェクト。
row_block	フェッチ先のロー・ブロック・オブジェクト。

戻り値

成功の場合は 1、それ以外の場合は 0。

UDF が値 1 を返した場合は、まだローが残っていることをコンシューマが認識し、fetch_into メソッドを再度呼び出す必要があります。一方、UDF が値 0 を返した場合は、他にローが残っていないことを表します。fetch_into メソッドを再度呼び出す必要はありません。

次のプロシージャ定義は、入力パラメータのテーブルを利用し、それを結果テーブルとして生成する TPF 関数の例です。両テーブルは、それぞれ v4 API メソッドの get_value と set_value で取得および返される SQL 値のインスタンスです。

```
CREATE PROCEDURE FETCH_EX( IN a INT, INT b TABLE( c1 INT ) )
    RESULT SET ( rc INT )
```

このプロシージャ定義には次の 2 つのテーブル・オブジェクトが含まれています。

- b という名前の入力テーブル・パラメータ
- 返される結果セット・テーブル

次の例は、呼び出し元 (ここではサーバ) が出力テーブルをフェッチする方法を示しています。サーバは fetch_into メソッドの使用を決定できます。入力テーブルは、呼び出された側 (ここでは TPF) がフェッチします。使用するフェッチ API は TPF が決定します。

```
SELECT rc from FETCH_EX( 1, TABLE( SELECT c1 from TABLE ) )
```

次の例が示すように、入力テーブルのフェッチや利用の前には、a_v4_extfn_proc 構造体の open_result_set API を使用してテーブル・コンテキストを確立する必要があります。open_result_set にはテーブル・オブジェクトが必要で、これは get_value API で取得できます。

```
    an_extfn_value    arg;
    ctx->get_value( args_handle, 3, &arg );

    if( arg.type != DT_EXTFN_TABLE ) {
        // handle error
    }

    a_v4_extfn_table_context    *rs = NULL;
    a_v4_extfn_table         *inTable = arg.data;
    ctx->open_result_set( ctx, inTable, &rs );
```

テーブル・コンテキストを作成したら、rs 構造体の fetch_into API を呼び出してローをフェッチします。

```
a_v4_extfn_row_block    *rb = // get a row block to hold a series of
INT values.
rs->fetch_into( rs, &rb )  // fetch the rows.
```

結果テーブルにローを生成する前には、テーブル・オブジェクトを作成して呼び出し元に返す必要があります。それには、a_v4_extfn_proc_context 構造体の set_value API を使用します。

この例が示すように、テーブル UDF は a_v4_extfn_table 構造体のインスタンスを作成する必要があります。テーブル UDF のそれぞれの呼び出しに対し、a_v4_extfn_table 構造体の別個のインスタンスを返します。テーブルには状態を示すフィールドがあり、現在のローと生成するロー数を保持できます。テーブルの状態はインスタンスのフィールドとして設定できます。

```
    typedef struct rg_table : a_v4_extfn_table {
        a_sql_uint32        rows_to_generate;
        a_sql_uint32        current_row;
    } my_table;
```

次の例では、ローを生成するごとに **current_row** をインクリメントして、生成するロー数に達するまで繰り返しています。生成するロー数に達すると、`fetch_into` は false を返し、ファイルの末尾であることを通知します。コンシューマは、テーブル UDF が実装する `fetch_into` API を呼び出します。`fetch_into` メソッドを呼び出すときに、コンシューマはテーブル・コンテキストとフェッチ先のロー・ブロックを渡します。

```
     rs->fetch_into( rs, &rb )

short UDF_CALLBACK my_table_func_fetch_into(
    a_v4_extfn_table_context *tctx,
    a_v4_extfn_row_block *rb)
/****************************************/
{
    my_table *myTable = tctx->table;

    if( rgTable->current_row < rgTable->rows_to_generate ) {
            // Produce the row...
    rgTable->current_row++;
    return 1;
    }

    return 0;
}
```

参照：

- fetch_into メソッド (146 ページ)
- テーブル・コンテキスト (a_v4_extfn_table_context) (337 ページ)
- ロー・ブロック (a_v4_extfn_row_block) (336 ページ)
- 外部プロシージャ・コンテキスト (a_v4_extfn_proc_context) (317 ページ)
- get_value (319 ページ)
- set_value (322 ページ)
- テーブル (a_v4_extfn_table) (337 ページ)

fetch_block

v4 API メソッド `fetch_block` では、ローのブロックをフェッチします。

宣言

```
short fetch_block(
a_v4_extfn_table_context *cntxt,
a_v4_extfn_row_block **row_block)
```

使用法

`fetch_block` メソッドは、コンシューマがデータについて特定のフォーマットを必要としていない場合にエントリ・ポイントとして使用します。`fetch_block` ではプロデューサに対し、データ転送領域を作成してその領域の

ポインタを渡すよう要求します。コンシューマはこのメモリを保持し、この領域からデータをコピーする必要があります。

コンシューマが特定のレイアウトを必要としない場合には、fetch_block の方が効率的です。 fetch_block の呼び出しでは、データを格納できるロー・ブロックを fetch_block に設定し、次の fetch_block 呼び出しでもそのブロックを渡します。このメソッドは a_v4_extfn_table_context 構造体に含まれています。

パラメータ

パラメータ	説明
cntxt	テーブル・コンテキスト・オブジェクト。
row_block	入力／出力パラメータ。最初の呼び出しでは、必ず NULL の **row_block** を指します。

fetch_block が呼び出され、**row_block** が NULL を指している場合には、UDF は a_v4_extfn_row_block 構造体を確保する必要があります。

戻り値
成功の場合は 1、それ以外の場合は 0。

UDF が値 1 を返した場合は、まだローが残っていることをコンシューマが認識し、fetch_block メソッドを再度呼び出します。一方、UDF が値 0 を返した場合は、他にローが残っていないことを表します。fetch_block メソッドを再度呼び出す必要はありません。

次のプロシージャ定義は、入力パラメータのテーブルを利用し、それを結果テーブルとして生成する TPF 関数の例です。両テーブルは、それぞれ v4 API メソッドの get_value と set_value で取得および返される SQL 値のインスタンスです。

```
CREATE PROCEDURE FETCH_EX( IN a INT, INT b TABLE( c1 INT ) )
    RESULT SET ( rc INT )
```

このプロシージャ定義には次の 2 つのテーブル・オブジェクトが含まれています。

- b という名前の入力テーブル・パラメータ
- 返される結果セット・テーブル

次の例は、呼び出し元 (ここではサーバ) が出力テーブルをフェッチする方法を示しています。サーバは fetch_block メソッドの使用を決定できます。入力テーブルは、呼び出された側 (ここでは TPF) がフェッチします。使用するフェッチ API は TPF が決定します。

```
SELECT rc from FETCH_EX( 1, TABLE( SELECT c1 from TABLE ) )
```

次の例が示すように、入力テーブルのフェッチや利用の前には、a_v4_extfn_proc 構造体の open_result_set API を使用してテーブル・コンテキストを確立する必要があります。open_result_set にはテーブル・オブジェクトが必要で、これは get_value API で取得できます。

```
    an_extfn_value    arg;
    ctx->get_value( args_handle, 3, &arg );

    if( arg.type != DT_EXTFN_TABLE ) {
        // handle error
    }

    a_v4_extfn_table_context    *rs = NULL;
    a_v4_extfn_table                  *inTable = arg.data;
    ctx->open_result_set( ctx, inTable, &rs );
```

テーブル・コンテキストを作成したら、rs 構造体の fetch_block API を呼び出してローをフェッチします。

```
a_v4_extfn_row_block    *rb =  // get a row block to hold a series of
INT values.
rs->fetch_block( rs, &rb )  // fetch the rows.
```

結果テーブルにローを生成する前には、テーブル・オブジェクトを作成して呼び出し元に返す必要があります。それには、a_v4_extfn_proc_context 構造体の set_value API を使用します。

この例が示すように、テーブル UDF は a_v4_extfn_table 構造体のインスタンスを作成する必要があります。テーブル UDF のそれぞれの呼び出しに対し、a_v4_extfn_table 構造体の別個のインスタンスを返します。テーブルには状態を示すフィールドがあり、現在のローと生成するロー数を保持できます。テーブルの状態はインスタンスのフィールドとして設定できます。

```
    typedef struct rg_table : a_v4_extfn_table {
        a_sql_uint32        rows_to_generate;
        a_sql_uint32        current_row;
    } my_table;
```

参照：

- fetch_block メソッド (145 ページ)
- テーブル・コンテキスト (a_v4_extfn_table_context) (337 ページ)
- ロー・ブロック (a_v4_extfn_row_block) (336 ページ)
- 外部プロシージャ・コンテキスト (a_v4_extfn_proc_context) (317 ページ)
- get_value (319 ページ)
- set_value (322 ページ)
- open_result_set (329 ページ)
- テーブル (a_v4_extfn_table) (337 ページ)

rewind

v4 API メソッド rewind を使用して、テーブルの先頭から結果セットを再度開始します。

宣言

```
short rewind(
       a_v4_extfn_table_context           *cntxt,
       )
```

使用法

開いている結果セットの rewind メソッドを呼び出して、テーブルを先頭までリワインドします。UDF が入力テーブルをリワインドするときには、**EXTFNAPIV4_STATE_OPTIMIZATION** 状態の間に **EXTFNAPIV4_DESCRIBE_PARM_TABLE_REQUEST_REWIND** パラメータを使用してプロデューサに伝える必要があります。

rewind() はオプションのエントリ・ポイントです。NULL の場合、リワインドはサポートされていません。それ以外の場合、rewind() エントリ・ポイントによって、結果セットはテーブルの先頭から再度開始されます。

パラメータ

パラメータ	説明
cntxt	テーブル・コンテキスト・オブジェクト。

戻り値

成功の場合は 1、それ以外の場合は 0。

参照：

- クエリ最適化状態 (139 ページ)
- 実行状態 (143 ページ)
- EXTFNAPIV4_DESCRIBE_PARM_TABLE_REQUEST_REWIND 属性 (Set) (298 ページ)

get_blob

v4 API メソッド get_blob を使用して、指定のカラムの BLOB オブジェクトを取得します。

宣言

```
 short get_blob(
a_v4_extfn_table_context *cntxt,
```

```
a_v4_extfn_column_data *col,
a_v4_extfn_blob **blob
)
```

使用法
BLOB は出力パラメータで返り、呼び出し元が所有します。 このメソッドは BLOB が格納されたカラムに対してのみ使用します。

カラムに BLOB が格納されているかどうかは、ヘルパー・マクロ EXTFN_COL_IS_BLOB を使用して判断します。ヘッダ・ファイル extfnapiv4.h での EXTFN_COL_IS_BLOB の宣言は次のとおりです。

```
#define EXTFN_COL_IS_BLOB(c, n)        (c[n].blob_handle != NULL)
```

パラメータ

パラメータ	説明
cntxt	テーブル・コンテキスト・オブジェクト。
col	BLOB を取得する対象のカラム・データ・ポインタ。
blob	成功の場合は、指定のカラムに該当する BLOB オブジェクトが格納されます。

戻り値
成功の場合は 1、それ以外の場合は 0。

参照：
- テーブル・コンテキスト (a_v4_extfn_table_context) (337 ページ)

テーブル関数 (a_v4_extfn_table_func)

コンシューマは a_v4_extfn_table_func 構造体を使用してプロデューサから結果を取得します。

実装

```
typedef struct a_v4_extfn_table_func {
//    size_t struct_size;

   /*  Open a result set. The UDF can allocate any resources needed
for the result set.
    */
    short (UDF_CALLBACK *_open_extfn)(a_v4_extfn_table_context *);

   /*  Fetch rows into a provided row block. The UDF should implement
this method if it does
```

```
        not have a preferred layout for its transfer area.
    */
   short (UDF_CALLBACK *_fetch_into_extfn)(a_v4_extfn_table_context
*, a_v4_extfn_row_block

*row_block);

   /*  Fetch a block that is allocated and configured by the UDF. The
UDF should implement this
        method if it has a preferred layout of the transfer area.
    */
    short (UDF_CALLBACK *_fetch_block_extfn)
(a_v4_extfn_table_context *, a_v4_extfn_row_block

**row_block);

   /*  Restart a result set at the beginning of the table. This is an
optional entry point.
    */
    short (UDF_CALLBACK *_rewind_extfn)(a_v4_extfn_table_context *);

    /*  Close a result set. The UDF can release any resources
allocated for the result set.
    */
    short (UDF_CALLBACK *_close_extfn)(a_v4_extfn_table_context *);

    /* The following fields are reserved for future use and must be
initialized to NULL. */
    void *_reserved1_must_be_null;
    void *_reserved2_must_be_null;

} a_v4_extfn_table_func;
```

メソッドの概要

メソッド	データ型	説明
_open_extfn	void	結果セットを開いてローのフェッチを開始するために、サーバが呼び出します。UDF は結果セットに必要なリソースを確保できます。
_fetch_into_extfn	short	指定のロー・ブロックにローをフェッチします。UDF は、転送領域のレイアウトについての意向が特にない場合にこのメソッドを実装します。
_fetch_block_extfn	short	UDF が確保および設定したブロックにフェッチします。UDF は、転送領域のレイアウトについて意向がある場合にこのメソッドを実装します。
_rewind_extfn	void	フェッチをテーブルの先頭から再度開始するためにサーバが呼び出すオプションの関数。

メソッド	データ型	説明
_close_extfn	void	結果セットを閉じてローのフェッチを終了するために、サーバが呼び出します。UDF は結果セット用として確保したリソースを解放できます。
_reserved1_must_be_null	void	今後のために予約済み。NULL に初期化する必要があります。
_reserved1_must_be_null	void	今後のために予約済み。NULL に初期化する必要があります。

説明
a_v4_extfn_table_func 構造体では、テーブルから結果をフェッチするために使用するメソッドを定義します。

参照：
- テーブル (a_v4_extfn_table) (337 ページ)
- テーブル・コンテキスト (a_v4_extfn_table_context) (337 ページ)
- _open_extfn (348 ページ)
- _fetch_into_extfn (349 ページ)
- _fetch_block_extfn (350 ページ)
- _rewind_extfn (350 ページ)
- _close_extfn (351 ページ)

_open_extfn

サーバは v4 API メソッド _open_extfn を呼び出してローのフェッチを開始します。

宣言
```
void _open_extfn(
     a_v4_extfn_table_context *cntxt,
)
```

使用法
UDF はこのメソッドを使用して、結果セットを開き、サーバに結果を送信するために必要なリソース (たとえばストリーム) を確保します。

パラメータ

パラメータ	説明
cntxt	プロシージャ・コンテキスト・オブジェクト。

参照：

- テーブル・コンテキスト (a_v4_extfn_table_context) (337 ページ)

_fetch_into_extfn

v4 API メソッド `_fetch_into_extfn` では、指定のロー・ブロックにローをフェッチします。

宣言

```
short _fetch_into_extfn(
    a_v4_extfn_table_context *cntxt,
    a_v4_extfn_row_block *row_block
    )
```

使用法

UDF は、転送領域のレイアウトについての意向が特にない場合に、このメソッドを実装する必要があります。

パラメータ

パラメータ	説明
cntxt	プロシージャ・コンテキスト・オブジェクト。
row_block	フェッチ先のロー・ブロック・オブジェクト。

戻り値

成功の場合は 1、それ以外の場合は 0。

参照：

- テーブル・コンテキスト (a_v4_extfn_table_context) (337 ページ)
- ロー・ブロック (a_v4_extfn_row_block) (336 ページ)

_fetch_block_extfn

v4 API メソッド `_fetch_block_extfn` では、UDF が確保および設定したブロックにフェッチします。

宣言

```
short _fetch_block_extfn(
a_v4_extfn_table_context *cntxt,
a_v4_extfn_row_block **
)
```

使用法
UDF は、転送領域のレイアウトについての意向がある場合に、このメソッドを実装する必要があります。

パラメータ

パラメータ	説明
cntxt	プロシージャ・コンテキスト・オブジェクト。
row_block	フェッチ先のロー・ブロック・オブジェクト。

戻り値
成功の場合は 1、それ以外の場合は 0。

参照：
- テーブル・コンテキスト (a_v4_extfn_table_context) (337 ページ)
- ロー・ブロック (a_v4_extfn_row_block) (336 ページ)

_rewind_extfn

v4 API メソッド _rewind_extfn では、結果セットをテーブルの先頭から再度開始します。

宣言

```
void _rewind_extfn(
a_v4_extfn_table_context *cntxt,
)
```

使用法
この関数はオプションのエントリ・ポイントです。UDF は、結果テーブルが先頭までリワインドされる場合に `_rewind_extfn` メソッドを実装します。UDF は、コストがかからず効率のよい方法でリワインド機能を提供できる場合にのみ、このメソッドの実装を検討します。

_rewind_extfn メソッドを実装することを選択した UDF は、そのことをコンシューマに通知するために、**EXTFNAPIV4_STATE_OPTIMIZATION** 状態のときに **EXTFNAPIV4_DESCRIBE_PARM_TABLE_HAS_REWIND** パラメータを引数 0 に対し設定する必要があります。

UDF は、リワインド機能を提供しないことも決定できます。その場合は、サーバがこれを補い、機能を提供します。

注意： サーバは、_rewind_extfn メソッドの呼び出しによるリワインドの実行を行わないことも選択できます。

パラメータ

パラメータ	説明
cntxt	プロシージャ・コンテキスト・オブジェクト。

戻り値
戻り値はありません。

参照：

- EXTFNAPIV4_DESCRIBE_PARM_TABLE_REQUEST_REWIND 属性 (get) (282 ページ)
- EXTFNAPIV4_DESCRIBE_PARM_TABLE_REQUEST_REWIND 属性 (Set) (298 ページ)
- EXTFNAPIV4_DESCRIBE_PARM_TABLE_HAS_REWIND 属性 (set) (300 ページ)
- EXTFNAPIV4_DESCRIBE_PARM_TABLE_HAS_REWIND 属性 (get) (283 ページ)
- クエリ処理の状態 (135 ページ)
- 実行状態 (a_v4_extfn_state) (312 ページ)
- テーブル・コンテキスト (a_v4_extfn_table_context) (337 ページ)

_close_extfn

サーバは v4 API メソッド _close_extfn を呼び出してローのフェッチを終了します。

宣言

```
void _close_extfn(
     a_v4_extfn_table_context *cntxt,
)
```

使用法
UDF は、フェッチが完了したときにこのメソッドを使用します。これによって結果セットを閉じ、結果セット用に確保したリソースを解放します。

パラメータ

パラメータ	説明
cntxt	プロシージャ・コンテキスト・オブジェクト。

参照：

- テーブル・コンテキスト (a_v4_extfn_table_context) (337 ページ)

a_v4_extfn の API トラブルシューティング

v4 API メソッドの describe_column、describe_parameter、describe_udf は、一般的なエラー・メッセージを返すことがあります。サーバ上に存在しない UDF を実行すると、"Could not execute statement" エラーが返ります。

describe_column の一般的なエラー

describe_column の get と set の呼び出しで返る一般的なエラーを示します。

get	set
EXTFNAPIV4_DESCRIBE_ BUFFER_SIZE_MISMATCH – **cntxt** または **describe_buffer** が NULL の場合か、**describe_buffer_length** が 0 の場合に返る get エラー。	EXTFNAPIV4_DESCRIBE_ BUFFER_SIZE_MISMATCH – **cntxt** または **describe_buffer** が NULL の場合か、**describe_buffer_length** が 0 の場合に返る set エラー。
EXTFNAPIV4_DESCRIBE_INVALID_STATE – **cntxt** パラメータが有効なコンテキストでない場合に返る get エラー。	EXTFNAPIV4_DESCRIBE_INVALID_STATE – **cntxt** パラメータが有効なコンテキストでない場合に返る set エラー。
EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER – 指定のパラメータ番号がプロシージャで有効な範囲内にない場合に返る get エラー。つまり、「パラメータ番号 **< 0**」の場合か、「パラメータ番号 **>** プロシージャのパラメータ数」の場合です。	EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER – 指定のパラメータ番号がプロシージャで有効な範囲内にない場合に返る set エラー。つまり、「パラメータ番号 **< 0**」の場合か、「パラメータ番号 **>** プロシージャのパラメータ数」の場合です。
EXTFNAPIV4_DESCRIBE_NON_TABLE_PARAMETER – **arg_num** がテーブル・パラメータでない場合に返る get エラー。	EXTFNAPIV4_DESCRIBE_NON_TABLE_PARAMETER – **arg_num** がテーブル・パラメータでない場合に返る set エラー。
EXTFNAPIV4_DESCRIBE_INVALID_COLUMN – カラム番号がテーブル・パラメータで有効な値でない場合に返る get エラー。	EXTFNAPIV4_DESCRIBE_INVALID_COLUMN – カラム番号がテーブル・パラメータで有効な値でない場合に返る set エラー。

get	set
EXTFNAPIV4_DESCRIBE_UNKNOWN_ATTRIBUTE – **describe_type** の値が、a_v4_extfn_describe_parm_type のいずれかの有効な記述タイプでない場合に返る get エラー。	EXTFNAPIV4_DESCRIBE_UNKNOWN_ATTRIBUTE – **describe_type** の値が、a_v4_extfn_describe_parm_type のいずれかの有効な記述タイプでない場合に返る set エラー。

describe_udf の一般的なエラー

describe_udf の get と set の呼び出しで返る一般的なエラーを示します。

get	set
EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER – **cntxt** 引数か **describe_buffer** 引数が NULL の場合、または **describe_buffer_length** が 0 の場合に返る get エラー。	EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER – **cntxt** 引数か **describe_buffer** 引数が NULL の場合、または **describe_buffer_length** が 0 の場合に返る set エラー。
EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER – **cntxt** パラメータが有効なコンテキストでない場合に返る get エラー。	EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER – **cntxt** パラメータが有効なコンテキストでない場合に返る set エラー。
EXTFNAPIV4_DESCRIBE_ UNKNOWN_ATTRIBUTE – **describe_type** の値が、a_v4_extfn_describe_udf_type のいずれかの記述タイプでない場合に返る get エラー。	EXTFNAPIV4_DESCRIBE_ UNKNOWN_ATTRIBUTE – **describe_type** の値が、a_v4_extfn_describe_udf_type のいずれかの記述タイプでない場合に返る set エラー。

describe_parameter の一般的なエラー

describe_parameter の get と set の呼び出しで返る一般的なエラーを示します。

get	set
EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER – **cntxt** か **describe_buffer** が NULL の場合、または **describe_buffer_length** が 0 の場合に返る get エラー。	EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER – **cntxt** か **describe_buffer** が NULL の場合、または **describe_buffer_length** が 0 の場合に返る set エラー。

get	set
EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER – **cntxt** パラメータが無効な場合に返る get エラー。	EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER – **cntxt** パラメータが無効な場合に返る set エラー。
EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER – 指定のパラメータ番号がプロシージャで有効な範囲内にない場合に返る get エラー。つまり、「パラメータ番号 < 0」の場合か、「パラメータ番号 > プロシージャのパラメータ数」の場合です。	EXTFNAPIV4_DESCRIBE_INVALID_PARAMETER – 指定のパラメータ番号がプロシージャで有効な範囲内にない場合に返る set エラー。つまり、「パラメータ番号 < 0」の場合か、「パラメータ番号 > プロシージャのパラメータ数」の場合です。
EXTFNAPIV4_DESCRIBE_ UNKNOWN_ATTRIBUTE – **describe_type** の値が a_v4_extfn_describe_parm_type で無効な記述タイプの場合に返る get エラー。	EXTFNAPIV4_DESCRIBE_ UNKNOWN_ATTRIBUTE – **describe_type** の値が a_v4_extfn_describe_parm_type で無効な記述タイプの場合に返る set エラー。
EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH – 要求された属性のサイズと指定された describe_buffer_length に不一致がある場合に返る get エラー。 a_sql_byte データ型など、固定長の属性については、サイズが一致する必要があります。char[] など、可変長の属性のデータ型については、指定のバッファは要求された属性の値を保持できる大きさ以上である必要があります。	EXTFNAPIV4_DESCRIBE_BUFFER_SIZE_MISMATCH – 要求された属性のサイズと指定された **describe_buffer_length** に不一致がある場合に返る set エラー。a_sql_byte データ型など、固定長の属性については、サイズが一致する必要があります。

参照：

- クエリ処理の状態 (135 ページ)

UDF がない場合に返るエラー

サーバ上に存在しない UDF を実行しようとするとエラーが返ります。

次のようなクエリを実行するとします。

```
select my_sum1(n_tabkey) from tabudf()
```

このとき、次のような状況だとします。

- `tabudf()` はテーブル UDF である。
- UDF `my_sum1()` はサーバ上に存在しない。

この場合、次のエラーが返ります。

```
Could not execute statement.
External procedures or functions are not allowed across server types.
SQLCODE=-1579, ODBC 3 State="HY000"
Line 1, column 1
```

UDF 用の外部環境

UDF が適切に定義されていないと、メモリ違反が発生する場合や、データベース・サーバの障害につながる場合があります。データベース・サーバの外部環境で UDF を実行できると、サーバへのこれらのリスクをなくすことができます。

外部環境で実行時例外が発生しても、サーバ・プロセスには影響が及びません。サーバは UDF の呼び出し元にエラーを通知し、その後の UDF の呼び出しで外部環境が再起動されます。

注意： 外部ランタイム環境には IQ_UDF ライセンスや IQ_IDA ライセンスは必要ありません。外部ランタイム環境には `a_v3_extfn` API や `a_v4_extfn` API は必要ありません。

データベース・サーバは UDF 向けに以下の外部ランタイム環境をサポートしています。

- CLR (Microsoft .NET 共通言語ランタイム)
- ESQL および ODBC (C/C++ Embedded SQL または ODBC サーバサイド要求)
- Java
- Perl
- PHP

引数を処理したりサーバに値を返したりするための API は、それぞれの環境ごとに異なります。たとえば、Java の外部環境では JDBC API を使用します。

システム・テーブル
システム・テーブル SYSEXTERNENV には、各外部環境を識別して起動するために必要な情報が格納されています。

システム・テーブル SYSEXTERNENVOBJECT には、非 Java の外部オブジェクトが格納されます。

SQL 文
次の SQL 構文を使用して、SYSEXTERNENV テーブルの外部環境の場所を設定および変更できます。

```
ALTER EXTERNAL ENVIRONMENT environment-name
    [ LOCATION location-string ]
```

データベース・サーバで使用するように外部環境が設定されたら、オブジェクトをデータベースにインストールし、それらのオブジェクトを外部環境内で使用するストアド・プロシージャおよび関数を作成することができます。オブジェクト、

ストアド・プロシージャ、およびストアド関数のインストール、作成、および使用方法は、Java クラスのインストール方法や、Java のストアド・プロシージャと関数の作成方法および使用方法と似ています。

外部環境のコメントを追加するには、次のような文を実行します。

```
COMMENT ON EXTERNAL ENVIRONMENT environment-name
 IS comment-string
```

Perl または PHP の外部オブジェクト (たとえば Perl スクリプト) をファイルまたは式からデータベースにインストールするには、次のような **INSTALL EXTERNAL OBJECT** 文を実行します。

```
INSTALL EXTERNAL OBJECT object-name-string
  [ update-mode ]
  FROM { FILE file-path | VALUE expression }
  ENVIRONMENT environment-name
```

インストールされている Perl または PHP の外部オブジェクトのコメントを追加するには、次のような文を実行します。

```
COMMENT ON EXTERNAL [ENVIRONMENT] OBJECT object-name-string
 IS comment-string
```

インストールされている Perl または PHP の外部オブジェクトをデータベースから削除するには、次のような **REMOVE EXTERNAL OBJECT** 文を使用します。

```
REMOVE EXTERNAL OBJECT object-name-string
```

データベースにインストールされた外部オブジェクトは、外部ストアド・プロシージャおよび関数の定義で使用できます (Java ストアド・プロシージャおよび関数を作成する現在のメカニズムに似ています)。

```
CREATE PROCEDURE procedure-name(...)
 EXTERNAL NAME '...'
 LANGUAGE environment-name

CREATE FUNCTION function-name(...)
 RETURNS ...
 EXTERNAL NAME '...'
 LANGUAGE environment-name
```

作成されたストアド・プロシージャおよび関数は、データベース内の他のストアド・プロシージャや関数と同じように使用できます。外部環境のストアド・プロシージャまたは関数が呼び出されると、データベース・サーバは、外部環境がまだ起動されていない場合はこれを自動的に起動し、外部環境がデータベースから外部オブジェクトをフェッチして実行するために必要な情報を送信します。実行結果の結果セットや戻り値は、必要に応じて返されます。

外部環境を要求に応じて起動または停止するには、**START EXTERNAL ENVIRONMENT** 文と **STOP EXTERNAL ENVIRONMENT** 文を次のように使用します。

```
 START EXTERNAL ENVIRONMENT environment-name
STOP EXTERNAL ENVIRONMENT environment-name
```

外部環境からの UDF の実行

CLR、ESQL と ODBC、Java、Perl、または PHP の外部環境での UDF の実行について説明します。

前提条件

ライセンスについての前提条件はありません。外部ランタイム環境には IQ_IDA ライセンスは必要ありません。外部ランタイム環境には `a_v3_extfn` API や `a_v4_extfn` API は必要ありません。

手順

1. データベース・サーバで使用する外部環境をセットアップします。
   ```
   ALTER EXTERNAL ENVIRONMENT environment-name
       [ LOCATION location-string ]
   ```
2. 外部オブジェクト (CLR、ESQL と ODBC、Java、Perl、または PHP) をデータベースにインストールします。
3. **CREATE PROCEDURE** 文と **CREATE FUNCTION** 文を使用して、外部環境内でこれらのオブジェクトを利用するストアド・プロシージャと関数を作成します。
4. ストアド・プロシージャまたは関数を参照します。ストアド・プロシージャはクエリの **FROM** 句で参照します。

参照：

- CLR 外部環境 (360 ページ)
- ESQL 外部環境と ODBC 外部環境 (363 ページ)
- Java 外部環境 (373 ページ)
- Perl 外部環境 (398 ページ)
- PHP 外部環境 (402 ページ)
- CREATE PROCEDURE 文 (Java UDF) (392 ページ)
- CREATE FUNCTION 文 (Java UDF) (394 ページ)

外部環境の制限

UDF 向けのすべての外部環境で、以下の制限が適用されます。

- **NO RESULT SET** オプションはサポートされない。
- **IN** パラメータのみがサポートされる。**INOUT/OUT** はサポートされない。
- LONG VARCHAR または LONG BINARY の結果値を持つ関数は使用できない。

参照：

- Java 外部環境の制限 (380 ページ)

CLR 外部環境

データベース・サーバは、CLR ストアド・プロシージャおよび関数をサポートしています。CLR ストアド・プロシージャまたは関数の動作は、SQL ストアド・プロシージャまたは関数と同じです。ただし、プロシージャまたは関数のコードは C# または Visual Basic などの .NET 言語で記述され、その実行はデータベース・サーバの外側 (つまり別の .NET 実行ファイル内) で行われます。

この .NET 実行ファイルのインスタンスは、データベースごとに 1 つだけです。CLR 関数およびストアド・プロシージャを実行するすべての接続が、同じ .NET 実行インスタンスを使用します。ただし、ネームスペースは接続ごとに異なります。静的変数は接続の間持続しますが、接続間で共有することはできません。.NET バージョン 2.0 のみがサポートされています。

外部 CLR 関数またはプロシージャを呼び出すには、ロードする DLL およびアセンブリ内で呼び出す関数を定義する EXTERNAL NAME 文字列を指定して、対応するストアド・プロシージャまたは関数を定義します。ストアド・プロシージャまたは関数を定義する際は、LANGUAGE CLR も指定する必要があります。次に、宣言の例を示します。

```
CREATE PROCEDURE clr_stored_proc(
    IN p1 INT,
    IN p2 UNSIGNED SMALLINT,
    OUT p3 LONG VARCHAR)
EXTERNAL NAME 'MyCLRTest.dll::MyCLRTest.Run( int, ushort, out
string )'
LANGUAGE CLR;
```

この例では、**clr_stored_proc** というストアド・プロシージャが、その実行時に DLL MyCLRTest.dll をロードし、関数 **MyCLRTest.Run** を呼び出します。**clr_stored_proc** プロシージャは 3 つの SQL パラメータを受け取ります。そのうち 2 つはそれぞれ INT 型と UNSIGNED SMALLINT 型の IN パラメータで、もう 1 つは

LONG VARCHAR 型の OUT パラメータです。.NET 側で、この 3 つのパラメータは int 型と ushort 型の入力引数、および string 型の出力引数に変換されます。out 引数のほかに、CLR 関数では ref 引数も使用できます。対応するストアド・プロシージャに INOUT パラメータがある場合、ユーザは ref CLR 引数を宣言する必要があります。

次の表は、CLR 引数の各種データ型と、それに対応する推奨される SQL データ型の一覧です。

CLR のデータ型	推奨される SQL データ型
bool	bit
byte	tinyint
short	smallint
ushort	unsigned smallint
int	int
uint	unsigned int
long	bigint
ulong	unsigned bigint
decimal	numeric
float	real
double	double
DateTime	timestamp
string	long varchar
byte[]	long binary

DLL の宣言では、相対パスまたは絶対パスのどちらかを指定できます。相対パスが指定された場合、外部 .NET 実行ファイルはそのパスだけでなく、それ以外の場所についても DLL を検索します。ただし、グローバル・アセンブリ・キャッシュ (GAC) では DLL を検索しません。

既存の Java ストアド・プロシージャおよび関数と同様に、CLR ストアド・プロシージャおよび関数もサーバ側の要求をデータベースに戻して、結果セットを返すことができます。また、Java と同じように、Console.Out および Console.Error に出力される情報は、すべてデータベース・サーバ・メッセージ・ウィンドウに自動的にリダイレクトされます。

CLR をデータベースで使用するには、データベース・サーバが CLR 実行ファイルを検出して開始できることを確認してください。データベース・サーバが CLR 実行ファイルを検出して開始できるかどうかを確認するには、次の文を実行します。

```
START EXTERNAL ENVIRONMENT CLR;
```

データベース・サーバが CLR を開始できない場合は、データベース・サーバが CLR 実行ファイルを検出できない可能性があります。CLR 実行ファイルは dbextclr12.exe です。

START EXTERNAL ENVIRONMENT CLR 文は、データベース・サーバが CLR 実行ファイルを開始できるかどうかを確認する以外で使用することはありません。通常、CLR ストアド・プロシージャまたは関数を呼び出すと、CLR は自動的に開始されます。

これと同様に、CLR のインスタンスを停止するために STOP EXTERNAL ENVIRONMENT CLR 文を使用する必要もありません。インスタンスは、接続が切断されると自動的に停止します。ただし、CLR をこれ以上使用することがなく、一部のリソースを解放する必要がある場合は、STOP EXTERNAL ENVIRONMENT CLR 文を使用して接続のための CLR インスタンスを解放します。

Perl、PHP、Java 外部環境とは異なり、CLR 環境ではデータベースに何もインストールする必要がありません。したがって、CLR 外部環境を使用する前に INSTALL 文を実行する必要がありません。

次の例に示す C# で記述された関数は、外部環境で実行できます。

```
public class StaticTest
{
    private static int val = 0;

    public static int GetValue() {
        val += 1;
        return val;
    }
}
```

この関数をダイナミック・リンク・ライブラリにコンパイルすると、外部環境から呼び出すことができます。`dbextclr12.exe` という実行イメージ・ファイルがデータベース・サーバによって開始され、この実行イメージ・ファイルがダイナミック・リンク・ライブラリをロードします。

Microsoft C# コンパイラを使用して、このアプリケーションをダイナミック・リンク・ライブラリに構築するには、次のようなコマンドを使用します。上の例のソース・コードは、`StaticTest.cs` というファイルにあるものと仮定しています。

```
csc /target:library /out:clrtest.dll StaticTest.cs
```

このコマンドは、コンパイル済みのコードを **clrtest.dll** という DLL に置きます。コンパイル済みの C# 関数 GetValue を呼び出すには、Interactive SQL を使用して、ラッパを次のように定義します。

```
CREATE FUNCTION stc_get_value()
RETURNS INT
EXTERNAL NAME 'clrtest.dll::StaticTest.GetValue() int'
LANGUAGE CLR;
```

CLR では、EXTERNAL NAME 文字列は 1 行の SQL 文で指定されます。場合によっては、EXTERNAL NAME 文字列に DLL のパスを含めて、DLL を検出できるようにする必要があります。依存アセンブリの場合 (たとえば、myLib.dll に myOtherLib.dll 内の関数を呼び出すコードがある、または何らかの形で前者が後者に依存する場合) は、.NET Framework によって依存性がロードされます。指定されたアセンブリは CLR 外部環境によってロードされますが、依存アセンブリが確実にロードされるようにするためには追加の手順が必要になる場合があります。解決策としては、.NET Framework にインストールされている Microsoft gacutil ユーティリティを使用してすべての依存性をグローバル・アセンブリ・キャッシュ (GAC) に登録するという方法があります。カスタム開発のライブラリを使用する場合、gacutil を使用して GAC に登録する前に、厳密な名前キーでライブラリが署名してある必要があります。

サンプルのコンパイル済み C# 関数を実行するには、次の文を実行します。

```
SELECT stc_get_value();
```

C# 関数が呼び出されるたびに、整数値の結果が新しく生成されます。返される値は、1、2、3、の順に続きます。

ESQL 外部環境と ODBC 外部環境

コンパイル済みネイティブ C 関数をデータベース・サーバ内でなく外部環境で実行するには、ストアド・プロシージャまたは関数を EXTERNAL NAME 句で定義し、後続の LANGUAGE 属性で C_ESQL32、C_ESQL64、C_ODBC32、C_ODBC64 のいずれか 1 つを指定します。

Perl、PHP、および Java の外部環境とは異なり、データベースにソース・コードやコンパイル済みオブジェクトはインストールしません。したがって、ESQL および ODBC の外部環境を使用する前に INSTALL 文を実行する必要がありません。

次の例に示す C++ で記述された関数は、データベース・サーバ内でも外部環境内でも実行できます。

```
#include <windows.h>
#include <stdio.h>
```

```
#include <stdlib.h>
#include <string.h>
#include "extfnapi.h"

BOOL APIENTRY DllMain( HMODULE hModule,
    DWORD  ul_reason_for_call,
    LPVOID lpReserved
    )
{
    return TRUE;
}

// Note: extfn_use_new_api used only for
// execution in the database server

extern "C" __declspec( dllexport )
a_sql_uint32 extfn_use_new_api( void )
{
    return( EXTFN_API_VERSION );
}

extern "C" __declspec( dllexport )
void SimpleCFunction(
    an_extfn_api *api,
    void *arg_handle )
{
    short               result;
    an_extfn_value      arg;
    an_extfn_value      retval;
    int *               intptr;
    int                 i, j, k;

    j = 1000;
    k = 0;
    for( i = 1; i <= 4; i++ )
    {
        result = api->get_value( arg_handle, i, &arg );
        if( result == 0 || arg.data == NULL ) break;
        if( arg.type & DT_TYPES != DT_INT ) break;
        intptr = (int *) arg.data;
        k += *intptr * j;
        j = j / 10;
    }
    retval.type = DT_INT;
    retval.data = (void*)&k;
    retval.piece_len = retval.len.total_len =
       (a_sql_uint32) sizeof( int );
    api->set_value( arg_handle, 0, &retval, 0 );
    return;
}
```

この関数をダイナミック・リンク・ライブラリまたは共有オブジェクトにコンパイルすると、外部環境から呼び出すことができます。dbexternc12 という実行イメージ・ファイルがデータベース・サーバによって開始され、この実行イメー

ジ・ファイルがダイナミック・リンク・ライブラリまたは共有オブジェクトをロードします。

32 ビットまたは 64 ビット・バージョンのデータベース・サーバを使用でき、どちらのバージョンのデータベース・サーバでも 32 ビットまたは 64 ビット・バージョンの dbexternc12 を開始できます。これは、外部環境を使用するメリットの 1 つです。データベース・サーバによって開始された dbexternc12 は、接続が切断されるか STOP EXTERNAL ENVIRONMENT 文が (正しい環境名で) 実行されるまで終了しません。外部環境呼出しを実行する接続には、dbexternc12 のコピーがそれぞれ与えられます。

コンパイル済みネイティブ関数 SimpleCFunction を呼び出すには、次のようにラッパを定義します。

```
CREATE FUNCTION SimpleCDemo(
  IN arg1 INT,
  IN arg2 INT,
  IN arg3 INT,
  IN arg4 INT )
RETURNS INT
EXTERNAL NAME 'SimpleCFunction@c:¥¥c¥¥extdemo.dll'
LANGUAGE C_ODBC32;
```

これは、コンパイル済みネイティブ関数をデータベース・サーバのアドレス空間にロードする場合の記述方法と、ほとんど同じです。ただ 1 つ異なるのは、LANGUAGE C_ODBC32 句を使用することです。この句は、SimpleCDemo が外部環境で実行される関数であり、32 ビットの ODBC 呼び出しを使用することを指定しています。C_ESQL32、C_ESQL64、C_ODBC32、C_ODBC64 の言語の指定は、サーバ側の要求を作成するときに、外部 C 関数で 32 ビットまたは 64 ビットの ODBC、ESQL、または `a_v4_extfn` API 呼び出しのどれを行うかをデータベース・サーバに知らせます。

サーバ側の要求を作成する際にネイティブ関数が ODBC 呼び出し、ESQL 呼び出し、SQL Anywhere C API 呼び出しのいずれも使用しない場合は、32 ビットのアプリケーションには C_ODBC32 または C_ESQL32 を、64 ビットのアプリケーションには C_ODBC64 または C_ESQL64 を使用できます。上記の外部 C 関数はこれに該当します。この関数はこれらの API を一切使用しません。

サンプルのコンパイル済みネイティブ関数を実行するには、次の文を実行します。

```
SELECT SimpleCDemo(1,2,3,4);
```

サーバ側の ODBC を使用するには、C/C++ コードでデフォルトのデータベース接続を使用する必要があります。データベース接続のハンドルを取得するには、EXTFN_CONNECTION_HANDLE_ARG_NUM 引数を指定して get_value を呼び出します。この引数は、新しい外部環境接続を開くのではなく、現在の外部環境接続を返すようにデータベース・サーバに伝えます。

```
#include <windows.h>
#include <stdio.h>
#include "odbc.h"
#include "extfnapi.h"

BOOL APIENTRY DllMain( HMODULE hModule,
    DWORD  ul_reason_for_call,
    LPVOID lpReserved
    )
{
    return TRUE;
}

extern "C" __declspec( dllexport )
void ServerSideFunction( an_extfn_api *api, void *arg_handle )
{
    short               result;
    an_extfn_value      arg;
    an_extfn_value      retval;
    SQLRETURN           ret;

    ret = -1;
    // set up the return value struct
    retval.type = DT_INT;
    retval.data = (void*) &ret;
    retval.piece_len = retval.len.total_len =
        (a_sql_uint32) sizeof( int );

    result = api->get_value( arg_handle,
                    EXTFN_CONNECTION_HANDLE_ARG_NUM,
                    &arg );
    if( result == 0 || arg.data == NULL )
    {
        api->set_value( arg_handle, 0, &retval, 0 );
        return;
    }

    HDBC dbc = (HDBC)arg.data;
    HSTMT stmt = SQL_NULL_HSTMT;
    ret = SQLAllocHandle( SQL_HANDLE_STMT, dbc, &stmt );
    if( ret != SQL_SUCCESS ) return;
    ret = SQLExecDirect( stmt,
            (SQLCHAR *) "INSERT INTO odbcTab "
                "SELECT table_id, table_name "
                "FROM SYS.SYSTAB", SQL_NTS );
    if( ret == SQL_SUCCESS )
    {
        SQLExecDirect( stmt,
            (SQLCHAR *) "COMMIT", SQL_NTS );
    }
    SQLFreeHandle( SQL_HANDLE_STMT, stmt );

    api->set_value( arg_handle, 0, &retval, 0 );
    return;
}
```

上記の ODBC コードが extodbc.cpp ファイルに保存されている場合、次のコマンドを使用して Windows 用に構築できます。

```
cl extodbc.cpp /LD /Ic:¥sa12¥sdk¥include odbc32.lib
```

次の例では、テーブルを作成し、コンパイル済みネイティブ関数を呼び出すストアド・プロシージャのラッパを定義してから、ネイティブ関数を呼び出してテーブルにデータを移植します。

```
CREATE TABLE odbcTab(c1 int, c2 char(128));

CREATE FUNCTION ServerSideODBC( )
RETURNS INT
EXTERNAL NAME 'ServerSideFunction@extodbc.dll'
LANGUAGE C_ODBC32;

SELECT ServerSideODBC();

// The following statement should return two identical rows
SELECT COUNT(*) FROM odbcTab
UNION ALL
SELECT COUNT(*) FROM SYS.SYSTAB;
```

同様に、サーバ側の ESQL を使用するには、C/C++ コードでデフォルトのデータベース接続を使用する必要があります。データベース接続のハンドルを取得するには、EXTFN_CONNECTION_HANDLE_ARG_NUM 引数を指定して get_value を呼び出します。この引数は、新しい外部環境接続を開くのではなく、現在の外部環境接続を返すようにデータベース・サーバに伝えます。

```
#include <windows.h>
#include <stdio.h>

#include "sqlca.h"
#include "sqlda.h"
#include "extfnapi.h"

BOOL APIENTRY DllMain( HMODULE hModule,
    DWORD  ul_reason_for_call,
    LPVOID lpReserved
    )
{
    return TRUE;
}

EXEC SQL INCLUDE SQLCA;
static SQLCA *_sqlc;
EXEC SQL SET SQLCA "_sqlc";
EXEC SQL WHENEVER SQLERROR { ret = _sqlc->sqlcode; };

extern "C" __declspec( dllexport )
void ServerSideFunction( an_extfn_api *api, void *arg_handle )
{
    short               result;
    an_extfn_value      arg;
```

```
    an_extfn_value      retval;

    EXEC SQL BEGIN DECLARE SECTION;
    char *stmt_text =
        "INSERT INTO esqlTab "
            "SELECT table_id, table_name "
            "FROM SYS.SYSTAB";
    char *stmt_commit =
        "COMMIT";
    EXEC SQL END DECLARE SECTION;

    int ret = -1;

    // set up the return value struct
    retval.type = DT_INT;
    retval.data = (void*) &ret;
    retval.piece_len = retval.len.total_len =
        (a_sql_uint32) sizeof( int );

    result = api->get_value( arg_handle,
                    EXTFN_CONNECTION_HANDLE_ARG_NUM,
                    &arg );
    if( result == 0 || arg.data == NULL )
    {
        api->set_value( arg_handle, 0, &retval, 0 );
        return;
    }
    ret = 0;
    _sqlc = (SQLCA *)arg.data;

    EXEC SQL EXECUTE IMMEDIATE :stmt_text;
    EXEC SQL EXECUTE IMMEDIATE :stmt_commit;

    api->set_value( arg_handle, 0, &retval, 0 );
}
```

上記の Embedded SQL 文が extesql.sqc ファイルに保存されている場合、次のコマンドを使用して Windows 用に構築できます。

```
sqlpp extesql.sqc extesql.cpp
cl extesql.cpp /LD /Ic:¥sa12¥sdk¥include c:¥sa12¥sdk¥lib
¥x86¥dblibtm.lib
```

次の例では、テーブルを作成し、コンパイル済みネイティブ関数を呼び出すストアド・プロシージャのラッパを定義してから、ネイティブ関数を呼び出してテーブルにデータを移植します。

```
CREATE TABLE esqlTab(c1 int, c2 char(128));

CREATE FUNCTION ServerSideESQL( )
RETURNS INT
EXTERNAL NAME 'ServerSideFunction@extesql.dll'
LANGUAGE C_ESQL32;

SELECT ServerSideESQL();
```

```
// The following statement should return two identical rows
SELECT COUNT(*) FROM esqlTab
UNION ALL
SELECT COUNT(*) FROM SYS.SYSTAB;
```

前述の例のように、サーバ側の SQL Anywhere C API 呼び出しを使用するには、C/C++ コードでデフォルトのデータベース接続を使用する必要があります。データベース接続のハンドルを取得するには、EXTFN_CONNECTION_HANDLE_ARG_NUM 引数を指定して get_value を呼び出します。この引数は、新しい外部環境接続を開くのではなく、現在の外部環境接続を返すようにデータベース・サーバに伝えます。次の例は、接続ハンドルを取得し、C API 環境を初期化し、接続ハンドルを SQL Anywhere C API で使用できる接続オブジェクト (a_sqlany_connection) に変換するためのフレームワークを示したものです。

```
include <windows.h>
#include "sacapidll.h"
#include "extfnapi.h"

BOOL APIENTRY DllMain( HMODULE hModule,
    DWORD  ul_reason_for_call,
    LPVOID lpReserved
    )
{
    return TRUE;
}

extern "C" __declspec( dllexport )
void ServerSideFunction( an_extfn_api *extapi, void *arg_handle )
{
    short               result;
    an_extfn_value      arg;
    an_extfn_value      retval;
    unsigned            offset;
    char                *cmd;

    SQLAnywhereInterface  capi;
    a_sqlany_connection * sqlany_conn;
    unsigned int          max_api_ver;

    result = extapi->get_value( arg_handle,
                    EXTFN_CONNECTION_HANDLE_ARG_NUM,
                    &arg );

    if( result == 0 || arg.data == NULL )
    {
        return;
    }
    if( !sqlany_initialize_interface( &capi, NULL ) )
    {
        return;
    }
```

```
    if( !capi.sqlany_init( "MyApp",
            SQLANY_CURRENT_API_VERSION,
            &max_api_ver ) )
    {
        sqlany_finalize_interface( &capi );
        return;
    }
    sqlany_conn = sqlany_make_connection( arg.data );

    // processing code goes here

    capi.sqlany_fini();

    sqlany_finalize_interface( &capi );
    return;
}
```

上記の C コードが extcapi.c ファイルに保存されている場合、次のコマンドを使用して Windows 用に構築できます。

```
cl /LD /Tp extcapi.c /Tp c:¥sa12¥SDK¥C¥sacapidll.c
      /Ic:¥sa12¥SDK¥Include c:¥sa12¥SDK¥Lib¥X86¥dbcapi.lib
```

次の例では、コンパイル済みネイティブ関数を呼び出すストアド・プロシージャのラッパを定義してから、ネイティブ関数を呼び出します。

```
CREATE FUNCTION ServerSideC()
RETURNS INT
EXTERNAL NAME 'ServerSideFunction@extcapi.dll'
LANGUAGE C_ESQL32;

SELECT ServerSideC();
```

上記の例の LANGUAGE 属性では C_ESQL32 を指定しています。64 ビットのアプリケーションの場合は C_ESQL64 を使用します。SQL Anywhere C API は ESQL と同じレイヤ (ライブラリ) に構築されているため、Embedded SQL の LANGUAGE 属性を使用する必要があります。

前述のとおり、外部環境呼び出しを実行する接続は、dbexternc12 のコピーをそれぞれ開始します。この実行可能アプリケーションは、最初の外部環境呼び出しが実行される際にサーバによって自動的にロードされます。ただし、START EXTERNAL ENVIRONMENT 文を使用して dbexternc12 をプリロードすることもできます。外部環境呼び出しを初めて実行する際のわずかな遅延を回避したい場合には便利です。次に、この文の例を示します。

```
START EXTERNAL ENVIRONMENT C_ESQL32
```

dbexternc12 のプリロードは、外部関数をデバッグする場合も便利です。デバッガを使用して実行中の dbexternc12 プロセスにアタッチし、外部関数にブレークポイントを設定できます。

STOP EXTERNAL ENVIRONMENT 文は、ダイナミック・リンク・ライブラリや共有オブジェクトを更新する場合に便利です。現在の接続でネイティブ・ライブラリ・ローダの dbexternc12 を終了し、ダイナミック・リンク・ライブラリや共有オブジェクトへのアクセスを解放します。複数の接続が同じダイナミック・リンク・ライブラリまたは共有オブジェクトを使用している場合は、dbexternc12 の各コピーを終了する必要があります。STOP EXTERNAL ENVIRONMENT 文には、適切な外部環境名を指定する必要があります。次に、この文の例を示します。

```
STOP EXTERNAL ENVIRONMENT C_ESQL32
```

外部関数から結果セットを返すためには、コンパイル済みネイティブ関数はネイティブ関数呼び出しインタフェースを使用する必要があります。

次のコード・フラグメントは、結果セット情報の構造体を設定する方法を示したものです。カラム・カウント、カラム情報の構造体の配列へのポインタ、カラム・データ値の構造体の配列へのポインタを含んでいます。この例は、SQL Anywhere C API も使用しています。

```
an_extfn_result_set_info    rs_info;

int columns = capi.sqlany_num_cols( sqlany_stmt );

an_extfn_result_set_column_info *col_info =
    (an_extfn_result_set_column_info *)
    malloc( columns * sizeof(an_extfn_result_set_column_info) );

an_extfn_result_set_column_data *col_data =
    (an_extfn_result_set_column_data *)
    malloc( columns * sizeof(an_extfn_result_set_column_data) );

rs_info.number_of_columns   = columns;
rs_info.column_infos        = col_info;
rs_info.column_data_values  = col_data;
```

次のコード・フラグメントは、結果セットを記述する方法を示したものです。SQL Anywhere C API を使用して、C API によって実行された SQL クエリのカラム情報を取得します。SQL Anywhere C API から取得した各カラムの情報は、カラムの名前、型、幅、インデックス、および NULL 値インジケータに変換され、結果セットの記述に使用されます。

```
a_sqlany_column_info        info;
for( int i = 0; i < columns; i++ )
{
    if( sqlany_get_column_info( sqlany_stmt, i, &info ) )
    {
        // set up a column  description
        col_info[i].column_name  = info.name;
        col_info[i].column_type  = info.native_type;
        switch( info.native_type )
        {
           case DT_DATE:      // DATE is converted to string by C API
           case DT_TIME:      // TIME is converted to string by C API
```

```
          case DT_TIMESTAMP:  // TIMESTAMP is converted to string by
C API
          case DT_DECIMAL:    // DECIMAL is converted to string by C
API
               col_info[i].column_type  = DT_FIXCHAR;
               break;
          case DT_FLOAT:     // FLOAT is converted to double by C API
               col_info[i].column_type  = DT_DOUBLE;
               break;
          case DT_BIT:       // BIT is converted to tinyint by C API
               col_info[i].column_type  = DT_TINYINT;
               break;
        }
        col_info[i].column_width = info.max_size;
       col_info[i].column_index = i + 1; // column indices are origin
1
        col_info[i].column_can_be_null = info.nullable;
    }
}
// send the result set description
if( extapi->set_value( arg_handle,
                    EXTFN_RESULT_SET_ARG_NUM,
                    (an_extfn_value *)&rs_info,
                    EXTFN_RESULT_SET_DESCRIBE ) == 0 )
{
    // failed
    free( col_info );
    free( col_data );
    return;
}
```

結果セットが記述されると、結果セットのローを返すことができます。次のコード・フラグメントは、結果セットのローを返す方法を示したものです。SQL Anywhere C API を使用して、C API によって実行された SQL クエリのローをフェッチします。SQL Anywhere C API によって返されたローは、呼び出しを行った環境に 1 つずつ送り返されます。カラム・データ値の構造体の配列に格納してから、各ローを返す必要があります。カラム・データ値の構造体はカラム・インデックス、データ値へのポインタ、データ長、追加フラグから構成されます。

```
a_sqlany_data_value *value = (a_sqlany_data_value *)
    malloc( columns * sizeof(a_sqlany_data_value) );

while( capi.sqlany_fetch_next( sqlany_stmt ) )
{
    for( int i = 0; i < columns; i++ )
    {
        if( capi.sqlany_get_column( sqlany_stmt, i, &value[i] ) )
        {
            col_data[i].column_index = i + 1;
            col_data[i].column_data  = value[i].buffer;
            col_data[i].data_length  =
(a_sql_uint32)*(value[i].length);
            col_data[i].append  = 0;
            if( *(value[i].is_null) )
```

```
            {
                // Received a NULL value
                col_data[i].column_data = NULL;
            }
        }
    }
    if( extapi->set_value(  arg_handle,
                        EXTFN_RESULT_SET_ARG_NUM,
                        (an_extfn_value *)&rs_info,
                        EXTFN_RESULT_SET_NEW_ROW_FLUSH ) == 0 )
    {
        // failed
        free( value );
        free( col_data );
        free( col_data );
        extapi->set_value( arg_handle, 0, &retval, 0 );
        return;
    }
}
```

Java 外部環境

データベース・サーバは、Java ストアド・プロシージャおよび関数をサポートしています。Java ストアド・プロシージャまたは関数の動作は、SQL ストアド・プロシージャまたは関数と同じです。ただし、プロシージャまたは関数のコードは Java で記述され、その実行はデータベース・サーバの外側 (つまり Java VM 環境内) で行われます。

接続ごとに 1 つのインスタンスではなく、各データベースの Java VM に 1 つのインスタンスがあります。Java ストアド・プロシージャは結果セットを返すことができます。

データベースで Java のサポートを使用するためには、いくつかの前提条件があります。

1. Java Runtime Environment のコピーをデータベース・サーバ・コンピュータにインストールする必要があります。
2. データベース・サーバが Java 実行ファイル (Java VM) を検出できる必要があります。

Java をデータベースで使用するには、データベース・サーバが Java 実行ファイルを検出して開始できることを確認してください。これは、次の文を実行して確認できます。

```
START EXTERNAL ENVIRONMENT JAVA;
```

データベース・サーバが Java を開始できない場合は、データベース・サーバが Java 実行ファイルを検出できないことが問題の原因であると考えられます。この場合は、ALTER EXTERNAL ENVIRONMENT 文を実行して、Java 実行ファイルの

ロケーションを明示的に設定してください。実行ファイル名を必ず含めてください。

```
ALTER EXTERNAL ENVIRONMENT JAVA
  LOCATION 'java-path';
```

次に例を示します。

```
ALTER EXTERNAL ENVIRONMENT JAVA
  LOCATION 'c:¥¥jdk1.6.0¥¥jre¥¥bin¥¥java.exe';
```

データベース・サーバが使用する Java VM の場所は、次の SQL クエリで確認できます。

```
SELECT db_property('JAVAVM');
```

START EXTERNAL ENVIRONMENT JAVA 文は、データベース・サーバが Java VM を開始できるかどうかを確認する以外で使用することはありません。通常、Java ストアド・プロシージャまたは関数を呼び出すと、Java VM は自動的に開始されます。

これと同様に、Java のインスタンスを停止するために **STOP EXTERNAL ENVIRONMENT JAVA** 文を使用する必要もありません。インスタンスは、データベースへの接続がすべて切断されると自動的に停止します。ただし、Java をこれ以上使用することがなく、一部のリソースを解放する必要がある場合は、**STOP EXTERNAL ENVIRONMENT JAVA** 文によって Java VM の使用カウントを減分できます。

データベース・サーバが Java VM 実行ファイルを開始できることを確認したら、次に必要な Java クラス・コードをデータベースにインストールします。これは、**INSTALL JAVA** 文を使用して行います。たとえば、次の文を実行して Java クラスをファイルからデータベースにインストールできます。

```
INSTALL JAVA
NEW
FROM FILE 'java-class-file';
```

データベースには、Java JAR ファイルもインストールできます。

```
INSTALL JAVA
NEW
JAR 'jar-name'
FROM FILE 'jar-file';
```

Java クラスは、次のようにして変数からインストールできます。

```
CREATE VARIABLE JavaClass LONG VARCHAR;
SET JavaClass = xp_read_file('java-class-file')
INSTALL JAVA
NEW
```

```
FROM JavaClass;
```

Java クラスをデータベースから削除するには、次のように **REMOVE JAVA** 文を使用します。

```
REMOVE JAVA CLASS java-class
```

Java JAR をデータベースから削除するには、次のように **REMOVE JAVA** 文を使用します。

```
REMOVE JAVA JAR 'jar-name'
```

既存の Java クラスを変更するには、次のように **INSTALL JAVA** 文の **UPDATE** 句を使用します。

```
INSTALL JAVA
UPDATE
FROM FILE 'java-class-file'
```

データベース内の既存の Java JAR ファイルを更新することもできます。

```
INSTALL JAVA
UPDATE
JAR 'jar-name'
FROM FILE 'jar-file';
```

Java クラスは、次のようにして変数から更新できます。

```
CREATE VARIABLE JavaClass LONG VARCHAR;
SET JavaClass = xp_read_file('java-class-file')
INSTALL JAVA
UPDATE
FROM JavaClass;
```

Java クラスをデータベースにインストールしたら、次に Java メソッドへのインタフェースとなるストアド・プロシージャおよび関数を作成できます。**EXTERNAL NAME** 文字列には、Java メソッドを呼び出し、OUT パラメータおよび戻り値を返すために必要な情報が含まれています。**EXTERNAL NAME** 句の LANGUAGE 属性には **JAVA** を指定する必要があります。**EXTERNAL NAME** 句のフォーマットは次のとおりです。

EXTERNAL NAME '*java-call*' **LANGUAGE JAVA**

java-call：

```
[package-name.]class-name.method-name method-signature
```

method-signature：

```
( [ field-descriptor, ... ] ) return-descriptor
```

field-descriptor と return-descriptor：

- Z
- | B
- | S
- | I
- | J
- | F
- | D
- | C
- | V
- | [descriptor
- | Lclass-name;

Java メソッド・シグニチャは、パラメータの型と戻り値の型を簡潔に文字で表現したものです。パラメータの数がメソッド・シグニチャに指定された数字よりも小さい場合は、この差が **DYNAMIC RESULT SETS** に指定された数と等しくなるようにします。また、プロシージャ・パラメータ・リストよりも多いメソッド・シグニチャ内の各パラメータには、メソッド・シグニチャ **[Ljava/SQL/ResultSet;** が必要です。

Java UDF については、**DYNAMIC RESULT SETS** を設定する必要はありません。**DYNAMIC RESULT SETS** は 1 と暗黙的にみなされます。

field-descriptor と *return-descriptor* の意味は次のとおりです。

フィールド・タイプ	Java データ型
B	byte
C	char
D	double
F	float
I	int
J	long
L class-name;	クラス class-name のインスタンス。クラス名は、完全に修飾された名前で、ドットを / に置き換えたものとします。例：java/lang/String
S	short
V	void
Z	Boolean
[	配列の次元ごとに 1 つ使用

次に例を示します。

```
double some_method(
  boolean a,
  int b,
  java.math.BigDecimal c,
  byte [][] d,
  java.sql.ResultSet[] rs ) {
}
```

この例では、次のシグニチャを得られます。

```
'(ZILjava/math/BigDecimal;[[B[Ljava/SQL/ResultSet;)D'
```

次のプロシージャは、Java メソッドへのインタフェースを作成するものです。この Java メソッドはいかなる値も返しません (V)。

```
CREATE PROCEDURE insertfix()
EXTERNAL NAME 'JDBCExample.InsertFixed()V'
LANGUAGE JAVA;
```

次のプロシージャは、String ([Ljava/lang/String;) 入力引数を持つ Java メソッドへのインタフェースを作成するものです。この Java メソッドはいかなる値も返しません (V)。

```
CREATE PROCEDURE InvoiceMain( IN arg1 CHAR(50) )
EXTERNAL NAME 'Invoice.main([Ljava/lang/String;)V'
LANGUAGE JAVA;
```

次のプロシージャは、Java メソッド `Invoice.init` へのインタフェースを作成するものです。この Java メソッドは、文字列引数 (Ljava/lang/String;)、double (D)、別の文字列引数 (Ljava/lang/String;)、別の double (D) を受け取り、値を返しません (V)。

```
 CREATE PROCEDURE init( IN arg1 CHAR(50),
                        IN arg2 DOUBLE,
                        IN arg3 CHAR(50),
                        IN arg4 DOUBLE)
EXTERNAL NAME 'Invoice.init(Ljava/lang/String;DLjava/lang/
String;D)V'
LANGUAGE JAVA
```

次に示す Java の例には、文字列引数を受け取ってデータベース・サーバ・メッセージ・ウィンドウに書き込む関数 main があります。また、Java の String を返す関数 **whare** もあります。

```
import java.io.*;

public class Hello
{
    public static void main( String[] args )
    {
        System.out.print( "Hello" );
               for ( int i = 0; i  < args.length; i++ )
                   System.out.print( " " + args[i] );
```

```
        System.out.println();
    }
    public static String whare()
    {
        return( "I am SQL Anywhere." );
    }
}
```

上記の Java コードは、`Hello.java` ファイルにあり、Java コンパイラを使用してコンパイルします。生成されるクラス・ファイルは次のようにデータベースにロードされます。

```
INSTALL JAVA
NEW
FROM FILE 'Hello.class';
```

`Hello` クラスの `main` メソッドへのインタフェースとなるストアド・プロシージャは、Interactive SQL を使用して次のように作成されます。

```
CREATE PROCEDURE HelloDemo( IN name LONG VARCHAR )
EXTERNAL NAME 'Hello.main([Ljava/lang/String;)V'
LANGUAGE JAVA;
```

main の引数は `java.lang.String` の配列として記述されています。Interactive SQL で次の SQL 文を実行することでインタフェースをテストします。

```
CALL HelloDemo('SQL Anywhere');
```

メッセージは、データベース・サーバ・メッセージ・ウィンドウに表示されます。`System.out` への出力はすべてサーバ・メッセージ・ウィンドウにリダイレクトされます。

`Hello` クラスの `whare` メソッドへのインタフェースとなるストアド・プロシージャは、Interactive SQL を使用して次のように作成されます。

```
CREATE FUNCTION Whare()
RETURNS LONG VARCHAR
EXTERNAL NAME 'Hello.whoAreYou(V)Ljava/lang/String;'
LANGUAGE JAVA;
```

関数 `whare` は `java.lang.String` を返すものとして記述されています。Interactive SQL で次の SQL 文を実行することでインタフェースをテストします。

```
SELECT Whare();
```

Interactive SQL の結果ウィンドウに応答が表示されます。

Java 外部環境が起動しないとき (Java の呼び出し時にアプリケーションで `"main thread not found"` エラーが発生するとき) のトラブルシューティングでは、DBA は次の点を確認します。

- Java VM のビット数がデータベース・サーバと異なる場合には、VM と同じビット数のクライアント・ライブラリがデータベース・サーバ・マシンにインストールされていることを確認する。
- 共有オブジェクト `sajdbc.jar` と `dbjdbc12/libdbjdbc12` のソフトウェア・ビルドが同じであることを確認する。
- データベース・サーバ・マシン上に複数の `sajdbc.jar` が存在する場合には、すべて同じソフトウェア・バージョンに同期されていることを確認する。
- データベース・サーバ・マシンの負荷が非常に高い場合には、タイムアウトによりエラーとなっている可能性がある。

参照：

- INSTALL JAVA 文 (389 ページ)
- CREATE PROCEDURE 文 (Java UDF) (392 ページ)
- CREATE FUNCTION 文 (Java UDF) (394 ページ)
- REMOVE 文 (396 ページ)
- START JAVA 文 (397 ページ)
- STOP JAVA 文 (397 ページ)

マルチプレックスでの Java 外部環境

Java 外部環境の UDF をマルチプレックス設定で使用する前に、その UDF を必要とするマルチプレックスの各ノードに Java クラス・ファイルまたは JAR ファイルをインストールします。

Sybase Control Center、Sybase Central、または Interactive SQL の **INSTALL JAVA** 文を使用して、Java クラス・ファイルと JAR をインストールします。

参照：

- INSTALL JAVA 文 (389 ページ)

Sybase Central によるクラスのインストール

Java クラスをデータベースで使用できるようにするには、Sybase Central を使用してそのクラスをデータベースにインストールします。インストールするクラスのパスとファイル名を確認します。

クラスをインストールするには、DBA 権限が必要です。

クラスをインストールするには、次の手順に従います (Sybase Central の場合)。

1. DBA ユーザとしてデータベースに接続します。
2. [外部環境] フォルダを開きます。

3. このフォルダの中にある [Java] フォルダを開きます。
4. 右ウィンドウ枠を右クリックし、[新規] - [Java クラス] を選択します。
5. ウィザードの指示に従います。

Interactive SQL によるクラスのインストール

Java クラスをデータベースで使用できるようにするには、Interactive SQL で INSTALL JAVA 文を使用してそのクラスをデータベースにインストールします。インストールするクラスのパスとファイル名を確認します。

1. DBA ユーザとしてデータベースに接続します。
2. 次の文を実行します。

```
INSTALL JAVA NEW
FROM FILE 'path¥¥ClassName.class';
```

`path` はクラス・ファイルが保持されるディレクトリ、`ClassName.class` はクラス・ファイル名を表します。

二重円記号は、円記号がエスケープ文字として処理されるのを防ぐために使用します。

たとえば、`c:¥source` ディレクトリに保持されている `Utility.class` ファイルにクラスをインストールするには、次の文を実行します。

```
INSTALL JAVA NEW
FROM FILE 'c:¥¥source¥¥Utility.class';
```

相対パスを使用する場合は、データベース・サーバの現在の作業ディレクトリからの相対パスとします。

Java 外部環境の制限

Java UDF と Java テーブル UDF を開発する前に、UDF 向けの Java 外部環境に固有の制限について理解してください。

- 集合 Java 関数はサポートされていない。
- Java UDF が関係するクエリ・フラグメントは DQP または SMP 処理には適さない。
- 現在のクエリに関係しているテーブルを Java 外部環境内から **DROP** することはできない。
- 現在のクエリに関係しているテーブルを Java 外部環境内から **ALTER** することはできない。
- `UNSIGNED SMALLINT` データ型はサポートされていない。
- 数値関数の精度は 255 以下に制限される。
- Java テーブル UDF では単一の結果セットのみを使用できる。

参照：
- 外部環境の制限 (360 ページ)

Java VM のメモリ・オプション

Java 仮想マシン (VM) の起動に必要な追加コマンド・ライン・オプションを指定するには、**java_vm_options** オプションを使用します。

次の構文を使用します。

```
SET OPTION PUBLIC.java_vm_options='java-options';
```

次の例では、**java_vm_options** を使用して、Java VM の最大ヒープ・サイズを 512 メガバイトに設定します。

```
SET OPTION PUBLIC.java_vm_options='-Xmx512m';
```

次の例では、Java VM の初期ヒープ・サイズを 32 メガバイトに設定します。

```
SET OPTION PUBLIC.java_vm_options='-Xms32m';
```

Java UDF 向けの SQL データ型変換

SQL から Java と Java から SQL へのデータ型変換は、JDBC 標準に準拠して行われます。LOB データ型 `LONG VARCHAR` と `LONG BINARY` は、入力値ではサポートされていますが、戻り値ではサポートされていません。

SQL から Java のデータ型変換

Java スカラ UDF と Java テーブル UDF の入力値で使用されるデータ型変換です。

SQL 型	Java 型
BIGINT	long
BINARY	byte[ ]
BIT	boolean
CHAR	String
DATE	java.sql.Date
DECIMAL	java.math.BigDecimal
DOUBLE	double
IMAGE	byte[ ]
INTEGER	int

SQL 型	Java 型
LONG BINARY	byte[ ] **注意：** ラージ・オブジェクト・データのサポートには、別途ライセンスが必要な Sybase IQ オプションが必要です。
LONG VARCHAR	String **注意：** ラージ・オブジェクト・データのサポートには、別途ライセンスが必要な Sybase IQ オプションが必要です。
MONEY	java.math.BigDecimal
NUMERIC	java.math.BigDecimal
REAL	float
SMALLINT	short
SMALLMONEY	java.math.BigDecimal
TEXT	String
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp
TINYINT	byte
UNSIGNED BIGINT	java.math.BigDecimal (精度 20、位取り 0)
UNSIGNED INT	java long
VARBINARY	byte[ ]
VARCHAR	String

Java から SQL のデータ型変換

Java スカラ UDF と Java テーブル UDF の戻り値のデータ型です。

Java 型	SQL 型
String	CHAR
String	VARCHAR
String	TEXT

Java 型	SQL 型
java.math.BigDecimal	NUMERIC
java.math.BigDecimal	MONEY
java.math.BigDecimal	SMALLMONEY
boolean	BIT
byte	TINYINT
short	SMALLINT
int	INTEGER
long	BIGINT
float	REAL
double	DOUBLE
byte[ ]	VARBINARY
byte[ ]	IMAGE
java.sql.Date	DATE
java.sql.Time	TIME
java.sql.Timestamp	DATETIME/TIMESTAMP
java.lang.Double	DOUBLE
java.lang.Float	REAL
java.lang.Integer	INTEGER
java.lang.Long	BIGINT

Java スカラ UDF の作成

Java クラスを作成およびコンパイルし、クラス・ファイルをサーバにインストールし、関数定義を作成します。

前提条件

- Java の知識があり、.java ファイルをコンパイルできること。生成された .class ファイルがファイル・システムのどこに置かれるかを理解していること。

- Interactive SQL の知識があること。Interactive SQL から `iqdemo` データベースに接続でき、Interactive SQL から **START EXTERNAL ENVIRONMENT JAVA** コマンドを発行できること。

手順

このタスクをテンプレートとして、独自の Java UDF を作成します。

1. 次の Java コードを `HelloJavaUDF.java` というファイルに記述します。

```
public class HelloJavaUDF
{
    public static String helloJava( String name )
    {
        // Simply return Hello and the name passed in.
        return "Hello " + name;
    }
}
```

   このコードでは Java クラス `HelloJavaUDF` が作成されます。このクラスは静的メソッド `helloJava` を持ちます。このメソッドは単一の文字列引数を受け取り、文字列を返します。

2. `HellowJavaUDF.java` をコンパイルします。
   `javac <pathtojavafile>/HelloJavaUDF.java`
3. Interactive SQL で `iqdemo` データベースに接続します。
4. Interactive SQL でクラス・ファイルをサーバにインストールします。

絶対パスを使用	`INSTALL JAVA NEW FROM FILE '<absolutepathtofile>/HelloJavaUDF.class'`
	例：
	`INSTALL JAVA NEW FROM FILE 'd:/mydirectory/HelloJavaUDF.class'`

相対パスを使用	`INSTALL JAVA NEW FROM FILE '<pathrelativetocwd>/HelloJavaUDF.class'`
	例：
	`INSTALL JAVA NEW FROM FILE 'myreldir/HelloJavaUDF.class'`

5. Interactive SQL で関数定義を作成します。
   次の情報を指定します。
   - Java パッケージ名、クラス名、メソッド名
   - 関数の引数の Java データ型と、対応する SQL データ型

- Java UDF への SQL 名の割り当て

```
CREATE FUNCTION my_helloJava(IN name VARCHAR(249) )
RETURNS VARCHAR(255)
EXTERNAL NAME 'example.HelloJavaUDF.helloJava(Ljava/lang/
String;)Ljava/lang/String;'
LANGUAGE JAVA
```

6. Interactive SQL で、iqdemo データベースに対するクエリで Java UDF を使用します。
```
SELECT my_helloJava( GivenName ) FROM Customers WHERE ID <
110
```

参照：

- SQL から Java のデータ型変換 (381 ページ)
- Java から SQL のデータ型変換 (382 ページ)

SQL の substr 関数の Java スカラ UDF バージョンの作成

SQL 関数から複数の引数を受け取る Java UDF を作成します。

前提条件

- Java の知識があり、.java ファイルをコンパイルできること。生成された .class ファイルがファイル・システムのどこに置かれるかを理解していること。
- Interactive SQL の知識があること。Interactive SQL から iqdemo データベースに接続でき、Interactive SQL から **START EXTERNAL ENVIRONMENT JAVA** コマンドを発行できること。

手順

1. 次の Java コードを MyJavaSubstr というファイルに記述します。
```
public class MyJavaSubstr
{
   public static String my_java_substr( String in, int start, int
length )
    {
    String rc = null;

    if ( start < 1 )
    {
        start = 1;
    }

    // Convert the SQL start, length to Java start, end.
    start --;  // Java is 0 based, but SQL is one based.
    int endindex = start+length;
```

```
    try {
        if ( in != null )
        {
            rc = in.substring( start, endindex );
        }
    } catch ( IndexOutOfBoundsException ex )
    {
        System.out.println("ScalarTestFunctions:
                my_java_substr("+in+","+start+","+length+")
failed");
        System.out.println(ex);
    }
    return rc;
 }
```

2. Interactive SQL で `iqdemo` データベースに接続します。
3. Interactive SQL でクラス・ファイルをサーバにインストールします。
   `INSTALL JAVA NEW FROM FILE '<pathtofile>/MyJavaSubstr.class'`
4. Interactive SQL で関数定義を作成します。

```
CREATE or REPLACE FUNCTION java_substr(IN a VARCHAR(255), IN b
INT, IN c INT )
  RETURNS VARCHAR(255)
  EXTERNAL NAME
    'example.MyJavaSubstr.my_java_substr(Ljava/lang/
String;II)Ljava/lang/String;'
  LANGUAGE JAVA
```

   このコードの **Ljava/lang/String;II** という部分は、`String, int, int` のパラメータ型を表しています。
5. Interactive SQL で、`iqdemo` データベースに対するクエリで Java UDF を使用します。
   select GivenName, java_substr(Surname,1,1) from Customers where lcase(java_substr(Surname,1,1)) = 'a';

Java テーブル UDF の作成

Java のロー・ジェネレータを作成、コンパイル、インストールし、Java テーブル UDF の関数定義を作成します。

前提条件

- Java の知識があり、`.java` ファイルをコンパイルできること。生成された `.class` ファイルがファイル・システムのどこに置かれるかを理解していること。

- Interactive SQL の知識があること。Interactive SQL から `iqdemo` データベースに接続でき、Interactive SQL から **START EXTERNAL ENVIRONMENT JAVA** コマンドを発行できること。

手順

この例は、Java のロー・ジェネレータ (RowGenerator) を実行します。このロー・ジェネレータは、入力値として単一の整数を受け取り、その数のローを結果セットで返します。結果セットには 2 つのカラムがあります。1 つは `INTEGER`、もう 1 つは `VARCHAR` です。RowGenerator では次の 2 つのユーティリティ・クラスを使用しています。

- `example.ResultSetImpl`
- `example.ResultSetMetaDataImpl`

これらは `java.sql.ResultSet` インタフェースと `java.sql.ResultSetMetaData` インタフェースのシンプルな実装です。

1. 次のコードを `RowGenerator.java` というファイルに記述します。

```
package example;

import java.sql.*;

public class RowGenerator {

  public static void rowGenerator( int numRows, ResultSet rset[] ) {
    // Create the meta data needed for the result set
    ResultSetMetaDataImpl rsmd = new ResultSetMetaDataImpl(2);

    //The first column is the SQL type INTEGER.
    rsmd.setColumnType(1, Types.INTEGER);
    rsmd.setColumnName(1,"c1");
    rsmd.setColumnLabel(1,"c1");
    rsmd.setTableName(1,"MyTable");

    // The second column is the SQL type VARCHAR length 255
    rsmd.setColumnType(2, Types.VARCHAR);
    rsmd.setColumnName(2,"c2");
    rsmd.setColumnLabel(2,"c2");
    rsmd.setColumnDisplaySize(2, 255);
    rsmd.setTableName(2,"MyTable");

    // Create result set using the ResultSetMetaData
    ResultSetImpl rs = null;
    try {
      rs = new ResultSetImpl( (ResultSetMetaData)rsmd );
      rs.beforeFirst();  // Make sure we are at the beginning.
    } catch( Exception e ) {
      System.out.println( "Error: couldn't create result set." );
      System.out.println( e.toString() );
    }
```

```
    // Add the rows to the result set and populate them
    for( int i = 0; i < numRows; i++ ) {
      try {
        rs.insertRow();   // insert a new row.
       rs.updateInt( 1, i );   // put the integer value in the first column
       rs.updateString( 2, ("Str" + i) );  // put the VARCHAR/String value
in the second column
      } catch( Exception e ) {
        System.out.println( "Error: couldn't insert row/data on row " +
i );
        System.out.println( e.toString() );
      }
    }

    try {
       rs.beforeFirst();  // rewind the result set so that the server gets
it from the beginning.
    } catch( Exception e ) {
        System.out.println( e.toString() );
    }
    rset[0] = rs;  // assign the result set to the 1st of the passed in
array.
  }
}
```

2. RowGenerator.java、ResultSetImpl.java、ResultSetMetaData.java をコンパイルします。Windows ではディレクトリ%ALLUSERSPROFILE%¥samples¥java に (UNIX では $IQDIR15/samples/java に) ResultSetImpl.java と ResultSetMetaData.java があります。

   ```
   javac <pathtojavafile>/ResultSetMetaDataImpl.java
   ```

   ```
   javac <pathtojavafile>/ResultSetImpl.java
   ```

   ```
   javac <pathtojavafile>/RowGenerator.java
   ```

3. Interactive SQL で iqdemo データベースに接続します。
4. Interactive SQL で 3 つのクラス・ファイルをインストールします。

   ```
   INSTALL JAVA NEW FROM FILE '<pathtofile>/
   ResultSetMataDataImpl.class'
   ```

   ```
   INSTALL JAVA NEW FROM FILE '<pathtofile>/
   ResultSetImpl.class'
   ```

   ```
   INSTALL JAVA NEW FROM FILE '<pathtofile>/
   RowGenerator.class'
   ```

5. Interactive SQL で Java テーブル関数の定義を作成します。
   次の情報を指定します。
   - Java パッケージ名、クラス名、メソッド名

- 関数の引数の Java データ型と、対応する SQL データ型
- Java UDF への SQL 名の割り当て

```
CREATE or REPLACE PROCEDURE rowgenerator( IN numRows INTEGER )
    RESULT ( c1 INTEGER , c2 VARCHAR(255) )
    EXTERNAL NAME
        'example.RowGenerator.rowGenerator(I[Ljava/sql/
ResultSet;)V'
    LANGUAGE JAVA
```

注意： RESULT の結果セットには 2 つのカラムを指定しています。1 つは `INTEGER`、もう 1 つは `VARCHAR(255)` です。Java プロトタイプには 2 つの引数を指定しています。1 つは INT (I)、もう 1 つは `java.sql.ResultSets ([Ljava/sql/ResultSet;)` の配列です。この Java プロトタイプは、関数の戻り値が `Void(V)` であることも示しています。

6. Interactive SQL で、`iqdemo` データベースに対するクエリで Java テーブル UDF を使用します。

   ```
   SELECT * from rowGenerator(5);
   ```

   クエリは、2 つのカラムから成る 5 つのローを返します。

参照：
- SQL から Java のデータ型変換 (381 ページ)
- Java から SQL のデータ型変換 (382 ページ)

Java 外部環境 SQL 文リファレンス

Java ストアド・プロシージャと関数の開発では、以下の SQL 文を使用します。

INSTALL JAVA 文

データベース内で Java クラスを使用できるようにします。

構文

INSTALL JAVA [*install-mode*] [**JAR** *jar-name*] **FROM** *source*

パラメータ

- **install-mode：** – { **NEW** | **UPDATE** }
- **source：** – { **FILE***filename* | **URL***url-value* }

例

- **例 1 –** 次の文は、クラスのファイル名とロケーションを指定して、ユーザの作成した "Demo" という名前の Java クラスをインストールします。

```
INSTALL JAVA NEW
FROM FILE 'D:¥JavaClass¥Demo.class'
```

インストール後、クラスはその名前で参照されます。その元のファイル・パスの位置は使用されなくなります。たとえば、次の文は前の文でインストールしたクラスを使用します。

```
CREATE VARIABLE d Demo
```

Demo クラスがパッケージ sybase.work のメンバの場合は、次の例のように、完全に修飾されたクラス名を使用してください。

```
CREATE VARIABLE d sybase.work.Demo
```

- **例 2 –** 次の文は、zip ファイルに含まれるすべてのクラスをインストールし、それらを JAR ファイル名でデータベース内に関連付けます。

```
INSTALL JAVA
JAR 'Widgets'
FROM FILE 'C:¥Jars¥Widget.zip'
```

zip ファイルのロケーションは保持されません。クラスは完全に修飾されたクラス名 (パッケージ名とクラス名) を参照しなければなりません。

使用法

- **インストール・モード (install-mode) –** インストール・モードに **NEW** を指定するには、参照される Java クラスが、現在インストールされているクラスの更新ではなく、新しいクラスである必要があります。 データベース内に同じ名前のクラスがすでにある場合、インストール・モードに **NEW** を指定すると、エラーになります。

  インストール・モードに **UPDATE** を指定することによって、指定したデータベースにすでにインストールされている Java クラスの代わりとして使用する Java クラスを、参照される Java クラスに含めることができます。

  クラスまたは JAR を更新するには、DBA 権限と、ディスク上のファイルで使用できる新バージョンのコンパイルされたクラス・ファイルまたは JAR ファイルが必要です。

  新しい定義を使用するのは、クラスのインストール後に設定された新しい接続か、クラスのインストール後最初にそのクラスを使用する接続だけです。Java VM がクラス定義をロードすると、クラス定義は接続が閉じるまでメモリに保存されます。

  現在の接続で Java クラスまたはクラスを基にしたオブジェクトを使用している場合、新しいクラス定義を使用するには接続をいったん切断し、その後再接続する必要があります。

インストール・モードが省略されると、デフォルトは **NEW** です。

- **JAR –** これを指定する場合、*file-name* または *text-pointer* には、JAR ファイル、または JAR を含むカラムを指定する必要があります。 JAR ファイルには、通常 `.jar` または `.zip` の拡張子が付きます。

  インストールされた JAR および zip ファイルは、圧縮または展開できます。ただし、Sun JDK **jar** ユーティリティによって作成された JAR ファイルはサポートされていません。 それ以外の zip ユーティリティによって作成されたファイルはサポートされています。

  jar オプションを指定する場合、JAR は、含まれているクラスがインストールされた後で JAR として保持されます。 この JAR は、これらの各クラスに関連付けられた JAR です。 **JAR** オプションによってデータベースにインストールされている JAR のセットは、データベースの「保持された JAR」と呼ばれます。

  保持された JAR は、**INSTALL** 文と **REMOVE** 文によって参照されます。保持された JAR は、他の Java-SQL クラスの使用には影響を与えません。SQL システムは、他のシステムによって、指定されたデータに関連付けられたクラスを要求された場合に、保持された JAR を使用します。 要求されたクラスに、関連付けられている JAR がある場合、SQL システムは、個々のクラスではなく、その JAR を提供します。

  jar-name は長さが 255 バイトまでの文字列値です。*jar-name* は、後続の **INSTALL** 文、**UPDATE** 文、**REMOVE** 文に保持された JAR を識別するのに使用します。
- **ソース –** インストールする Java クラスのロケーションを指定します。

  file-name でサポートされるフォーマットは、'`c:¥libs¥jarname.jar`' や '`/usr/u/libs/jarname.jar`' のような完全に修飾されたファイル名と、データベース・サーバの現在の作業ディレクトリを基準にした相対ファイル名です。

  filename には、クラス・ファイルまたは JAR ファイルを指定する必要があります。

  各クラスのクラス定義は、最初にそのクラスを使用するときに各接続の VM でロードされます。クラスを **INSTALL** (インストール) すると、接続の VM が暗黙的に再起動されます。 このため、**INSTALL** 文の *install-mode* が **NEW** か **UPDATE** かに関わらず、すぐに新しいクラスにアクセスできます。

  これ以外の接続では、VM が次に新しいクラスにアクセスするときにロードされます。 クラスがすでに VM によってロードされている場合、VM がその接続のために (たとえば、**STOP JAVA** や **START JAVA** を使用して) 再起動されるまで、接続からはその新しいクラスは見えません。

標準

- SQL—ISO/ANSI SQL 文法のベンダ拡張。
- Sybase—Adaptive Server Enterprise ではサポートされていません。

パーミッション

- **INSTALL** 文を実行するには、DBA 権限が必要です。
- すべてのインストールされたクラスは、すべてのユーザがすべての方法で参照できます。

CREATE PROCEDURE 文 (Java UDF)

外部の Java テーブル UDF へのインタフェースを作成します。

外部プロシージャ向けの **CREATE PROCEDURE** 文のリファレンス情報については、別のトピックを参照してください。テーブル UDF 向けの **CREATE PROCEDURE** 文のリファレンス情報については、別のトピックを参照してください。

Sybase IQ テーブルを参照するクエリは、カタログ・ストア・テーブルのみを参照するクエリとは構文とパラメータが異なります。

Java テーブル UDF は **FROM** 句でのみサポートされています。

構文

少なくとも 1 つの Sybase IQ テーブルを参照するクエリの場合は次のとおりです。

```
CREATE[ OR REPLACE ] PROCEDURE
[ owner.]procedure-name ( [ parameter, …] )
[ RESULT (result-column, ...)]
[ SQL SECURITY { INVOKER | DEFINER } ]
EXTERNAL NAME 'java-call'  [ LANGUAGE Java ] }
```

カタログ・ストア・テーブルのみを参照するクエリの場合は次のとおりです。

```
CREATE[ OR REPLACE ] PROCEDURE
[ owner.]procedure-name ( [ parameter, …] )
[  RESULT (result-column, ...)]
| NO RESULT SET
[ DYNAMIC RESULT SETS integer-expression ]
[ SQL SECURITY { INVOKER | DEFINER } ]
EXTERNAL NAME 'java-call' [ LANGUAGE Java ] }
```

パラメータ

- **parameter –** 少なくとも 1 つの Sybase IQ テーブルを参照するクエリの場合は次のとおりです。

  [**IN***parameter_modeparameter-namedata-type* [**DEFAULT***expression*]

  カタログ・ストア・テーブルのみを参照するクエリの場合は次のとおりです。

[**IN** | **OUT** | **INOUT**] *parameter_mode**parameter-name**data-type* [**DEFAULT***expression*]

- **result-column –** column-name data-type
- **JAVA – [ALLOW | DISALLOW SERVER SIDE REQUESTS]**
- **java-call – '[package-name.]class-name.method-name method-signature'**

使用法

Java テーブル関数では 1 つの結果セットのみを使用できます。 Java テーブル関数を Sybase IQ テーブルとジョインする場合や、Sybase IQ テーブルのカラムが Java テーブル関数の引数である場合、単一の結果セットのみがサポートされます。

FROM 句の項目が Java テーブル関数のみの場合は、*N* 個の結果セットを使用できます。

JAVA: [ALLOW | DISALLOW SERVER SIDE REQUESTS]:

DISALLOW がデフォルトです。

ALLOW はサーバ側接続を許可することを示します。

注意： ALLOW は、必要な場合以外は指定しないでください。**ALLOW** を設定すると、特定の種類の Sybase IQ テーブル・ジョインの速度が低下します。プロシージャ定義を **ALLOW** から **DISALLOW** に (またはその逆に) 変更した場合、新しい接続を確立するまではその変更は認識されません。

UDF を使用するときに **ALLOW SERVER SIDE REQUESTS** と **DISALLOW SERVER SIDE REQUESTS** の両方を同じクエリの中で指定しないでください。

標準

- SQL—ISO/ANSI SQL 準拠。
- Sybase—Transact-SQL の **CREATE PROCEDURE** 文は異なります。
- SQLJ—Java 結果セットの構文拡張は、推奨される SQLJ1 規格に指定されています。

パーミッション

テンポラリ・プロシージャを作成する場合を除き、RESOURCE 権限を持っている必要があります。 DBA 権限を持つユーザは、owner を指定することにより他のユーザの UDF を作成できます。 外部 UDF を作成するか、または他のユーザ向けの外部 UDF を作成するには、DBA 権限が必要です。

プロシージャ内でのテンポラリ・テーブルの参照
プロシージャ間でテンポラリ・テーブルを共有すると、テーブル定義が矛盾している場合に問題が発生する場合があります。

たとえば、procA と procB という 2 つのプロシージャがあり、その両方がテンポラリ・テーブル temp_table を定義して、sharedProc という名前の別のプロシージャを呼び出すとします。procA と procB のどちらもまだ呼び出されていないため、テンポラリ・テーブルはまだ存在しません。

ここで、procA の temp_table の定義が procB の定義と少し異なるとします。両方とも同じカラム名と型を使用していますが、カラムの順序が異なります。

procA を呼び出すと、予期した結果が返されます。一方で、procB を呼び出すと、異なる結果が返されます。

これは、procA が呼び出されたときに temp_table が作成され、その後で sharedProc が呼び出されたためです。sharedProc が呼び出されたときに、内部の **SELECT** 文が解析されて検証され、その後で、解析された文の表現がキャッシュされました。別の **SELECT** 文がもう一度実行されたときに使用できるようにするためです。キャッシュされたバージョンは、procA のテーブル定義のカラムの順序を反映しています。

procB を呼び出すと temp_table が再作成されますが、カラムの順序が異なります。procB が sharedProc を呼び出すと、データベース・サーバは **SELECT** 文のキャッシュされた表現を使用します。そのため、結果が異なります。

次のいずれかを実行すると、このような状況の発生を防ぐことができます。

- このような方法で使用されるテンポラリ・テーブルは、一致するように定義する
- 代わりにグローバル・テンポラリ・テーブルを使用することを検討する

CREATE FUNCTION 文 (Java UDF)

新しい外部 Java テーブル UDF 関数をデータベースで作成します。

構文

```
CREATE [ OR REPLACE | TEMPORARY ] FUNCTION [ owner.]function-name
( [ parameter, … ] )
 RETURNS data-type routine-characteristics
 [ SQL SECURITY { INVOKER | DEFINER } ]
 { compound-statement
   | AS tsql-compound-statement
   | EXTERNAL NAME 'java-call' LANGUAGE JAVA }
```

パラメータ

- **parameter – IN***parameter-namedata-type* [**DEFAULT***expression*]
- **routine-characteristics – ONEXCEPTIONRESUME** | [**NOT**] **DETERMINISTIC**
- **tsql-compound-statement** – *sql-statement sql-statement …*
- **environment-name – JAVA [ALLOW | DISALLOW SERVER SIDE REQUESTS]**
- **java-call** – '[*package-name.*]*class-name.method-namemethod-signature*'
- **method-signature** – ([*field-descriptor, ….*]) *return-descriptor*
- **field-descriptor と return-descriptor – Z** | **B** | **S** | **I** | **J** | **F** | **D** | **C** | **V** | **[** *descriptor* | **L** *class-name***;**

例

- **例 1** – 次の文は、Java で記述された外部関数を作成します。

```
CREATE FUNCTION dba.encrypt( IN name char(254) )
RETURNS VARCHAR
EXTERNAL NAME
'Scramble.encrypt (Ljava/lang/String;)Ljava/lang/String;'
LANGUAGE JAVA
```

使用法

LONG BINARY と LONG VARCHAR は戻り値のデータ型として使用できません。

EXTERNAL NAME LANGUAGE JAVA—**LANGUAGE JAVA** 句が付いた **EXTERNAL NAME** を使用する関数は、Java メソッドのラッパです。

iq-environment-name: JAVA [ALLOW/ *DISALLOW SERVER SIDE REQUESTS*]:

DISALLOW がデフォルトです。

ALLOW はサーバ側接続を許可することを示します。

注意： ALLOW は、必要な場合以外は指定しないでください。**ALLOW** を設定すると、特定の種類の Sybase IQ テーブル・ジョインの速度が低下します。

UDF を使用するときに **ALLOW SERVER SIDE REQUESTS** と **DISALLOW SERVER SIDE REQUESTS** の両方を同じクエリの中で指定しないでください。

標準

- SQL—ISO/ANSI SQL 準拠。
- Sybase—Adaptive Server Enterprise ではサポートされていません。

パーミッション

RESOURCE 権限が必要です。

外部 Java 関数には DBA 権限が必要です。

REMOVE 文

クラス、パッケージ、または JAR ファイルをデータベースから削除します。削除されたクラスは、変数型として利用できなくなります。

構文

```
REMOVE JAVA classes_to_remove
```

パラメータ

- **classes_to_remove：** – { **CLASS***java_class_name* [, *java_class_name*]… | **PACKAGE***java_package_name* [, *java_package_name*]… | **JAR***jar_name* [, *jar_name*]… [**RETAIN CLASSES**] }
- **jar_name：** – *character_string_expression*

例

- **例 1 –** 次の文は、現在のデータベースから "Demo" という名前の Java クラスを削除します。

  ```
  REMOVE JAVA CLASS Demo
  ```

使用法

クラス、パッケージ、または JAR を削除するためには、それがインストールされていることが必要です。

java_class_name—削除される 1 つまたは複数の Java クラスの名前です。これらのクラスは、現在のデータベースにインストールされている必要があります。

java_package_name—削除される 1 つまたは複数の Java パッケージの名前です。これらのパッケージは現在のデータベース内にあるパッケージ名でなければなりません。

jar_name—最大長 255 の文字列値。

各 *jar_name* は、現在のデータベース内に保持されている *jar_name* と同じでなければなりません。*jar_name* と同じかどうかは、SQL システムの文字列比較規則で決定されます。

JAR...RETAIN CLASSES を指定すると、指定した JAR がデータベースに保持されなくなり、保持されたクラスは関連付けられた JAR を持たなくなります。**RETAIN CLASSES** を指定した場合、これが **REMOVE** 文の唯一のアクションになります。

標準

- SQL—ISO/ANSI SQL 文法のベンダ拡張。
- Sybase—Adaptive Server Enterprise ではサポートされていません。同様の機能が、ネストされたトランザクションを使用する Adaptive Server Enterprise 互換の方法で使用できます。

パーミッション

DBA 権限を持っているか、オブジェクトの所有者である必要があります。

START JAVA 文

Java VM を起動します。

構文

```
START EXTERNAL ENVIRONMENT JAVA
```

例

- **例 1 –** Java VM を起動します。

  ```
  START EXTERNAL ENVIRONMENT JAVA
  ```

使用法

START EXTERNAL ENVIRONMENT JAVA を使用して VM を適宜ロードし、ユーザが Java の機能を使い始めるときに、VM ロード中の一時停止が発生しないようにします。

標準

- SQL—ISO/ANSI SQL 文法のベンダ拡張。
- Sybase—なし。

パーミッション

DBA 権限が必要です。

STOP JAVA 文

Java VM に関連するリソースを解放します。

構文

```
STOP EXTERNAL ENVIRONMENT JAVA
```

使用法

STOP EXTERNAL ENVIRONMENT JAVA の主な目的は、システム・リソースを経済的に使用することです。

標準

- SQL—ISO/ANSI SQL 文法のベンダ拡張。
- Sybase—なし。

パーミッション

DBA 権限

Perl 外部環境

Perl のストアド・プロシージャまたは関数は、SQL のストアド・プロシージャまたは関数と同じように動作します。ただし、プロシージャまたは関数のコードは Perl で記述され、プロシージャまたは関数の実行は、データベース・サーバの外部 (つまり、Perl 実行プログラム・インスタンス内) で行われます。

Perl 実行ファイルのインスタンスは、Perl ストアド・プロシージャおよび関数を使用する接続ごとに存在します。この動作は、Java ストアド・プロシージャおよび関数と異なります。Java の場合は、接続ごとに 1 つのインスタンスではなく、各データベースの Java VM に 1 つのインスタンスがあります。Perl と Java のもう 1 つの大きな相違点は、Perl ストアド・プロシージャは結果セットを返さないのに対し、Java ストアド・プロシージャは結果セットを返すことができる点です。

データベースでの Perl のサポートを使用するためには、いくつかの前提条件があります。

1. Perl をデータベース・サーバ・コンピュータにインストールする必要があります。また、データベース・サーバで Perl 実行ファイルを検出できることが必要です。

1. DBD::SQLAnywhere ドライバをデータベース・サーバ・コンピュータにインストールする必要があります。
2. Windows の場合、Microsoft Visual Studio もインストールされていることが必要です。これが前提条件であるのは、DBD::SQLAnywhere ドライバをインストールするために必要だからです。

上記の前提条件に加え、データベース管理者は Perl 外部環境モジュールをインストールする必要もあります。

外部環境モジュールをインストールするには、次の手順に従います (Windows の場合)。

- SDK¥PerlEnv サブディレクトリから、次のコマンドを実行します。

```
perl Makefile.PL
nmake
nmake install
```

外部環境モジュールをインストールするには、次の手順に従います (UNIX の場合)。

- sdk/perlenv サブディレクトリから、次のコマンドを実行します。

```
perl Makefile.PL
make
make install
```

Perl 外部環境モジュールが構築されてインストールされると、データベースでの Perl のサポートを使用できるようになります。

Perl をデータベースで使用するには、データベース・サーバが Perl 実行ファイルを検出して開始できることを確認してください。これは、次の文を実行して確認できます。

```
START EXTERNAL ENVIRONMENT PERL;
```

データベース・サーバが Perl を開始できない場合は、データベース・サーバが Perl 実行ファイルを検出できないことが問題の原因であると考えられます。この場合は、**ALTER EXTERNAL ENVIRONMENT** 文を実行して、Perl 実行ファイルのロケーションを明示的に設定してください。実行ファイル名を必ず含めてください。

```
ALTER EXTERNAL ENVIRONMENT PERL
  LOCATION 'perl-path';
```

次に例を示します。

```
ALTER EXTERNAL ENVIRONMENT PERL
  LOCATION 'c:¥¥Perl¥¥bin¥¥perl.exe';
```

START EXTERNAL ENVIRONMENT PERL 文は、データベース・サーバが Perl を開始できるかどうかを確認する以外で使用することはありません。通常、Perl ストアド・プロシージャまたは関数を呼び出すと、Perl は自動的に開始されます。

これと同様に、Perl のインスタンスを停止するために **STOP EXTERNAL ENVIRONMENT PERL** 文を使用する必要もありません。インスタンスは、接続が切断されると自動的に停止します。ただし、Perl をこれ以上使用することがなく、一部のリソースを解放する必要がある場合は、**STOP EXTERNAL ENVIRONMENT PERL** 文を使用して接続のための Perl インスタンスを解放します。

データベース・サーバが Perl 実行ファイルを開始できることを確認したら、次に必要な Perl コードをデータベースにインストールします。これは、**INSTALL** 文を

使用して行います。たとえば、次の文を実行して Perl スクリプトをファイルからデータベースにインストールできます。

```
INSTALL EXTERNAL OBJECT 'perl-script'
  NEW
  FROM FILE 'perl-file'
  ENVIRONMENT PERL;
```

また、次のように、Perl コードを式から構築してインストールできます。

```
INSTALL EXTERNAL OBJECT 'perl-script'
  NEW
  FROM VALUE 'perl-statements'
  ENVIRONMENT PERL;
```

また、次のように、Perl コードを変数から構築してインストールできます。

```
CREATE VARIABLE PerlVariable LONG VARCHAR;
SET PerlVariable = 'perl-statements';
INSTALL EXTERNAL OBJECT 'perl-script'
  NEW
  FROM VALUE PerlVariable
  ENVIRONMENT PERL;
```

Perl コードをデータベースから削除するには、次のように **REMOVE** 文を使用します。

```
REMOVE EXTERNAL OBJECT 'perl-script'
```

既存の Perl コードを変更するには、次のように **INSTALL EXTERNAL OBJECT** 文の **UPDATE** 句を使用します。

```
INSTALL EXTERNAL OBJECT 'perl-script'
  UPDATE
  FROM FILE 'perl-file'
  ENVIRONMENT PERL
 INSTALL EXTERNAL OBJECT 'perl-script'
  UPDATE
  FROM VALUE 'perl-statements'
  ENVIRONMENT PERL
 SET PerlVariable = 'perl-statements';
INSTALL EXTERNAL OBJECT 'perl-script'
  UPDATE
  FROM VALUE PerlVariable
  ENVIRONMENT PERL
```

Perl コードがデータベースにインストールされたら、必要な Perl ストアド・プロシージャおよび関数を作成できます。Perl ストアド・プロシージャおよび関数を作成するときは、LANGUAGE に必ず PERL を指定します。また、EXTERNAL NAME 文字列には、Perl サブルーチンを呼び出し、OUT パラメータを返して、値を返すために必要な情報が含まれています。次のグローバル変数は、各呼び出し時に Perl コードで使用できます。

- **$sa_perl_return** – これは、関数呼び出しの戻り値を設定するために使用します。
- **$sa_perl_argN** – N は正の整数 [0 .. n] です。これは、SQL 引数を Perl コードに渡すために使用します。たとえば、`$sa_perl_arg0` は引数 0、`$sa_perl_arg1` は引数 1 を示し、以降の引数も同様です。
- **$sa_perl_default_connection** – これは、サーバ側の Perl 呼び出しを作成するために使用します。
- **$sa_output_handle** – これは、Perl コードの出力をデータベース・サーバ・メッセージ・ウィンドウに送信するために使用します。

Perl ストアド・プロシージャは、入出力の引数および戻り値にあらゆるデータ型セットを指定して作成できます。非バイナリのデータ型はすべて Perl 呼び出しの作成時に文字列にマッピングされますが、バイナリ・データは数値の配列にマッピングされます。簡単な Perl の例を次に示します。

```
INSTALL EXTERNAL OBJECT 'SimplePerlExample'
  NEW
  FROM VALUE 'sub SimplePerlSub{
    return( ($_[0] * 1000) +
            ($_[1] * 100) +
            ($_[2] * 10) +
            $_[3] );
  }'
  ENVIRONMENT PERL;

CREATE FUNCTION SimplePerlDemo(
    IN thousands INT,
    IN hundreds INT,
    IN tens INT,
    IN ones INT)
  RETURNS INT
  EXTERNAL NAME '<file=SimplePerlExample>
    $sa_perl_return = SimplePerlSub(
      $sa_perl_arg0,
      $sa_perl_arg1,
      $sa_perl_arg2,
      $sa_perl_arg3)'
  LANGUAGE PERL;

// The number 1234 should appear
SELECT SimplePerlDemo(1,2,3,4);
```

次に示す Perl の例は、文字列を受け取り、それをデータベース・サーバ・メッセージ・ウィンドウに書き込みます。

```
INSTALL EXTERNAL OBJECT 'PerlConsoleExample'
  NEW
  FROM VALUE 'sub WriteToServerConsole { print $sa_output_handle
$_[0]; }'
  ENVIRONMENT PERL;
```

```
CREATE PROCEDURE PerlWriteToConsole( IN str LONG VARCHAR)
  EXTERNAL NAME '<file=PerlConsoleExample>
    WriteToServerConsole( $sa_perl_arg0 )'
  LANGUAGE PERL;

// 'Hello world' should appear in the database server messages window
CALL PerlWriteToConsole( 'Hello world' );
```

サーバ側の Perl を使用するには、Perl コードに *$sa_perl_default_connection* 変数を使用する必要があります。次の例では、テーブルを作成してから Perl ストアド・プロシージャを呼び出して、テーブルにデータを移植します。

```
CREATE TABLE perlTab(c1 int, c2 char(128));

INSTALL EXTERNAL OBJECT 'ServerSidePerlExample'
  NEW
  FROM VALUE 'sub ServerSidePerlSub
    { $sa_perl_default_connection->do(
        "INSERT INTO perlTab SELECT table_id, table_name FROM
SYS.SYSTAB" );
      $sa_perl_default_connection->do(
        "COMMIT" );
    }'
  ENVIRONMENT PERL;

CREATE PROCEDURE PerlPopulateTable()
  EXTERNAL NAME '<file=ServerSidePerlExample> ServerSidePerlSub()'
  LANGUAGE PERL;

CALL PerlPopulateTable();

// The following should return 2 identical rows
SELECT count(*) FROM perlTab
UNION ALL
SELECT count(*) FROM SYS.SYSTAB;
```

PHP 外部環境

PHP のストアド・プロシージャまたは関数は、SQL のストアド・プロシージャまたは関数と同じように動作します。ただし、プロシージャまたは関数のコードは PHP で記述され、プロシージャまたは関数の実行は、データベース・サーバの外部 (つまり、PHP 実行プログラム・インスタンス内) で行われます。

PHP 実行ファイルのインスタンスは、PHP ストアド・プロシージャおよび関数を使用する接続ごとに存在します。この動作は、Java ストアド・プロシージャおよび関数と異なります。Java の場合は、接続ごとに 1 つのインスタンスではなく、各データベースの Java VM に 1 つのインスタンスがあります。PHP と Java のもう 1 つの大きな相違点は、PHP ストアド・プロシージャは結果セットを返さないのに対し、Java ストアド・プロシージャは結果セットを返すことができる点です。

PHP で返すのは、PHP スクリプトの出力である LONG VARCHAR 型のオブジェクトだけです。

データベースでの PHP のサポートを使用するためには、2 つの前提条件があります。

1. PHP をデータベース・サーバ・コンピュータにインストールする必要があります。また、データベース・サーバで PHP 実行ファイルを検出できることが必要です。
2. PHP 拡張をデータベース・サーバ・コンピュータにインストールする必要があります。

上記 2 つの前提条件に加え、データベース管理者は PHP 外部環境モジュールをインストールする必要もあります。いくつかのバージョンの PHP 向けのビルド済みモジュールが含まれています。ビルド済みモジュールをインストールするには、適切なドライバ・モジュールを `php.ini` で指定されている PHP 拡張ディレクトリにコピーします。UNIX では、シンボリック・リンクを使用することもできます。

外部環境モジュールをインストールするには、次の手順に従います (Windows の場合)。

1. PHP インストール・ディレクトリにある `php.ini` ファイルを探して、テキスト・エディタで開きます。`extension_dir` ディレクトリのロケーションを指定する行を探します。`extension_dir` に特定のディレクトリが設定されていない場合は、システム・セキュリティの安全上、独立したディレクトリを指定することをおすすめします。
2. 目的の外部環境 PHP モジュールを、インストール・ディレクトリから PHP のインストール・ディレクトリにコピーします。選択したバージョンを反映するように x.y を変更します。

   ```
   copy "%SQLANY12%¥Bin32¥php-5.x.y_sqlanywhere_extenv12.dll"
     php-dir¥ext
   ```
3. 外部環境 PHP モジュールを自動的にロードするために、次の行を `php.ini` ファイルの Dynamic Extensions セクションに追加します。選択したバージョンを反映するように x.y を変更します。

   ```
   extension=php-5.x.y_sqlanywhere_extenv12.dll
   ```

   `php.ini` を保存して閉じます。
4. PHP ドライバも、インストール・ディレクトリから PHP の拡張ディレクトリにインストールされていることを確認します。ファイル名は `php-5.x.y_sqlanywhere.dll` のようになります (x および y はバージョン番号を表します)。ステップ 2 でコピーしたファイルのバージョン番号と一致する必要があります。

外部環境モジュールをインストールするには、次の手順に従います (UNIX の場合)。

1. PHP インストール・ディレクトリにある `php.ini` ファイルを探して、テキスト・エディタで開きます。`extension_dir` ディレクトリのロケーションを指定する行を探します。`extension_dir` に特定のディレクトリが設定されていない場合は、システム・セキュリティの安全上、独立したディレクトリを指定することをおすすめします。
2. 目的の外部環境 PHP モジュールを、インストール・ディレクトリから PHP のインストール・ディレクトリにコピーします。選択したバージョンを反映するように x.y を変更します。
   ```
   cp $SQLANY12/bin32/php-5.x.y_sqlanywhere_extenv12.so
     php-dir/ext
   ```
3. 外部環境 PHP モジュールを自動的にロードするために、次の行を `php.ini` ファイルの Dynamic Extensions セクションに追加します。選択したバージョンを反映するように x.y を変更します。
   ```
   extension=php-5.x.y_sqlanywhere_extenv12.so
   ```
   `php.ini` を保存して閉じます。
4. PHP ドライバも、インストール・ディレクトリから PHP の拡張ディレクトリにインストールされていることを確認します。ファイル名は `php-5.x.y_sqlanywhere.so` のようになります (x および y はバージョン番号を表します)。ステップ 2 でコピーしたファイルのバージョン番号と一致する必要があります。

PHP をデータベースで使用するには、データベース・サーバが PHP 実行ファイルを検出して開始できる必要があります。データベース・サーバが PHP 実行ファイルを検出して開始できるかどうかを確認するには、次の文を実行します。

```
START EXTERNAL ENVIRONMENT PHP;
```

「外部実行ファイル」が見つからないというメッセージが表示された場合、問題の原因はデータベース・サーバが PHP 実行ファイルを検出できていないことにあります。この場合は、**ALTER EXTERNAL ENVIRONMENT** 文を実行し、PHP 実行ファイルのロケーション (実行ファイル名も含む) を明示的に設定するか、PHP 実行ファイルがあるディレクトリが PATH 環境変数に含まれていることを確認する必要があります。

```
ALTER EXTERNAL ENVIRONMENT PHP
  LOCATION 'php-path';
```

次に例を示します。

```
ALTER EXTERNAL ENVIRONMENT PHP
  LOCATION 'c:¥¥php¥¥php-5.2.6-win32¥¥php.exe';
```

デフォルト設定に戻すには、次の文を実行します。

```
ALTER EXTERNAL ENVIRONMENT PHP
  LOCATION 'php';
```

START EXTERNAL ENVIRONMENT PHP 文は、データベース・サーバが PHP を開始できるかどうかを確認する以外で使用することはありません。通常、PHP ストアド・プロシージャまたは関数を呼び出すと、PHP は自動的に開始されます。

これと同様に、PHP のインスタンスを停止するために **STOP EXTERNAL ENVIRONMENT PHP** 文を使用する必要もありません。インスタンスは、接続が切断されると自動的に停止します。ただし、PHP をこれ以上使用することがなく、一部のリソースを解放する必要がある場合は、**STOP EXTERNAL ENVIRONMENT PHP** 文を使用して接続のための PHP インスタンスを解放します。

データベース・サーバが PHP 実行ファイルを開始できることを確認したら、次に必要な PHP コードをデータベースにインストールします。これは、**INSTALL** 文を使用して行います。たとえば、次の文を実行して、特定の PHP スクリプトをデータベースにインストールできます。

```
INSTALL EXTERNAL OBJECT 'php-script'
  NEW
  FROM FILE 'php-file'
  ENVIRONMENT PHP;
```

PHP コードは、次のようにして式から構築してインストールすることもできます。

```
INSTALL EXTERNAL OBJECT 'php-script'
  NEW
  FROM VALUE 'php-statements'
  ENVIRONMENT PHP;
```

PHP コードは、次のようにして変数から構築してインストールすることもできます。

```
CREATE VARIABLE PHPVariable LONG VARCHAR;
SET PHPVariable = 'php-statements';
INSTALL EXTERNAL OBJECT 'php-script'
  NEW
  FROM VALUE PHPVariable
  ENVIRONMENT PHP;
```

PHP コードをデータベースから削除するには、次のように **REMOVE** 文を使用します。

```
REMOVE EXTERNAL OBJECT 'php-script';
```

既存の PHP コードを変更するには、次のように **INSTALL** 文の **UPDATE** 句を使用します。

```
INSTALL EXTERNAL OBJECT 'php-script'
  UPDATE
  FROM FILE 'php-file'
  ENVIRONMENT PHP;
 INSTALL EXTERNAL OBJECT 'php-script'
```

```
  UPDATE
  FROM VALUE 'php-statements'
  ENVIRONMENT PHP;
 SET PHPVariable = 'php-statements';
INSTALL EXTERNAL OBJECT 'php-script'
  UPDATE
  FROM VALUE PHPVariable
  ENVIRONMENT PHP;
```

PHP コードがデータベースにインストールされたら、次に必要な PHP ストアド・プロシージャおよび関数を作成できます。PHP ストアド・プロシージャおよび関数を作成するときは、LANGUAGE に必ず PHP を指定します。また、EXTERNAL NAME 文字列には、PHP サブルーチンを呼び出し、OUT パラメータを返すために必要な情報が含まれています。

引数は $argv 配列で PHP スクリプトに渡されます。これは、PHP がコマンド・ラインから引数を受け取る方法 ($argv[1] が最初の引数) に似ています。出力パラメータを設定するには、出力パラメータを適切な $argv 要素に割り当てます。戻り値は、常にスクリプトの出力 (LONG VARCHAR) です。

PHP ストアド・プロシージャは、入出力の引数にあらゆるデータ型セットを指定して作成できます。ただし、PHP スクリプト内で使用されるために、パラメータは boolean、integer、double、または string の間で変換されます。戻り値は、常に LONG VARCHAR 型のオブジェクトです。簡単な PHP の例を次に示します。

```
INSTALL EXTERNAL OBJECT 'SimplePHPExample'
  NEW
  FROM VALUE '<?php function SimplePHPFunction(
    $arg1, $arg2, $arg3, $arg4 )
    { return ($arg1 * 1000) +
      ($arg2 * 100) +
      ($arg3 * 10) +
      $arg4;
    } ?>'
  ENVIRONMENT PHP;

CREATE FUNCTION SimplePHPDemo(
  IN thousands INT,
  IN hundreds INT,
  IN tens INT,
  IN ones INT)
RETURNS LONG VARCHAR
EXTERNAL NAME '<file=SimplePHPExample> print SimplePHPFunction(
    $argv[1], $argv[2], $argv[3], $argv[4]);'
LANGUAGE PHP;

// The number 1234 should appear
SELECT SimplePHPDemo(1,2,3,4);
```

PHP では、**EXTERNAL NAME** 文字列は 1 行の SQL 文で指定されます。

サーバ側の PHP を使用するには、PHP コードでデフォルトのデータベース接続を使用します。データベース接続のハンドルを取得するには、空の文字列引数 (" または "") を指定して sasql_pconnect を呼び出します。空の文字列引数を指定することで、新しい外部環境接続を開くのではなく、現在の外部環境接続を返すように PHP ドライバに伝えます。次の例では、テーブルを作成してから PHP ストアド・プロシージャを呼び出して、テーブルにデータを移植します。

```
CREATE TABLE phpTab(c1 int, c2 char(128));

INSTALL EXTERNAL OBJECT 'ServerSidePHPExample'
  NEW
  FROM VALUE '<?php function ServerSidePHPSub() {
    $conn = sasql_pconnect( '''' );
    sasql_query( $conn,
    "INSERT INTO phpTab
       SELECT table_id, table_name FROM SYS.SYSTAB" );
    sasql_commit( $conn );
  } ?>'
  ENVIRONMENT PHP;

CREATE PROCEDURE PHPPopulateTable()
EXTERNAL NAME '<file=ServerSidePHPExample> ServerSidePHPSub()'
LANGUAGE PHP;

CALL PHPPopulateTable();

// The following should return 2 identical rows
SELECT count(*) FROM phpTab
UNION ALL
SELECT count(*) FROM SYS.SYSTAB;
```

PHP では、EXTERNAL NAME 文字列は 1 行の SQL 文で指定されます。上の例では、SQL での引用符の解析方法に従って、単一引用符が二重になっています。PHP ソース・コードがファイル内にある場合は、単一引用符を二重にしません。

エラーをデータベース・サーバに戻すには、PHP 例外処理を実行します。次の例は、これを行う方法を示します。

```
CREATE TABLE phpTab(c1 int, c2 char(128));

INSTALL EXTERNAL OBJECT 'ServerSidePHPExample'
  NEW
  FROM VALUE '<?php function ServerSidePHPSub() {
    $conn = sasql_pconnect( '''' );
    if( !sasql_query( $conn,
      "INSERT INTO phpTabNoExist
         SELECT table_id, table_name FROM SYS.SYSTAB" )
    ) throw new Exception(
      sasql_error( $conn ),
      sasql_errorcode( $conn )
    );
    sasql_commit( $conn );
  } ?>'
```

```
  ENVIRONMENT PHP;

CREATE PROCEDURE PHPPopulateTable()
  EXTERNAL NAME
    '<file=ServerSidePHPExample> ServerSidePHPSub()'
  LANGUAGE PHP;

CALL PHPPopulateTable();
```

上の例は、テーブル phpTabNoExist が見つからないことを示す SQLE_UNHANDLED_EXTENV_EXCEPTION エラーで終了します。

索引

記号

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W

X

し

す

せ

た

て

は

ふ

め

ゆ

ら

り

れ

ろ