ST_Geometry SQL 関数リファレンス

THE SCIENCE OF WHERE

目次

ST_Geometry で使用する SQL 関数

この参照ドキュメントでは、Oracle、PostgreSQL、および SQLite の Esri ST_Geometry 空間データ タイプで使用できる関数のリストを記載し説明します。

Esri ST_Geometry SQL 関数およびタイプは、次のいずれかを実行したときに作成されます。

- Oracle データベースにジオデータベースを作成する。
- PostgreSQL データベースにジオデータベースを作成する際に ST_Geometry を使用する。
- Oracle または PostgreSQL データベースに ST_Geometry 空間データ タイプをインストールする。
- [SQLite データベースの作成 (Create SQLite Database)] ジオプロセシング ツールまたは ArcPy 関数を使用し、ST_Geometry 空間データ タイプを格納する SQLite データベースを作成し、ST_Geometry 関数を読み込んで、データベースとともに使用する。
- モバイル ジオデータベースとともに使用する ST_Geometry 関数を読み込む。

Oracle および PostgreSQL データベースの場合、ST_Geometry タイプおよびその関数は、sde という名前のスキーマに作成されます。 SQLite の場合、タイプと関数は SQLite データベースまたはモバイル ジオデータベースに対して SQL を実行する前に読み込む必要があるライブラリに格納されます。

ヒント:

Esri ST_Geometry タイプの詳細については、ArcGIS Pro ヘルプの次のページをご参照ください。

- PostgreSQL の ST_Geometry
- Oracle の ST_Geometry
- データベースと ST_Geometry
- SQLite ST_Geometry ライブラリの読み込み
- SQL アクセスにおけるモバイル ジオデータベースへの ST_Geometry の読み込み

SQL 関数ページの形式

このドキュメントの関数ページは次のような構成になっています。

- 定義 - その関数が何を行うかについての簡単な説明
- 構文 - その関数を使用するための SQL 構文

注意:

関係演算子の場合は、パラメーターを指定する順序が重要になります。最初のパラメーターでは選択対象のテーブルを指定し、2 つ目のパラメーターではフィルターとして使用するテーブルを指定する必要があります。

- 戻り値のタイプ - その関数を実行したときに返されるデータのタイプ
- 例 - その関数を使用するサンプル

SQL 関数の一覧

以下のリンクをクリックすると、Oracle、PostgreSQL、および SQLite の ST_Geometry タイプで使用できる関数の説明にジャンプします。

ST_Geometry 関数を Oracle で使用する場合、関数と演算子を sde で修飾する必要があります。 たとえば、ST_Buffer は sde.ST_Buffer と指定します。 sde. を追加すると、その関数が sde ユーザーのスキーマに格納されていることが明示されます。 PostgreSQL の場合、この修飾は任意ですが、修飾子を付加することをお勧めします。 SQLite データベースには sde スキーマがないため、SQLite でこの関数を使用する場合は、修飾を付加しないでください。

ST_Geometry SQL 関数で入力として WKT (Well Known Text) 文字列を指定するときは、科学表記法を使用して非常に大きい値や非常に小さい値を指定できます。 たとえば、フィーチャを構築するときに WKT を使用して座標を指定する場合、座標の 1 つが 0.000023500001816501026 とすると、代わりに「2.3500001816501026e-005」と入力できます。

ヒント:

PostGIS タイプ、Oracle SDO_Geometry、Microsoft SQL Server 空間タイプ、IBM Db2 ST_Geometry、SAP HANA ST_Geometry など、他の空間タイプの場合、それぞれに使用される関数の詳細については、データベース管理システム ベンダーが提供するドキュメントをご参照ください。

以下の Esri ST_Geometry SQL 関数は、用途に基づいてグループ分けできます。

コンストラクター関数

コンストラクター関数は、いずれかのジオメトリ タイプまたはジオメトリのテキスト記述を受け取り、ジオメトリを作成します。 次の表は、コンストラクター関数の一覧で、各関数における ST_Geometry 実装の対応状況を示しています。

コンストラクター関数

機能	Oracle	PostgreSQL	SQLite
ST_Centroid	X	X	X
ST_Curve	X		X
ST_GeomCollection	X	X	
ST_GeomCollFromWKB		X	
ST_Geometry	X	X	X
ST_GeomFromText	X		X
ST_GeomFromWKB	X	X	X
ST_LineFromText	X		X
ST_LineFromWKB	X	X	X
ST_LineString	X	X	X
ST_MLineFromText	X		X
ST_MLineFromWKB	X	X	X
ST_MPointFromText	X		X
ST_MPointFromWKB	X	X	X
ST_MPolyFromText	X		X

機能	Oracle	PostgreSQL	SQLite
ST_MPolyFromWKB	X	X	X
ST_MultiCurve	X		
ST_MultiLineString	X	X	X
ST_MultiPoint	X	X	X
ST_MultiPolygon	X	X	X
ST_MultiSurface	X		
ST_Point	X	X	X
ST_PointFromText	X		X
ST_PointFromWKB	X	X	X
ST_PolyFromText	X		X
ST_PolyFromWKB	X	X	X
ST_Polygon	X	X	X
ST_Surface	X		X

アクセサ関数

ジオメトリを入力として、それに関する特定の情報を返す関数がいくつかあります。

一部のアクセサ関数は、1 つまたは複数のフィーチャが特定の条件を満たすかどうかをチェックします。 ジオメトリが条件を満たす場合、関数は 1 (Oracle および SQLite) または t (true) (PostgreSQL) を返します。 ジオメトリが条件を満たさない場合、関数は 0 (Oracle および SQLite) または f (false) (PostgreSQL) を返します。

次の関数は、但し書きがあるものを除いて、すべての実装に適用されます。

アクセサ関数
ST_Area
ST_AsBinary
ST_AsText
ST_CoordDim
ST_Dimension
ST_EndPoint
ST_Entity
ST_Equalsrs (PostgreSQL のみ)
ST_ExteriorRing
ST_GeomFromCollection (PostgreSQL のみ)
ST_GeometryType
ST_GeoSize (PostgreSQL のみ)
ST_Is3d

ST_IsClosed
ST_IsEmpty
ST_IsMeasured
ST_IsRing
ST_IsSimple
ST_Length
ST_M
ST_MaxM
ST_MaxX
ST_MaxY
ST_MaxZ
ST_MinM
ST_MinX
ST_MinY
ST_MinZ
ST_NumGeometries
ST_NumInteriorRing
ST_NumPoints
ST_Perimeter
ST_SRID
ST_StartPoint
ST_X
ST_Y
ST_Z

関係関数

関係関数は、ジオメトリを入力として受け取り、ジオメトリ間に空間リレーションシップが存在するかどうかを判定します。 空間リレーションシップの条件が満たされている場合、関数は 1 (Oracle および SQLite) または t (true) (PostgreSQL) を返します。 空間リレーションシップの条件が満たされていない (リレーションシップが存在しない) 場合、関数は 0 (Oracle および SQLite) または f (false) (PostgreSQL) を返します。

次の関数は、但し書きがあるものを除いて、すべての実装に適用されます。

関係関数

ST_Contains
ST_Crosses
ST_Disjoint

ST_Distance
ST_DWithin
ST_EnvIntersects (Oracle および SQLite のみ)
ST_Equals
ST_Intersects
ST_OrderingEquals (Oracle および PostgreSQL のみ)
ST_Overlaps
ST_Relate
ST_Touches
ST_Within

ジオメトリ処理関数

空間データを入力として、それに対して空間処理を実行して、ジオメトリを返します。

次の関数は、但し書きがあるものを除いて、すべての実装に適用されます。

ジオメトリ処理関数

ST_Aggr_ConvexHull (Oracle および SQLite のみ)
ST_Aggr_Intersection (Oracle および SQLite のみ)
ST_Aggr_Union
ST_Boundary
ST_Buffer
ST_ConvexHull
ST_Difference
ST_Envelope
ST_ExteriorRing
ST_GeometryN
ST_InteriorRingN
ST_Intersection
ST_PointN
ST_PointOnSurface
ST_SymmetricDiff
ST_Transform
ST_Union

SQL と Esri ST_Geometry

データベース管理システムの SQL (Structured Query Language)、データ タイプ、およびテーブル形式を使用して、ST_Geometry タイプがインストールされたデータベースまたはジオデータベースに格納された情報を処理できます。 SQL は、データ定義コマンドとデータ操作コマンドをサポートするデータベース言語です。

SQL 経由でデータにアクセスすることで、ジオデータベースまたはデータベースによって管理されるテーブル形式データを外部アプリケーションから処理できます。 この外部アプリケーションは、非空間データベース アプリケーションであっても、カスタムの空間アプリケーションであってもかまいません。

SQL を使用してジオデータベースまたはデータベースにデータを挿入する場合や、ジオデータベースまたはデータベースのデータを編集する場合は、SQL ステートメントの実行後に、`COMMIT` または `ROLLBACK` ステートメントを発行して、変更内容を確実にデータベースにコミットするか、元に戻してください。 これにより、編集している行、ページ、またはテーブルがロックされたままになるのを防ぐことができます。

SQL を使用して ST_Geometry データを挿入

SQL を使用して、ST_Geometry 列のあるデータベースまたはジオデータベースのテーブルに空間データを挿入できます。 ST_Geometry コンストラクター関数を使用して特定のジオメトリ タイプを挿入します。 特定の空間処理関数の出力を既存のテーブルへの出力とするように指定することもできます。

SQL を使用してジオメトリをテーブルに挿入する場合は、次の点に注意してください。

- 有効な空間参照 ID (SRID) を指定する必要があります。
- 同じ列のすべてのジオメトリが、同じ SRID を使用する必要があります。
- ArcGIS で引き続きこのテーブルを使用する場合、ObjectID として使用されているフィールドに NULL や一意でない値を設定することはできません。

空間参照 ID

ST_Geometry 空間タイプを使用している Oracle のテーブルにジオメトリを挿入するときに指定する SRID は、ST_SPATIAL_REFERENCES テーブルに存在し、対応するレコードが SDE.SPATIAL_REFERENCES テーブルに存在する必要があります。 ST_Geometry 空間タイプを使用している PostgreSQL のテーブルにジオメトリを挿入するときに指定する SRID は、public.sde_spatial_references テーブルに存在する必要があります。 これらのテーブルには、空間参照と SRID が事前に入力されています。

ST_Geometry 空間タイプを使用している SQLite のテーブルにジオメトリを挿入するときに指定する SRID は、st_spatial_reference_systems テーブルに存在する必要があります。

テーブルに存在しないカスタム空間参照を使用する必要がある場合、最も簡単な方法は、を使用して、目的の空間参照値を持つフィーチャクラスを読み込むか作成することです。 このとき、作成するフィーチャクラスには、ST_Geometry 格納を使用します。 こうすると、Oracle の場合は SDE.SPATIAL_REFERENCES テーブルと ST_SPATIAL_REFERENCES テーブルに、PostgreSQL の場合は public.sde_spatial_references テーブルに、SQLite の場合は st_aux_spatial_reference_systems_table に、レコードが作成されます。

ジオデータベースで、空間テーブルに割り当てられた SRID を確認するには、LAYERS (Oracle) または sde_layers (PostgreSQL) テーブルをクエリします。 SQL を使用して空間テーブルを作成してデータを挿入するときは、この SRID を使用できます。

注意:
このドキュメントのサンプルを使用するため、ST_SPATIAL_REFERENCES テーブルと sde_spatial_references テーブルに、不明な空間参照であることを表すレコードが追加されています。 このレコードの SRID は 0 です。 この SRID をこのドキュメントのサンプルに使用できますが、 これは公式の SRID ではなく、サンプル SQL コードを実行するために提供されています。 本番データにはこの SRID を使用しないことをお勧めします。

ObjectID

ArcGIS でデータをクエリするには、テーブルに一意のオブジェクト識別子フィールドが存在する必要があります。

ArcGIS で作成されたフィーチャクラスには、識別子フィールドとして使用される ObjectID フィールドが常に存在します。 ArcGIS を使用してフィーチャクラスにレコードを挿入すると、必ず一意の値が ObjectID フィールドに挿入されます。 ジオデータベース テーブルの ObjectID フィールドは ArcGIS によって管理されます。 ArcGIS から作成されたデータベース テーブルの ObjectID フィールドは、データベース管理システムによって管理されます。

SQL を使用してジオデータベース テーブルにレコードを挿入する場合は、一意の有効な ObjectID 値を挿入する必要があります。

ArcGIS の外部で作成したデータベース テーブルには、ArcGIS が ObjectID として使用できるフィールド (または一連のフィールド) が必要です。 テーブルの ID フィールドに、データベースにネイティブで用意されている自動増加データ タイプを使用する場合、SQL を使用してレコードを挿入したときにデータベースによってこのフィールドに値が設定されます。 一意の識別子フィールドの値を手動で管理する場合は、SQL からテーブルを編集するときに、この ID の値が必ず一意になるようにしてください。

注意:
ArcGIS またはデータベース管理システムによって管理されていない一意の識別子フィールドを持つテーブルからデータを公開することはできません。

SQL を使用して ST_Geometry データを編集

既存のレコードを SQL で編集すると、多くの場合、テーブルに格納された非空間属性が影響を受けます。ただし、SQL UPDATE ステートメントの中で、コンストラクター関数を使用して ST_Geometry 列内のデータを編集することができます。

データがジオデータベースに格納されている場合は、SQL で編集するときに、次のガイドラインに従うことも必要になります。

- データをバージョン対応登録したかジオデータベース履歴管理を有効にした場合は、SQL を使用してレコードを更新しないでください。
- リレーションシップ クラスやフィーチャリンク アノテーション、トポロジ、属性ルール、ネットワークなど、ジオデータベースの振舞いを通じてデータベース内の他のオブジェクトに影響を与える属性は変更しないでください。
- SQL を使用してテーブル スキーマを変更しないでください。

⚠ 注意:

SQL を使用してジオデータベースにアクセスしている場合は、バージョニング、トポロジ、ネットワーク、テレイン、フィーチャリンク アノテーション、その他のクラスまたはワークスペースのエクステンションなど、ジオデータベースの機能は適用されません。 一部のジオデータベース機能に必要なテーブル間の関係を維持するために、トリガーやストアド プロシージャなどのデータベース管理システム機能を使用できる場合があります。 これらの機能を考慮せずにジオデータベースに対して SQL コマンドを実行すると (たとえば、`INSERT` ステートメントを実行してジオデータベース履歴管理が有効になっているテーブルにレコードを追加したり、既存のフィーチャクラスに列を追加したりすると)、ジオデータベース機能が適用されず、ジオデータベース内のデータ間の関係が破損してしまう可能性があります。

SQLite ST_Geometry ライブラリの読み込み

SQLite データベースに対して ST_Geometry 関数を含む SQL コマンドを実行する前に、次の操作を実行します。

1. My Esri から ArcGIS Pro ST_Geometry ライブラリ (SQLite) zip ファイルをダウンロードして解凍します。
2. データベースと同じコンピューターに SQL エディターをインストールします。
3. SQLite データベースと SQL エディターにアクセス可能な場所に ST_Geometry ファイルを配置します。この場所から ST_Geometry を読み込みます。
   SQLite データベースが Microsoft Windows コンピューター上にある場合は、`stgeometry_sqlite.dll` ファイルを使用します。 SQLite データベースが Linux コンピューター上にある場合は、`libstgeometry_sqlite.so` ファイルを使用します。
4. SQL エディターを開き、SQLite データベースに接続します。
5. ST_Geometry ライブラリを読み込みます。
   以下の 1 つ目の例では、ライブラリは Windows コンピューター上の SQLite データベースに読み込まれます。2 つ目の例では、SQLite データベースのライブラリを Linux コンピューター上のデータベースに読み込みます。

```
--Load the ST_Geometry library on Windows.
SELECT load_extension(
 'stgeometry_sqlite.dll',
 'SDE_SQL_funcs_init'
);

--Load the ST_Geometry library on Linux.
SELECT load_extension(
 'libstgeometry_sqlite.so',
 'SDE_SQL_funcs_init'
);
```

これで、SQLite データベースに対して ST_Geometry 関数を含む SQL コマンドを実行できるようになります。

ST_Geometry のコンストラクター関数

コンストラクター関数は、WKT (Well-Known Text) 表現または WKB (Well-Known Binary) からジオメトリを作成します。

ジオメトリを構築するために WKT (Well-Known Text) 表現を入力する場合、メジャーは最後に指定する必要があります。 たとえば、テキストに x、y、z、m の座標を含む場合、この順序で指定しなければなりません。

ジオメトリは 0 個以上のポイントで構成されます。 ポイントの数が 0 である場合、ジオメトリは空であるとみなされます。 ポイント サブタイプは 0 または 1 個のポイントに制限される唯一のジオメトリであり、その他すべてのサブタイプは 0 個以上のポイントで構成されます。

以下のセクションでは、ジオメトリ スーパークラスおよびサブクラス ジオメトリについて説明し、それらを作成できる関数を示します。

既存のジオメトリに対して実行された空間処理の結果としてジオメトリを作成することもできます。

ジオメトリ スーパークラス

ST_Geometry スーパークラスをインスタンス化することはできませんが、列を ST_Geometry タイプとして定義することができます。この列に挿入される実際のデータは、ポイント、ラインストリング、ポリゴン、マルチポイント、マルチラインストリング、またはマルチポリゴンのいずれかのエンティティとして定義します。

以下の関数を使用して、前述したエンティティ タイプのうちのいずれかを保持するスーパークラスを作成できます。

- ST_Geometry
- ST_GeomFromText (Oracle および SQLite のみ)
- ST_GeomFromWKB

サブクラス

フィーチャを特定のサブクラスとして定義できます。そのサブクラスで許可されているエンティティ タイプのみを挿入できます。 たとえば、ST_PointFromWKB は、ポイント エンティティのみを作成できます。

ST_Point

ST_Point は、座標空間で場所を 1 つ占有する 0 次元のジオメトリです。 ST_Point には XY 座標値が 1 つあり、常にシンプルであり、境界は NULL です。 ST_Point は、油田、建造物、水質調査サイトといったフィーチャを定義するために使用されます。

ポイントを作成する関数を以下に示します。

- ST_Point
- ST_PointFromText (Oracle および SQLite のみ)
- ST_PointFromWKB

ST_MultiPoint

ST_MultiPoint は ST_Point のコレクションであり、そのエレメントと同様に 0 次元です。 ST_MultiPoint は、同じ座標空間を占めるエレメントが 1 つもなければ、シンプルです。 ST_MultiPoint の境界は NULL です。
ST_MultiPoint は、地上波パターンや感染症の発生ポイントなどを定義するために使用されます。

マルチポイント ジオメトリを作成する関数を以下に示します。

- ST_MultiPoint
- ST_MPointFromText (Oracle のみ)
- ST_MPointFromWKB

ST_LineString

ST_LineString は、線形補間されたパスを定義する連続したポイントとして格納される 1 次元オブジェクトです。 ST_LineString はその内部と交わっていなければシンプルです。 閉じた ST_LineString の端点 (境界) は、空間内の同じポイントを占めます。 ST_LineString は閉じていてシンプルである場合はリングです。 ST_LineString は ST_Geometry スーパークラスから複数のプロパティを継承しており、その中には長さがあります。 ST_LineString は、道路、河川、電線などの線形フィーチャを定義するためによく使用されます。

端点は、ST_LineString が閉じていなければ、通常は ST_LineString の境界を形成します。ST_LineString が閉じていれば、境界は NULL です。 ST_LineString の内部は、閉じていなければ端点間を結ぶパスであり、閉じている場合は内部が連続しています。

ラインストリングを作成する関数を以下に示します。

- ST_LineString
- ST_LineFromText (Oracle および SQLite のみ)
- ST_LineFromWKB
- ST_Curve (Oracle および SQLite のみ)

ST_MultiLineString

ST_MultiLineString は、ST_LineString のコレクションです。

ST_MultiLineString の境界は、ST_LineString エレメントの交差していない端点です。 ST_MultiLineString のすべてのエレメントのすべての端点が交差している場合、ST_MultiLineString の境界は NULL です。 ST_Geometry スーパークラスから継承される他のプロパティに加えて、ST_MultiLineString には長さがあります。 ST_MultiLineString は、河川や道路網など、連続していないライン フィーチャを定義するために使用されます。

マルチラインストリングを作成する関数を、以下に示します。

- ST_MultiLineString
- ST_MLineFromText (Oracle および SQLite のみ)
- ST_MLineFromWKB
- ST_MultiCurve (Oracle のみ)

ST_Polygon

ST_Polygon は、一連のポイントとして格納される 2 次元サーフェスであり、外部の境界リングと 0 個以上の内部リングを定義します。 ST_Polygon は常にシンプルです。 ST_Polygon は、土地区画、水域、行政区域など、空間的な範囲を持つフィーチャを定義します。

次の図は、ST_Polygon オブジェクトの例を示しています。1 は境界が外部リングによって定義される ST_Polygon です。 2 は境界が外部リングと 2 つの内部リングによって定義される ST_Polygon です。 内部リングの内側にある領域は、ST_Polygon の外部の一部です。 3 は、リングが 1 つの接点で交わっているので、有効な ST_Polygon で

す。

外部リングと内部リングは ST_Polygon の境界を定義し、リング間で囲まれた空間は ST_Polygon の内部を定義します。 ST_Polygon のリングは接点で交わる場合がありますが、決して交差しません。 ST_Geometry スーパークラスから継承される他のプロパティに加えて、ST_Polygon には面積があります。

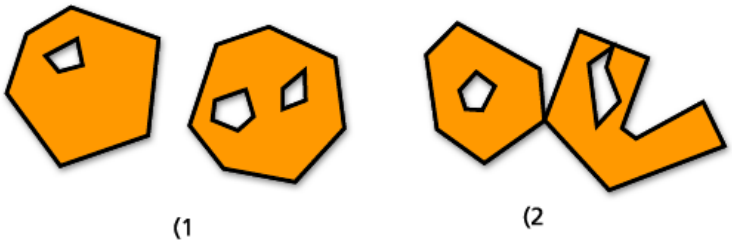
ポリゴンを作成する関数を以下に示します。

- ST_Polygon
- ST_PolyFromText (Oracle および SQLite のみ)
- ST_PolyFromWKB
- ST_Surface (Oracle および SQLite のみ)

ST_MultiPolygon

ST_MultiPolygon の境界は、そのエレメントの外部リングと内部リングの長さを累積したものです。
ST_MultiPolygon の内部は、そのエレメントである ST_Polygon の内郭を累積したものとして定義されます。
ST_MultiPolygon のエレメントの境界は接点でのみ交わる場合があります。 ST_Geometry スーパークラスから継承される他のプロパティに加えて、ST_MultiPolygon には面積があります。 ST_MultiPolygon は、森林地帯や太平洋諸島のような連続していない空間的な範囲を持つフィーチャを定義します。

次の図は、ST_MultiPolygon の例を示しています。1 は 2 つの ST_Polygon エレメントからなる ST_MultiPolygon です。 境界は 2 つの外部リングと 3 つの内部リングによって定義されます。 2 も 2 つの ST_Polygon エレメントからなる ST_MultiPolygon ですが、境界は 2 つの外部リングと 2 つの内部リングによって定義され、2 つの ST_Polygon エレメントは接点で交わっています。



マルチポリゴンを作成する関数を以下に示します。

- ST_MultiPolygon
- ST_MPolyFromText (Oracle および SQLite のみ)
- ST_MPolyFromWKB
- ST_MultiSurface (Oracle のみ)

既存のジオメトリからのジオメトリの作成

以下の関数は、厳密にはコンストラクター関数ではありませんが、既存のジオメトリを入力として、それに対して解析を実行することで、新しいジオメトリを返します。

- ST_Aggr_ConvexHull (Oracle および SQLite のみ)
- ST_Aggr_Intersection (Oracle および SQLite のみ)
- ST_Aggr_Union
- ST_Boundary
- ST_Buffer
- ST_Centroid
- ST_ConvexHull
- ST_Difference
- ST_Envelope
- ST_ExteriorRing
- ST_Intersection
- ST_SymmetricDiff
- ST_Transform
- ST_Union

ST_Geometry 用の空間アクセサ関数

空間アクセサ関数はジオメトリのプロパティを返します。 ST_Geometry フィーチャの次のプロパティを規定するアクセサ関数が用意されています。

次元性

ジオメトリの次元 (ディメンション) とは、ジオメトリの空間範囲を定義するのに必要な最小限の座標 (なし、X、Y) です。

ジオメトリは、0 次元、1 次元、または 2 次元です。

これらの次元は次のように定義されています。

- 0: 長さも面積もない
- 1: 長さがある (X または Y)
- 2: 面積がある (X および Y)

ポイント サブタイプとマルチポイント サブタイプは 0 次元です。 0 次元のフィーチャを表すポイントは 1 つの座標でモデリングできますが、データを表すマルチポイントは接続されていない座標の集団としてモデリングしなければなりません。

ラインストリング サブタイプとマルチラインストリング サブタイプは 1 次元です。 これらは、道路セグメントや河川系の分岐など、線形となるフィーチャを格納します。

ポリゴン サブタイプとマルチポリゴン サブタイプは 2 次元です。 木立、土地区画、水域など、定義可能なエリアを囲む外周を持つフィーチャは、ポリゴン データ タイプまたはマルチポリゴン データ タイプでモデリングできます。

次元は、サブタイプのプロパティとして重要なだけでなく、2 つのフィーチャの空間リレーションシップを決定する上でも重要です。 結果として得られるフィーチャの次元によって、処理が成功したかどうかがわかります。 空間アクセサ関数によってフィーチャの次元を調べることで、それらを比較すべき方法がわかります。

ジオメトリの次元を評価するには、ST_Dimension 関数を使用します。この関数は ST_Geometry フィーチャを受け取り、その次元を整数として返します。

ジオメトリの座標も次元を持ちます。ジオメトリが X 座標と Y 座標しか持たない場合、座標は 2 次元です。ジオメトリが X 座標、Y 座標、Z 座標を持つ場合、座標は 3 次元です。ジオメトリが X 座標、Y 座標、Z 座標、M 座標を持つ場合、座標は 4 次元です。

ST_CoordDim 関数を使用して、ジオメトリの座標値の次元を確認できます。

Z 座標

一部のジオメトリには、高度や深度 (3 次元) が関連付けられています。フィーチャのジオメトリを構成する各ポイントには、必要に応じて、地表面に対する高度や深度を表す Z 座標を含めることができます。

ST_Is3d 関数は、ST_Geometry を入力として受け取り、Z 座標が存在する場合は true を返し、そうでない場合は false を返します。

ポイントの Z 座標は ST_Z 関数を使用して取得できます。

ST_MaxZ 関数はジオメトリの最大 Z 座標を返し、ST_MinZ 関数は最小 Z 座標を返します。

メジャー

メジャーは各座標に割り当てられる値です。メジャーはリニア リファレンス アプリケーションとダイナミック セグメンテーション アプリケーションで使用されます。たとえば、幹線道路沿いの距離標識の場所には、その位置を示すメジャーが含まれることがあります。メジャーの値は、倍精度数値として格納できる任意の値を表します。

ST_IsMeasured 関数は、ST_Geometry を受け取り、メジャーが含まれている場合は true を返し、そうでない場合は false を返します。 この関数は Oracle および SQLite の ST_Geometry 実装だけで使用されます。

ST_M 関数を使用して、ポイントのメジャー値を検索することができます。

ST_MaxM 関数はジオメトリの最大 M 座標を返し、ST_MinM 関数は最小 M 座標を返します。

ジオメトリ タイプ

ジオメトリ タイプは、ジオメトリック エンティティのタイプを表します。たとえば、次のものがあります。

- ポイントとマルチポイント
- ラインとマルチライン
- ポリゴンとマルチポリゴン

ST_Geometry は、さまざまなサブタイプを格納できるスーパークラスです。 ジオメトリのサブタイプを判定するには、ST_GeometryType 関数または ST_Entity 関数 (Oracle および SQLite のみ) を使用します。

ポイント (頂点) のコレクションとポイントの数

ジオメトリは 0 個以上のポイントで構成されます。 ポイントの数が 0 である場合、ジオメトリは空であるとみなされます。 ポイント サブタイプは 0 または 1 個のポイントに制限される唯一のジオメトリであり、その他すべてのサブタイプは 0 個以上のポイントで構成されます。

ST_Point

ST_Point は、座標空間で場所を 1 つ占有する 0 次元のジオメトリです。 ST_Point には XY 座標値が 1 つあり、常にシンプルであり、境界は NULL です。 ST_Point は、油田、建造物、水質調査サイトといったフィーチャを定義するために使用されます。

ST_Point データ タイプだけを対象とする関数には、次のものがあります。

- ST_X は、ポイント データ タイプの X 座標値を倍精度数値として返します。
- ST_Y は、ポイント データ タイプの Y 座標値を倍精度数値として返します。
- ST_Z は、ポイント データ タイプの Z 座標値を倍精度数値として返します。
- ST_M は、ポイント データ タイプの M 座標値を倍精度数値として返します。

ST_MultiPoint

ST_MultiPoint は ST_Point のコレクションであり、そのエレメントと同様に 0 次元です。 ST_MultiPoint は、同じ座標空間を占めるエレメントが 1 つもなければ、シンプルです。 ST_MultiPoint の境界は NULL です。
ST_MultiPoint は、地上波パターンや感染症の発生ポイントなどを定義するために使用されます。

ST_NumGeometries 関数を使用して、マルチポイント ジオメトリのポイントの数を確認できます。

長さ、面積、周長

長さ、面積、周長は、ジオメトリの測定可能な特徴です。 ラインストリングおよびマルチラインストリングのエレメントは 1 次元であり、長さを有します。 ポリゴンとマルチポリゴンのエレメントは 2 次元サーフェスなので、面積と周長の測定が可能です。 ST_Length 関数、ST_Area 関数、および ST_Perimeter 関数を使用して、これらのプロパティを取得できます。 計測単位はデータの格納方法によって変わります。

ST_LineString

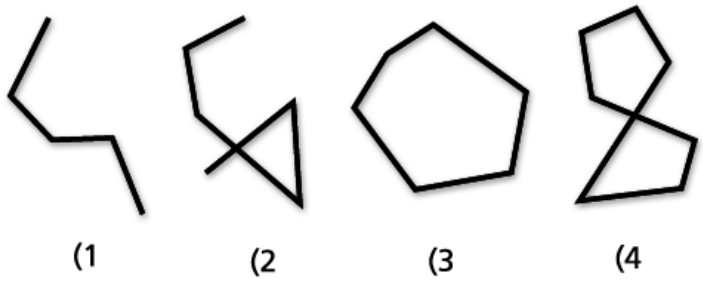
ST_LineString は、線形補間されたパスを定義する連続したポイントとして格納される 1 次元オブジェクトです。 ST_LineString はその内部と交わっていなければシンプルです。 閉じた ST_LineString の端点 (境界) は、空間内の同じポイントを占めます。 ST_LineString は閉じていてシンプルである場合はリングです。 ST_LineString は ST_Geometry スーパークラスから複数のプロパティを継承しており、その中には長さがあります。 ST_LineString は、道路、河川、電線などの線形フィーチャを定義するためによく使用されます。

端点は、ST_LineString が閉じていなければ、通常は ST_LineString の境界を形成します。ST_LineString が閉じていれば、境界は NULL です。 ST_LineString の内部は、閉じていなければ端点間を結ぶパスであり、閉じている場合は内部が連続しています。

ST_LineString を対象とする関数には、次のものがあります。

- ST_StartPoint はラインストリングの最初のポイントを返します。
- ST_EndPoint はラインストリングの最後のポイントを返します。
- ST_IsClosed は、指定された ST_LineString が閉じている (ラインストリングの始点と終点が交わっている) 場合は true を返し、閉じていない場合は false を返します。
- ST_IsRing は、指定された ST_LineString がリングの場合は true を返し、リングでない場合は false を返します。
- ST_Length は ST_LineString の長さを倍精度数値として返します。
- ST_NumPoints は、ST_LineString を評価し、そのポイントの数を整数として返します。
- ST_PointN は、ST_LineString と n 番目のポイントへのインデックスを受け取り、そのポイントを返します。

次の図は ST_LineString オブジェクトの例を示しています。(1 はシンプルで閉じていない ST_LineString、(2 はシンプルではなく閉じていない ST_LineString、(3 は閉じていてシンプルな ST_LineString (リング)、(4 は閉じていてシンプルではない ST_LineString (リングではない) です。



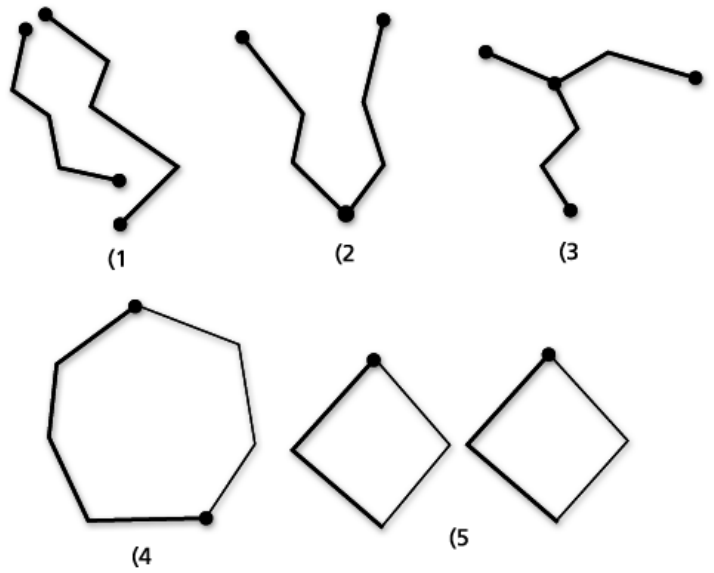
ST_MultiLineString

ST_MultiLineString は、ST_LineString のコレクションです。 ST_MultiLineString は、ST_LineString エレメントの端点でのみ交わる場合はシンプルです。 ST_LineString エレメントの内部が交差している場合、ST_MultiLineString は

シンプルではありません。

ST_MultiLineString の境界は、ST_LineString エレメントの交差していない端点です。 ST_MultiLineString のすべてのエレメントのすべての端点が交差している場合、ST_MultiLineString の境界は NULL です。 ST_Geometry スーパークラスから継承される他のプロパティに加えて、ST_MultiLineString には長さがあります。 ST_MultiLineString は、河川や道路網など、連続していないライン フィーチャを定義するために使用されます。

次の図は ST_MultiLineString の例を示しています。(1 はシンプルな ST_MultiLineString で、その境界は 2 つの ST_LineString エレメントの 4 つの端点です。 (2 は ST_LineString エレメントの端点のみが交わっているため、シンプルな ST_MultiLineString です。 この境界は 2 つの交わっていない端点です。 (3 は ST_LineString エレメントの内部の 1 つが交わっているので、シンプルではない ST_MultiLineString です。 この ST_MultiLineString の境界は、3 つの交わっていない端点です。 (4 はシンプルで閉じていない ST_MultiLineString です。 これが閉じていないのは、ST_LineString エレメントが閉じていないからです。 これがシンプルなのは、ST_LineString エレメントの内部が 1 つも交わっていないからです。 (5 は 1 つのシンプルで閉じている ST_MultiLineString です。 これが閉じているのは、すべてのエレメントが閉じているためです。 これがシンプルなのは、エレメントの内部が 1 つも交わっていないからです。



ST_MultiLineString を対象とする関数には、次のものがあります。

- ST_IsClosed は、指定された ST_MultiLineString が閉じている場合は値 true を返し、閉じていない場合は値 false を返します。
- ST_Length 関数は、ST_MultiLineString を評価し、すべての ST_LineString エレメントの長さを累計した値を倍精度数値として返します。
- ST_NumGeometries 関数は、マルチラインストリングのラインの数を返します。

ST_Polygon

ST_Polygon は、一連のポイントとして格納される 2 次元サーフェスであり、外部の境界リングと 0 個以上の内部リングを定義します。 ST_Polygon は常にシンプルです。 ST_Polygon は、土地区画、水域、行政区域など、空間的な範囲を持つフィーチャを定義します。

次の図は、ST_Polygon オブジェクトの例を示しています。1 は境界が外部リングによって定義される ST_Polygon

です。 2 は境界が外部リングと 2 つの内部リングによって定義される ST_Polygon です。 内部リングの内側にある領域は、ST_Polygon の外部の一部です。 3 は、リングが 1 つの接点で交わっているので、有効な ST_Polygon です。

外部リングと内部リングは ST_Polygon の境界を定義し、リング間で囲まれた空間は ST_Polygon の内部を定義します。 ST_Polygon のリングは接点で交わる場合がありますが、決して交差しません。 ST_Geometry スーパークラスから継承される他のプロパティに加えて、ST_Polygon には面積があります。

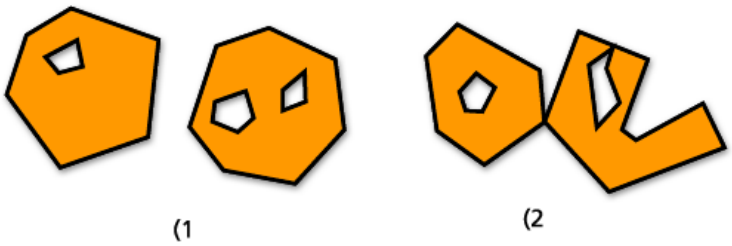
ST_Polygon を対象とする関数には、次のものがあります。

- ST_Area は ST_Polygon の面積を倍精度数値として返します。
- ST_Centroid は ST_Polygon のエンベロープの中心を表す ST_Point を返します。
- ST_ExteriorRing は ST_Polygon の外部リングを ST_LineString として返します。
- ST_InteriorRingN は、ST_Polygon とインデックスを評価し、n 番目の内部リングを ST_LineString として返します。
- ST_NumInteriorRing は ST_Polygon に含まれている内部リングの数を返します。
- ST_PointOnSurface は指定された ST_Polygon のサーフェス上にあることが保証される ST_Point を返します。

ST_MultiPolygon

ST_MultiPolygon の境界は、そのエレメントの外部リングと内部リングの長さを累積したものです。 ST_MultiPolygon の内部は、そのエレメントである ST_Polygon の内郭を累積したものとして定義されます。 ST_MultiPolygon のエレメントの境界は接点でのみ交わる場合があります。 ST_Geometry スーパークラスから継承される他のプロパティに加えて、ST_MultiPolygon には面積があります。 ST_MultiPolygon は、森林地帯や太平洋諸島のような連続していない空間的な範囲を持つフィーチャを定義します。

次の図は、ST_MultiPolygon の例を示しています。1 は 2 つの ST_Polygon エレメントからなる ST_MultiPolygon です。 境界は 2 つの外部リングと 3 つの内部リングによって定義されます。 2 も 2 つの ST_Polygon エレメントからなる ST_MultiPolygon ですが、境界は 2 つの外部リングと 2 つの内部リングによって定義され、2 つの ST_Polygon エレメントは接点で交わっています。



ST_MultiPolygon を対象とする関数には、次のものがあります。

- ST_Area は、ST_MultiPolygon の ST_Polygon エレメントの累積 ST_Area を表す倍精度数値を返します。

- ST_Centroid は、ST_MultiPolygon のエンベロープの中心である ST_Point を返します。
- ST_NumGeometries は、マルチポリゴンのポリゴンの数を返します。
- ST_PointOnSurface は、ST_MultiPolygon を評価し、その ST_Polygon エレメントのいずれかのサーフェス上にあることが保証される ST_Point を返します。

マルチパート ジオメトリのシンプル ジオメトリ

マルチパート ジオメトリは複数の個別のシンプル ジオメトリから構成されています。

ST_MultiPoint、ST_MultiLineString、ST_MultiPolygon といったマルチパート ジオメトリに含まれる個々のジオメトリの数を取得する場合は、 ST_NumGeometries 関数を使用します。 この関数は、ジオメトリのコレクションに含まれる個々のエレメントの数を返します。

ST_GeometryN 関数を使用して、マルチパート ジオメトリの N 番目の位置にあるジオメトリを取得できます。ここで N は関数に指定した数値です。 たとえば、マルチポイント ジオメトリの 3 番目のポイントを取得する場合、関数の実行時に 3 を指定します。

PostgreSQL のマルチパート ジオメトリから個々のジオメトリとその位置を返すには、ST_GeomFromCollection 関数を使用します。

内部、外部、境界

すべてのジオメトリはその内部、外部、境界によって定義された空間内の位置を占めます。ジオメトリの外部は、そのジオメトリが占有していない空間すべてです。ジオメトリの内部は、そのジオメトリが占有している空間です。ジオメトリの境界は、その内部と外部の間に位置する場所です。サブタイプは内部プロパティと外部プロパティを直接継承しますが、境界プロパティはサブタイプごとに異なります。

ソース ST_Geometry の境界を取得するには、ST_Boundary 関数を使用します。

シンプルかシンプルでないか

ST_Geometry の ST_Point や ST_Polygon といったサブタイプは常にシンプルです。 ただし、ST_LineString、ST_MultiPoint、ST_MultiLineString サブタイプは、シンプルな場合とシンプルでない場合があります。 それらがシンプルなのは、それらに適用されるトポロジ ルールがすべて満たされている場合です。

トポロジ ルールとしては、次のようなものがあります。

- ST_LineString がシンプルなのは、その内部が交わっていない場合であり、交わっている場合はシンプルではありません。
- ST_MultiPoint がシンプルなのは、2 つのエレメントが同じ座標空間を占める (XY 座標が同じである) ことがない場合であり、そうでない場合はシンプルではありません。
- ST_MultiLineString がシンプルなのは、マルチラインストリングの内部とそのエレメントの内部が交わっていない場合であり、エレメントの内部が交わっている場合はシンプルではありません。

ST_LineString、ST_MultiPoint、ST_MultiLineString がシンプルかどうかを判断するには、ST_IsSimple 関数を使用します。 ST_IsSimple は、ST_Geometry を受け取り、ジオメトリがシンプルである場合は true、シンプルでない場合は false を返します。

空か空でないか

ジオメトリが空なのは、ポイントが 1 つもない場合です。空のジオメトリのエンベロープ、境界、内部、外部は null です。空のジオメトリは常にシンプルです。空のラインストリングとマルチラインストリングの長さは 0 で

す。空のポリゴンとマルチポリゴンの面積は 0 です。

ジオメトリが空かどうかを判断するには、ST_IsEmpty 関数を使用します。 この関数は、ST_Geometry を分析して、ジオメトリが空である場合は true、空でない場合は false を返します。

閉じているかリングか

ラインストリング ジオメトリは、閉じている場合とリングの場合があります。 ラインストリングは、リングでなくても閉じていることがあります。 ST_IsClosed 関数を使用して、ラインストリングが閉じているかどうかを判断できます。この関数は、ラインストリングの始点と終点が交わっている場合に true を返します。 リングとは、閉じていてシンプルなラインストリングです。 ラインストリングがリングであるかどうかを判断するには、ST_IsRing 関数を使用します。この関数は、ラインストリングが閉じていてシンプルな場合に true を返します。

エンベロープ

すべてのジオメトリにはエンベロープがあります。ジオメトリのエンベロープは、XY 座標の最小値と最大値によって形成される境界を示すジオメトリです。ポイント ジオメトリの場合、XY 座標の最小値と最大値は同じであるため、座標の回りに四角形すなわちエンベロープが作成されます。ライン ジオメトリの場合、ラインの両端がエンベロープの 2 辺を表し、残りの 2 辺はラインのすぐ上とすぐ下に作成されます。

ST_Envelope 関数は、ST_Geometry を受け取り、ソース ST_Geometry のエンベロープを表す ST_Geometry を返します。

ジオメトリの最小および最大 X、Y 座標値を個別に取得するには、ST_MinX、ST_MinY、ST_MaxX、ST_MaxY の各関数を使用します。

空間参照系

空間参照系は、各ジオメトリの座標変換マトリックスを識別します。座標系、解像度、許容値から構成されます。

ジオデータベースで認識される空間参照系はすべて、ジオデータベース システム テーブルに格納されます。

ジオメトリの空間参照系に関する情報を取得するには、次の関数が使用されます。

- ST_SRID は、ST_Geometry を受け取り、空間参照 ID (SRID) を整数として返します。
- ST_Equalsrs は、2 つの異なるフィーチャクラスの空間参照系が同一 (true) か同一でないか (false) を判定します。

フィーチャのサイズ (PostgreSQL のみ)

フィーチャ (テーブル内の空間レコード) は一定の量 (バイト単位) の格納領域を占めます。 ST_GeoSize 関数を使用して、テーブル内の各フィーチャのサイズを判断できます。

ジオメトリのテキスト定義とバイナリ定義

空間テーブルの特定の行のジオメトリの WKT (Well-Known Text) 定義を取得するには ST_AsText 関数を使用し、WKB (Well-Known Binary) 定義を取得するには ST_AsBinary 関数を使用します。

空間リレーションシップ

GIS の主要な目的は、あるフィーチャ同士が重なり合うのか、 一方が他方に包含されているのか、 一方が他方と交差しているのかといった、フィーチャ間の空間リレーションシップの判断です。

ジオメトリの空間リレーションシップはさまざまなものが考えられます。 以下に、ジオメトリが別のジオメトリとの間にどのような空間リレーションシップをもつことができるかの例を示します。

- ジオメトリ B を通過するジオメトリ A。
- ジオメトリ B に完全に包含されたジオメトリ A。
- ジオメトリ B を完全に包含したジオメトリ A。
- 互いに交差または接触しないジオメトリ。
- 完全に一致するジオメトリ。
- 相互に重なり合うジオメトリ。
- 一点で接触するジオメトリ。

こうしたリレーションシップの有無を判断するには、空間リレーションシップ関数を使用します。 これらの関数では、クエリで指定したジオメトリについて次のプロパティが比較されます。

- ジオメトリによって占有されていないすべての空間である、ジオメトリ外部 (E)
- ジオメトリによって占有されている空間である、ジオメトリ内部 (I)
- ジオメトリの内部と外部の境界である、ジオメトリ境界 (B)

空間リレーションシップのクエリを構築する際は、検索する空間リレーションシップのタイプおよび比較の対象となるジオメトリを指定します。 クエリが true または false を返します。 つまり、指定した空間リレーションシップがジオメトリ間に存在するかどうかが判別されます。 多くの場合、空間リレーションシップのクエリを WHERE 句に設定することで、結果セットにフィルターを適用するために使用します。

たとえば、提案されている開発用地の位置が格納されたテーブルと、考古学的に重要な場所の位置が格納されたテーブルがある場合には、考古学的な場所と交差する開発用地がないか調べ、交差がある場合は該当する開発案の ID を返すクエリを実行できます。 この例では、PostgreSQL で ST_Disjoint 関数を使用しています。

```
SELECT d.projname,a.siteid
 FROM dev d, archsites a
 WHERE sde.st_disjoint(d.shape,a.shape)= 'f'

projname        siteid

bow wow chow    A1009
```

このクエリでは、分断されていない、つまり互いに交差している開発用地の名前 (projname) と考古学的な場所の ID (siteid) が返されます。 考古学的な場所である A1009 と交差している開発プロジェクト Bow Wow Chow が返されています。

ST_Geometry の関係関数

関係関数は、異なるタイプの空間リレーションシップをテストするために、述語を使用します。 これらの評価では、次の間のリレーションシップが比較されます。

- ジオメトリによって占有されていないすべての空間である、ジオメトリ外部 (E)
- ジオメトリによって占有されている空間である、ジオメトリ内部 (I)
- ジオメトリの内部と外部の境界である、ジオメトリ境界 (B)

リレーションシップは述語によって評価されます。 比較結果が関数の条件を満たす場合は、1 (Oracle と SQLite) または t (PostgreSQL) が返され、それ以外の場合は 0 (Oracle と SQLite) または f (PostgreSQL) が返されます。 空間リレーションシップを評価する述語は、一対のジオメトリを比較します。これらは異なるタイプやディメンションであってもかまいません。

述語によって比較されるのは、送信されたジオメトリの X 座標と Y 座標です。 Z 座標とメジャー値がある場合でも、これらは無視されます。 Z 座標とメジャーをもつジオメトリを、Z 座標とメジャーをもたないジオメトリと比較することもできます。

Clementini らによって開発された Dimensionally Extended 9 Intersection Model (DE-9IM) は、Egenhofer と Herring の 9 Intersection Model をディメンション的に拡張したものです。 DE-9IM は、タイプやディメンションが異なるジオメトリ間のペアワイズ空間リレーションシップを定義する数学的手法です。 このモデルは、あらゆる種類のジオメトリ間の空間リレーションシップを、結果として得られる交差のディメンションを考慮した、内部、境界、および外部のペアワイズ交差として表現します。

ジオメトリ a とジオメトリ b があり、a の内部、境界、外部がそれぞれ I(a)、B(a)、E(a)、b の内部、境界、外部がそれぞれ I(b)、B(b)、E(b) であるとします。 I(a)、B(a)、E(a) と I(b)、B(b)、E(b) の間の交差によって、3 x 3 のマトリックスが生成されます。 それぞれの交差によってディメンションの異なるジオメトリが生じることがあります。 たとえば、2 つのポリゴンの境界の交差が 1 つのポイントと 1 本のラインストリングで構成されることが考えられます。この場合に dim (ディメンション) 関数を使用すると、最大ディメンションとして 1 が返されます。

dim 関数は -1、0、1、または 2 の値を返します。 値 -1 は NULL セットに対応しており、交差が見つからない場合または dim(Ãƒ˜) の場合に返されます。

	内部	境界	外部
内部	dim(I(a) intersects I(b))	dim(I(a) intersects B(b))	dim(I(a) intersects E(b))
境界	dim(B(a) intersects I(b))	dim(B(a) intersects B(b))	dim(B(a) intersects E(b))
外部	dim(E(a) intersects I(b))	dim(E(a) intersects B(b))	dim(E(a) intersects E(b))

述語の交差例

空間リレーションシップの述語の結果は、DE-9IM の許容値を表すパターン マトリックスと比較することによって理解および確認できます。

パターン マトリックスには、各交差マトリックス セルに許容値が格納されます。 考えられるパターン値は次のとおりです。

T - 交差が存在する。dim = 0、1、または 2

F - 交差が存在しない。dim = -1

* - 交差が存在していても存在していなくてもよい。dim = -1、0、1、または 2

0 - 交差が存在し、最大ディメンションが 0 である。dim = 0

1 - 交差が存在し、最大ディメンションが 1 である。dim = 1

2 - 交差が存在し、最大ディメンションが 2 である。dim = 2

それぞれの述語には少なくとも 1 つのパターン マトリックスがありますが、述語によっては、さまざまなジオメトリ タイプを組み合わせたリレーションシップを説明するために 2 つ以上必要な場合もあります。

ジオメトリの組み合わせについての述語 ST_Within のパターン マトリックスには、次の形式があります。

			ジオメトリ b	
		内部	境界	外部
ジオメトリ a	内部	T	*	F
	境界	*	*	F
	外部	*	*	*

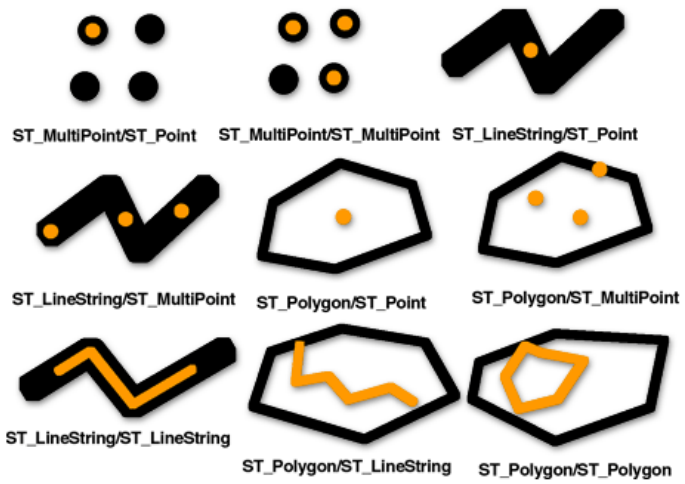
パターン マトリックスの例

述語 ST_Within は、両方のジオメトリの内部が交差し、かつジオメトリ a の内部も境界もジオメトリ b の外部と交差していない場合に true を返します。 これ以外の条件はいずれも影響しません。

以下のセクションでは、空間リレーションシップについて使用するさまざまな述語について説明します。 各セクションの図では、第 1 の入力ジオメトリが黒で、第 2 の入力ジオメトリがオレンジで示されています。

ST_Contains

ST_Contains は、第 2 のジオメトリが第 1 のジオメトリに完全に包含されている場合に 1 または t (true) を返します。 述語 ST_Contains が返す結果は、述語 ST_Within とはまったく逆になります。



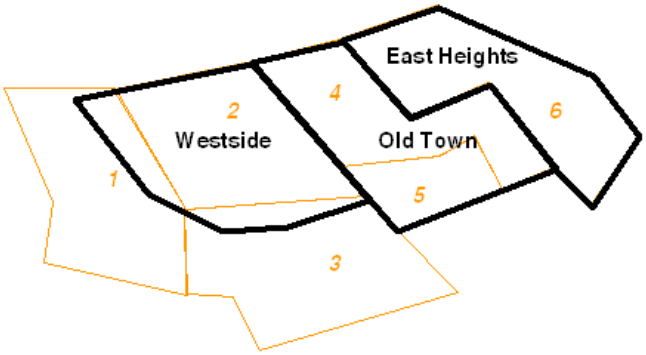
述語 ST_Contains のパターン マトリックスには、両方のジオメトリの内部が交差しており、かつ第 2 のジオメトリ (ジオメトリ b) の内部と境界が第 1 のジオメトリ (ジオメトリ a) の外部と交差していない必要があると記載されています。

			ジオメトリ b	

ST_Contains マトリックス

		内部	境界	外部
ジオメトリ a	内部	T	*	*
	境界	*	*	*
	外部	F	F	*

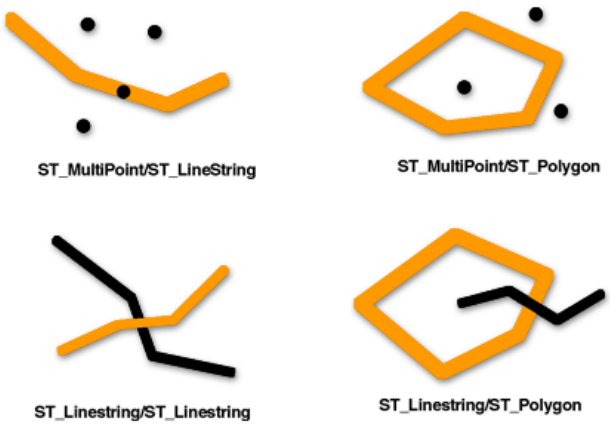
ST_Within または ST_Contains 関数を使用した場合は、別のジオメトリ内に完全に収まるジオメトリだけを検索できます。 これにより、結果に歪みを生じさせる可能性のあるフィーチャを選択から排除できます。 以下の例では、アイスクリームの移動販売者が、移動ルートを該当する地区に限定するために、子供 (潜在顧客) が最も多い地区を知りたいと思っています。 販売者は、指定した地区のポリゴンを、16 歳未満の子供の合計数という属性を持つ国勢調査地区と比較します。



国勢調査地区 1 と国勢調査地区 3 に住むすべての子供が Westside 地区に重なっている細長い部分に住んでいない限り、これらの国勢調査地区を選択に含めると、Westside 地区内の子供の人数が誤って多くなってしまう可能性があります。 各地区に完全に収まっている国勢調査地区だけを含めるよう指定する (ST_Within = 1) ことにより、アイスクリーム販売者は前述した Westside 地区の該当部分で時間とお金を浪費するリスクを回避できます。

ST_Crosses

ST_Crosses は、交差により 2 つのソース ジオメトリの最大ディメンションよりも 1 段階小さいディメンションを持つジオメトリとなり、その交差セットが両方のソース ジオメトリの内部にある場合に、1 または t (true) を返します。 1 または t (true) が返されるのは、ST_MultiPoint/ST_Polygon、ST_MultiPoint/ST_LineString、ST_Linestring/ST_LineString、ST_LineString/ST_Polygon、および ST_LineString/ST_MultiPolygon の比較に ST_Crosses を使用した場合だけです。



次の述語 ST_Crosses のパターン マトリックスは、ST_MultiPoint/ST_LineString、ST_MultiPoint/

ST_MultiLineString、ST_MultiPoint/ST_Polygon、ST_MultiPoint/ST_MultiPolygon、ST_LineString/ST_Polygon、および ST_LineString/ST_MultiPolygon に適用されます。 マトリックスには、内部が交差しており、かつ少なくとも第 1 のジオメトリ (ジオメトリ a) の内部が第 2 のジオメトリ (ジオメトリ b) の外部と交差している必要があると記載されています。

			ジオメトリ b	
		内部	境界	外部
ジオメトリ a	内部	T	*	T
	境界	*	*	*
	外部	*	*	*

ST_Crosses マトリックス 1

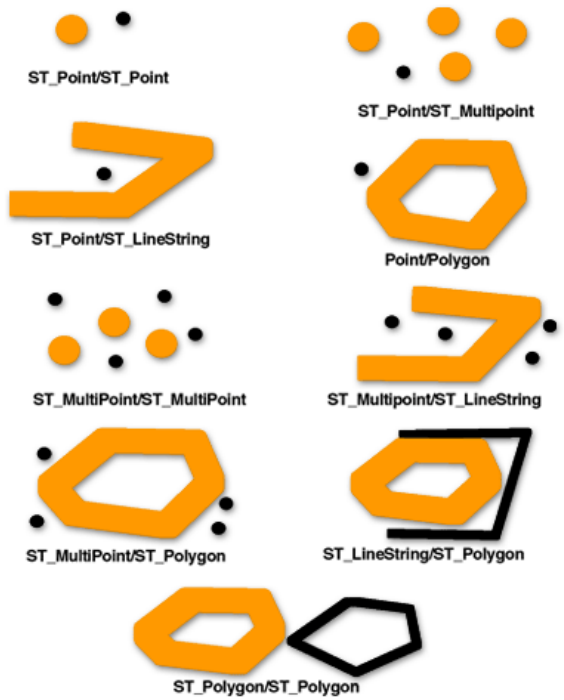
次の述語 ST_Crosses のマトリックスは、ST_LineString/ST_LineString、ST_LineString/ST_MultiLineString、および ST_MultiLineString/ST_MultiLineString に適用されます。 マトリックスには、内部の交差のディメンションが 0 (ポイントでの交差) でなければならないと記載されています。 この交差のディメンションが 1 (ラインストリングでの交差) の場合、述語 ST_Crosses によって false が返されます。ただし、この場合に述語 ST_Overlaps を使用すると、true が返されます。

			ジオメトリ b	
		内部	境界	外部
ジオメトリ a	内部	0	*	*
	境界	*	*	*
	外部	*	*	*

ST_Crosses マトリックス 2

ST_Disjoint

ST_Disjoint では、2 つのジオメトリの交差が空のセットの場合に 1 または t (true) が返されます。 互いに交差していないジオメトリは分断しています。



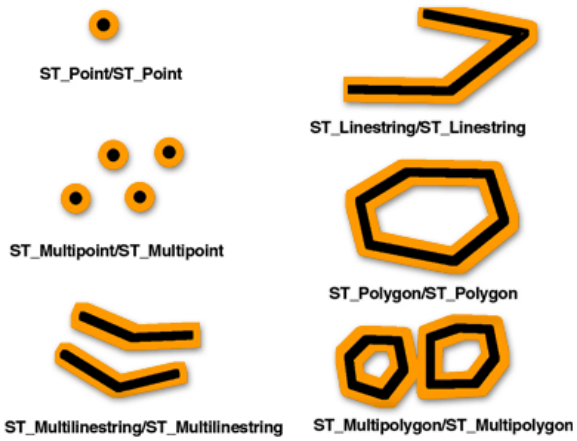
述語 ST_Disjoint のパターン マトリックスには、いずれのジオメトリの内部も境界も交差しないと記載されています。

			ジオメトリ b	
		内部	境界	外部
ジオメトリ a	内部	F	F	*
	境界	F	F	*
	外部	*	*	*

ST_Disjoint マトリックス

ST_Equals

ST_Equals は、同じタイプの 2 つのジオメトリの X 座標と Y 座標が同じである場合に 1 または t (true) を返します。 オフィス ビルの 1 階と 2 階であっても、X 座標と Y 座標が同じ、つまり等価となる場合があります。 ST_Equals では、2 つのフィーチャが誤って互いに上下に配置されていないかどうかも特定することができます。



等価性についての DE-9IM パターン マトリックスでは、内部同士が交差しており、一方のジオメトリの内部や境界に他方のジオメトリの外部と交差する部分のない状態を保証しています。

			ジオメトリ b	
		内部	境界	外部
ジオメトリ a	内部	T	*	F
	境界	*	*	F
	外部	F	F	*

ST_Equals マトリックス

ST_Intersects

ST_Intersects は、交差が空のセットにならない場合に 1 または t (true) を返します。 ST_Intersects は、ST_Disjoint とは正反対の結果を返します。

述語 ST_Intersects は、次のパターン マトリックスのいずれかの条件が true を返した場合に、true を返します。

述語 ST_Intersects は、両方のジオメトリの内部同士が交差する場合に true を返します。

			ジオメトリ b	
		内部	境界	外部
ジオメトリ a	内部	T	*	*
	境界	*	*	*
	外部	*	*	*

ST_Intersects マトリックス 1

述語 ST_Intersects は、第 1 のジオメトリの内部が第 2 のジオメトリの境界と交差している場合に true を返します。

			ジオメトリ b	
		内部	境界	外部

ST_Intersects マトリックス 2

ジオメトリ a	**内部**	*	T	*
	境界	*	*	*
	外部	*	*	*

述語 ST_Intersects は、第 1 のジオメトリの境界が第 2 のジオメトリの内部と交差している場合に true を返します。

			ジオメトリ b	
		内部	**境界**	**外部**
ジオメトリ a	**内部**	*	*	*
	境界	T	*	*
	外部	*	*	*

ST_Intersects マトリックス 3

述語 ST_Intersects は、両方のジオメトリの境界が交差している場合に true を返します。

			ジオメトリ b	
		内部	**境界**	**外部**
ジオメトリ a	**内部**	*	*	*
	境界	*	T	*
	外部	*	*	*

ST_Intersects マトリックス 4

ST_Overlaps

ST_Overlaps は、同じディメンションの 2 つのジオメトリを比較して、結果の交差セットが、ディメンションが等しくかつどちらの入力ジオメトリとも異なるジオメトリになる場合に、1 または t (true) を返します。

ST_Overlaps は、ディメンションが同じジオメトリであり、かつ交差セットが、同じディメンションのジオメトリを生じさせる場合にのみ、1 または t (true) を返します。 つまり、2 つの ST_Polygons の交差が 1 つの ST_Polygon になる場合、Overlap は 1 または t (true) を返します。

ST_LineString/ST_LineString

ST_Polygon/ST_Polygon

ST_MultiPoint/ST_MultiPoint

次のパターン マトリックスは、ST_Polygon/ST_Polygon、ST_MultiPoint/ST_MultiPoint、および ST_MultiPolygon/ST_MultiPolygon のオーバーレイに適用されます。 これらの組み合わせである場合、述語 Overlap は、両方のジオメトリの内部が他方の内部および外部と交差する場合に true を返します。

			ジオメトリ b	
		内部	境界	外部
ジオメトリ a	内部	T	*	T
	境界	*	*	*
	外部	T	*	*

ST_Overlaps マトリックス 1

次のパターン マトリックスは ST_LineString/ST_LineString と ST_MultiLineString/ST_MultiLineString のオーバーラップ重なりに適用されます。 この場合、ジオメトリの交差はディメンションが 1 のジオメトリ (別の ST_LineString または ST_MultiLineString) になる必要があります。 内部の交差のディメンションが 0 (ポイント) になる場合、述語 ST_Overlaps は false を返しますが、 ST_Crosses は true を返します。

			ジオメトリ b	
		内部	境界	外部
ジオメトリ a	内部	1	*	T
	境界	*	*	*
	外部	T	*	*

ST_Overlaps マトリックス 2

ST_Relate

ST_Relate は、パターン マトリックスによって指定される空間リレーションシップが有効な場合に 1 または t (true) を返します。 1 または t (true) という値は、ジオメトリ間に何らかの空間リレーションシップがあることを示します。

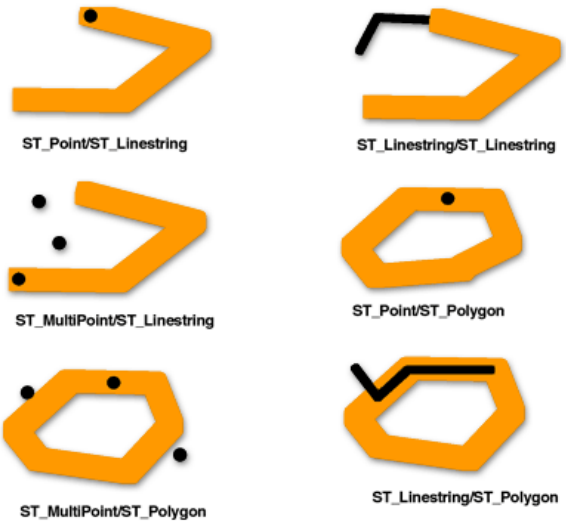
ジオメトリ a とジオメトリ b の内部または境界に何らかの関係がある場合には、ST_Relate が true になります。一方のジオメトリの外部が他方のジオメトリの内部または境界と交差しているかどうかは関係ありません。

			ジオメトリ b	
		内部	境界	外部
ジオメトリ a	内部	T	T	*
	境界	T	T	*
	外部	*	*	*

ST_Relate マトリックス

ST_Touches

ST_Touches は、両方のジオメトリに共通するポイントがどれも両方のジオメトリの内部と交差していない場合に 1 または t (true) を返します。 少なくとも 1 つのジオメトリが ST_LineString、ST_Polygon、ST_MultiLineString、または ST_MultiPolygon でなければなりません。



パターン マトリックスには、述語 ST_Touches が true を返すのは、ジオメトリの内部に交差がなく、一方のジオメトリの境界が他方のジオメトリの内部または境界と交差している場合であると記載されています。

述語 ST_Touches は、ジオメトリ b の境界がジオメトリ a の内部と交差している一方で、内部同士は交差していない場合に true を返します。

			ジオメトリ b	
		内部	**境界**	**外部**
ジオメトリ a	**内部**	F	T	*
	境界	*	*	*
	外部	*	*	*

ST_Touches マトリックス 1

述語 ST_Touches は、ジオメトリ a の境界がジオメトリ b の内部と交差している一方で、内部同士は交差していない場合に true を返します。

			ジオメトリ b	
		内部	**境界**	**外部**
ジオメトリ a	**内部**	F	*	*
	境界	T	*	*
	外部	*	*	*

ST_Touches マトリックス 2

述語 ST_Touches は、両方のジオメトリの境界が交差している一方で、内部が交差していない場合に true を返します。

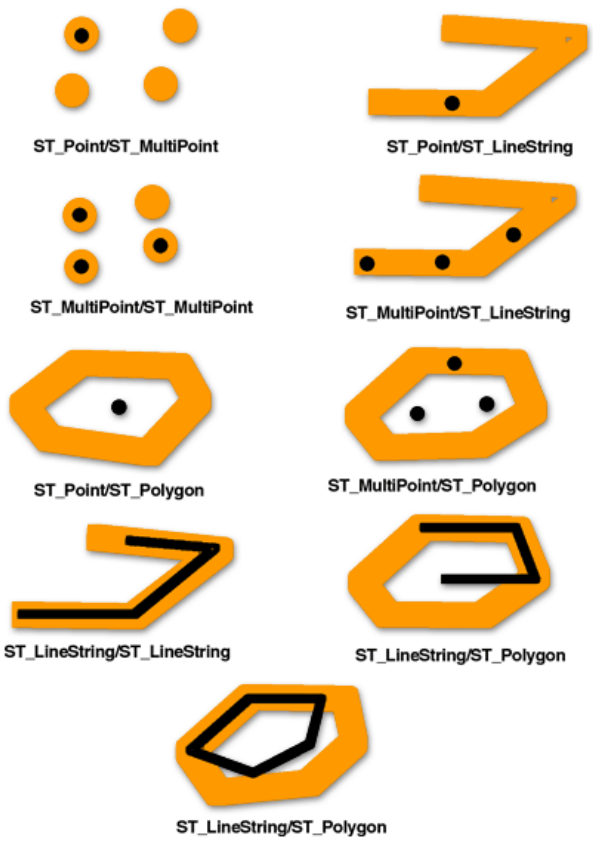
			ジオメトリ b	
		内部	**境界**	**外部**

ST_Touches マトリックス 3

ジオメトリ a	**内部**	F	*	*
	境界	*	T	*
	外部	*	*	*

ST_Within

ST_Within は、第 1 のジオメトリが完全に第 2 のジオメトリ内にある場合に 1 または t (true) を返します。ST_Within は、ST_Contains の結果とは正反対の状態についての評価です。



述語 ST_Within のパターン マトリックスには、両方のジオメトリの内部が交差しており、かつ第 1 のジオメトリ (ジオメトリ a) の内部と境界が第 2 のジオメトリ (ジオメトリ b) の外部と交差していない必要があると記載されています。

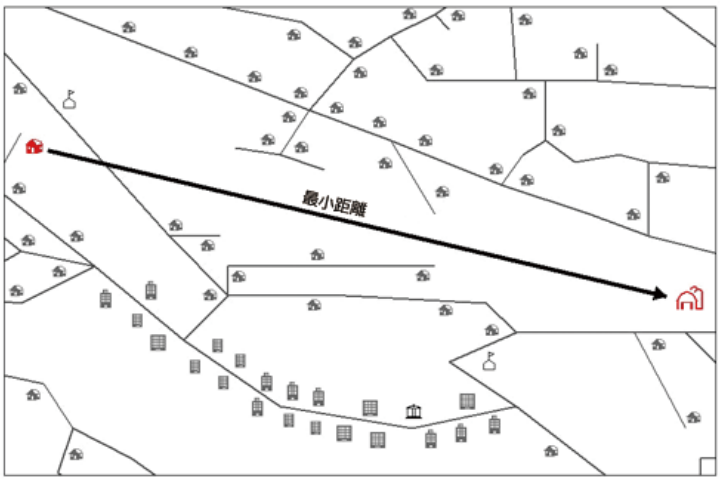
			ジオメトリ b	
		内部	**境界**	**外部**
ジオメトリ a	**内部**	T	*	F
	境界	*	*	F
	外部	*	*	*

ST_Within マトリックス

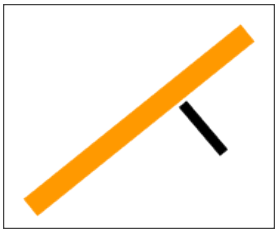
その他の空間リレーションシップ

次の関数ではジオメトリ間の空間リレーションシップが比較されますが、ジオメトリの内部、境界、外部だけでなく、その他の要素についても比較が行われます。

- ST_Distance 関数は、2 つの分断したジオメトリを入力として受け取り、この 2 つのジオメトリ間の最短距離を返します。 ジオメトリが分断していない (つまり、一致している) 場合、最短距離としてゼロが報告されます。離れたフィーチャの最短距離は、2 つの位置の間の最短距離を表します。 たとえば、この距離は、ある地点から別の地点まで運転する場合の移動距離ではなく、地図上の 2 つの位置間に直線を引いた場合に計測される距離です。



- ST_DWithin には、距離の値および比較するジオメトリを指定します。 ST_DWithin は、これらのジオメトリが指定した距離内にある場合に true を返します。
- ST_EnvIntersects 関数は、指定されたジオメトリの空間エンベロープが交差するかどうかを評価します。これに対し、ST_Intersects はジオメトリ自体が交差するかどうかを評価します。
  次の例では、2 つのラインのエンベロープは交差していますが、ライン自体は交差していません。



- ST_OrderingEquals 関数は ST_Equals による比較を拡張するものであり、ジオメトリの座標が同じ順序 (x,y または y,x) で定義されているかどうかも比較します。 ジオメトリが同じ空間を占有していても、X 座標と Y 座標が同じ順序で定義されていない場合、ST_OrderingEquals は false を返します。

空間処理

空間処理では、ジオメトリ関数を使用して入力として空間データを受け取り、データを解析して、出力データを生成します。この出力データは入力データに対して実行された解析によって導き出されます。

空間処理から取得可能な導出データには次のようなものがあります。

- 入力フィーチャの周囲のバッファーであるポリゴン
- ジオメトリのコレクションの解析結果である 1 つのフィーチャ
- 他のフィーチャと同じ物理的空間にないフィーチャの一部を判断するための比較結果である 1 つのフィーチャ
- 他のフィーチャと同じ物理的空間と交差するフィーチャの一部を特定するための比較結果である 1 つのフィーチャ
- 互いに同一の物理的空間にない両方の入力フィーチャの一部を構成するマルチパート フィーチャ
- 2 つのジオメトリの和 (ユニオン) であるフィーチャ

入力データの解析が実行されると、その結果として得られるジオメトリを表す座標またはテキストが返されます。この情報をより大きなクエリの一部として使用して追加的な解析を実行することも、この結果を別のテーブルの入力に使用することもできます。

たとえば、交差クエリの WHERE 句にバッファー演算を含めれば、指定したジオメトリが他のジオメトリの周囲の指定したサイズの領域に交差するかどうかを判断できます。

注意:
次の例では、ST_Geometry 関数を使用します。 他のデータベースと空間データ タイプに使用する個々のジオメトリ関数と構文については、該当するデータベースおよびデータ タイプのドキュメントをご参照ください。

以下の例では、道路閉鎖位置の 1,000 フィート以内にあるすべての物件の所有者に通知を送付する必要があります。 WHERE 句により、閉鎖される道路の周囲に 1,000 フィートのバッファーが生成されます。 次に、このバッファーが対象領域内の各物件と比較され、どれがバッファーと交差しているかが確認されます。

```
SELECT p.owner,p.address,s.stname
FROM parcels p, streets s
WHERE s.stname = 'Main'
AND sde.st_intersects (p.shape, sde.st_buffer (s.shape, 1000)) = 't';
```

この例では、WHERE 句で 1 つの特定の道路 (Main) が選択され、次に、その道路の周囲にバッファーが生成され、土地区画のテーブルのフィーチャと比較して交差があるかどうかが判断されます。* Main 道路上のバッファーと交差するすべての土地区画について、それらの土地区画の所有者名と住所が返されます。

注意:
*WHERE 句の各部分の実行順序は、データベース オプティマイザーによって決まります。

次の例では、地区と校区が格納されたテーブルについて空間処理 (ユニオン) を実施してその結果を取得し、結果として得られたフィーチャを別のテーブルに挿入します。

```
INSERT INTO combo c (shape)
VALUES (
(SELECT sde.st_union (n.shape,d.shape)
FROM neighborhoods n, school_districts d),5);
```

ST_Geometry での空間演算子の使用に関する詳細については、「ST_Geometry の空間処理関数」をご参照ください。

ST_Geometry の空間処理関数

空間処理では、ジオメトリ関数を使用して入力として空間データを受け取り、データを解析して、出力データを生成します。この出力データは入力データに対して実行された解析によって導き出されます。

次の各セクションで説明する処理を実行することで、入力フィーチャからフィーチャを作成できます。

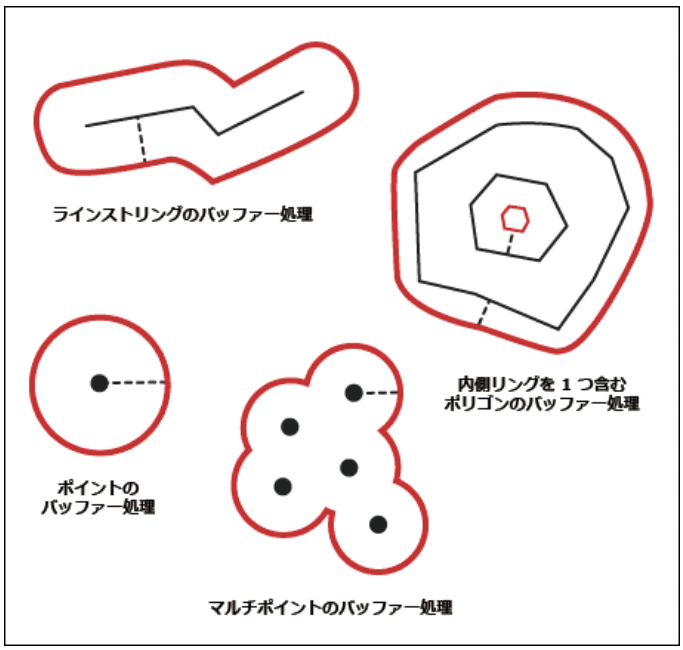
ジオメトリのバッファー

ST_Buffer 関数を実行すると、指定した距離でジオメトリを取り囲むジオメトリが生成されます。 1 つのジオメトリをバッファーしたとき、またはコレクションのバッファー ポリゴンが重なり合うのに十分なほど近接している場合には、ポリゴンが 1 つ生成されます。 バッファーされるコレクションのエレメント間が十分に離れているときには、それぞれのバッファー ST_Polygon について ST_MultiPolygon が生成されます。

ST_Buffer 関数では、正と負の距離が許容されますが、負の距離はディメンションが 2 次元のジオメトリ (ST_Polygon または ST_MultiPolygon) にしか適用できません。 ソース ジオメトリのディメンションが 2 次元よりも小さい場合、つまり、ST_Polygon でも ST_MultiPolygon でもないすべてのジオメトリでは、バッファー距離の絶対値が使用されます。 バッファー距離が正の場合は、ソース ジオメトリの中心から遠ざかるポリゴン リングが生成されます。ST_Polygon または ST_MultiPolygon の外部リングについては、距離が負の場合は中心に近づくように生成されます。 ST_Polygon または ST_MultiPolygon の内部リングについては、バッファー距離が正であればバッファー リングが中心に向かって生成され、負であれば中心から遠ざかるように生成されます。

バッファー処理では、重なり合うバッファー ポリゴンがマージされます。 負の距離がポリゴンの最大内幅の半分よりも大きい場合は、空のジオメトリになります。

次の図では、バッファーが赤色で描画されています。



凸包

ST_ConvexHull 関数は、3 つ以上の頂点を持ち凸を形成するジオメトリの凸包ポリゴンを返します。 ジオメトリの頂点が凸を形成しない場合、ST_ConvexHull は NULL を返します。 たとえば、2 つの頂点で構成される線上で ST_ConvexHull を使用すると、NULL が返されます。 同様に、ポイント フィーチャに ST_ConvexHull 処理を使用し

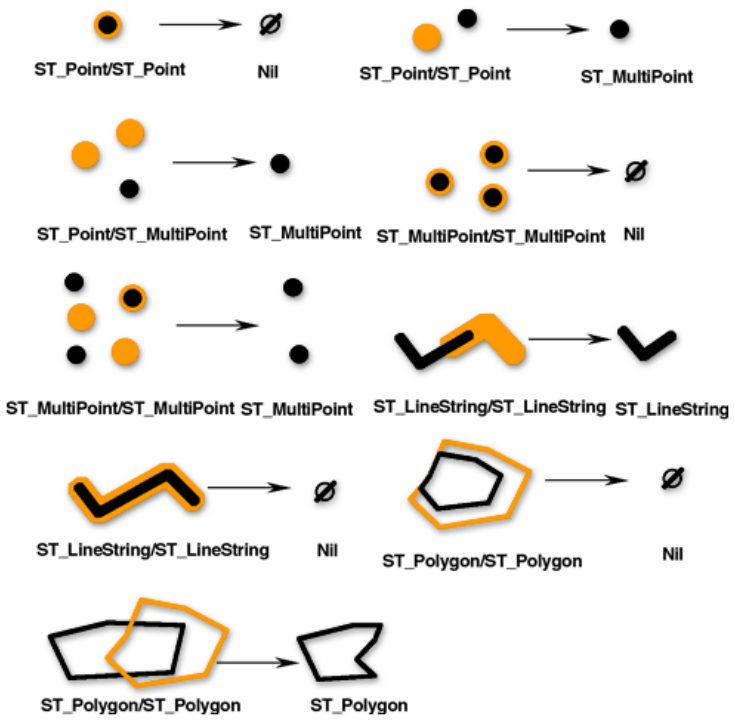
た場合にも、NULL が返されます。 一連のポイントを隣接させて TIN (Triangulated Irregular Network) を作成する場合には、通常、凸包の作成が最初のステップになります。

ジオメトリの違い

ST_Difference 関数を使用すると、第 2 のジオメトリが交差しない第 1 のジオメトリ部分が返されます。 これは空間の論理的 AND NOT です。

ST_Difference 関数は同じディメンションのジオメトリだけを対象とし、ソース ジオメトリと同じディメンションをもつコレクションを返します。 複数のソース ジオメトリが同等の場合は、空のジオメトリが返されます。

以下の図では、第 1 の入力ジオメトリが黒で、第 2 の入力ジオメトリがオレンジで記載されています。

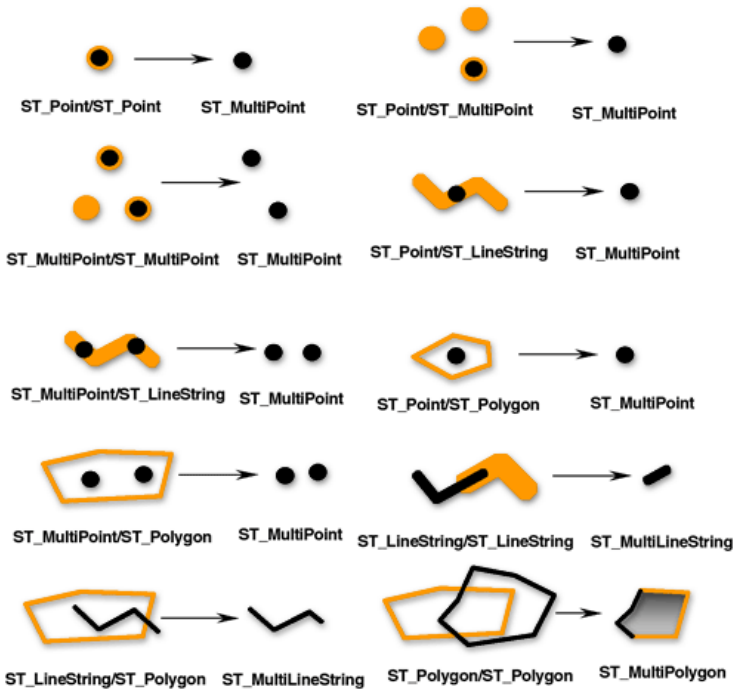


ジオメトリの交差

ST_Intersection 関数を使用すると、2 つのジオメトリの交差セットが返されます。 この交差セットは、必ず、ソース ジオメトリの最小ディメンションであるコレクションとして返されます。

たとえば、ST_Polygon に交差する ST_LineString に ST_Intersection 関数を使用すると、ST_Polygon の内部と境界に共通する ST_LineString の部分が ST_MultiLineString として返されます。 ソースの ST_LineString が 2 つ以上の不連続のセグメントで ST_Polygon に交差する場合には、ST_MultiLineString に 2 つ以上の ST_LineString が格納されます。 交差していないジオメトリの場合や、交差によって両方のソース ジオメトリよりも小さいディメンションとなる場合は、空のジオメトリが返されます。

以下に ST_Intersection 関数の例を図で説明します。 第 1 の入力ジオメトリが黒で、第 2 の入力ジオメトリがオレンジで記載されています。

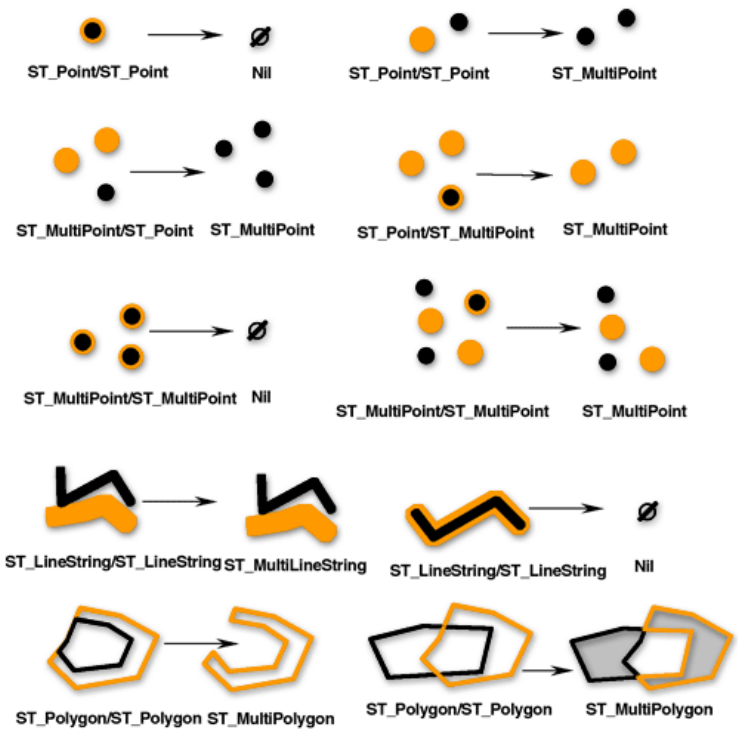


ジオメトリの対称差

ST_SymmetricDiff 関数は、交差セットの一部となっていないソース ジオメトリ部分を返します。 これは空間の論理 XOR です。

ソース ジオメトリは同じディメンションでなければなりません。 ST_SymmetricDiff 関数は、ジオメトリが等しければ空のジオメトリを返します。それ以外の場合、この関数はコレクションとして結果を返します。

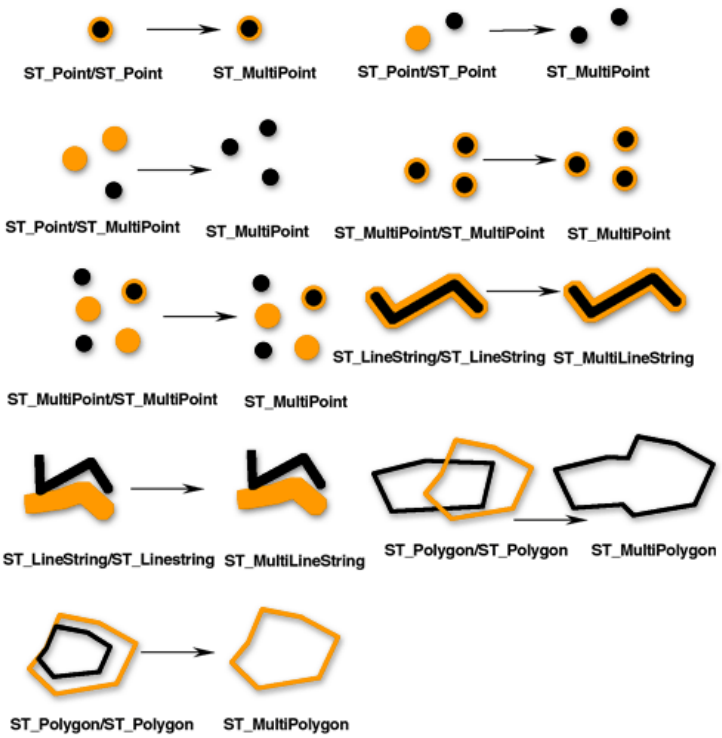
以下の図では、第 1 の入力ジオメトリが黒で、第 2 の入力ジオメトリがオレンジで記載されています。



ジオメトリの和 (ユニオン)

ST_Union 関数は、2 つのジオメトリのユニオン セットを返します。 これは空間の論理 OR です。 ソース ジオメトリは同じディメンションでなければなりません。 ST_Union では、結果が必ずコレクションとして返されます。

以下の図では、第 1 の入力ジオメトリが黒で、第 2 の入力ジオメトリがオレンジで記載されています。



集約

集約処理は、ジオメトリのコレクションに対して実行した解析結果として 1 つのジオメトリを返します。 ST_Aggr_ConvexHull 関数は、各入力ジオメトリの凸包ポリゴンで構成されるマルチポリゴンを返します。 頂点が 3 つに満たない入力ジオメトリには凸包がありません。 すべての入力ジオメトリで頂点が 3 つに満たない場合に ST_Aggr_ConvexHull を使用すると、NULL が返されます。

ST_Aggr_Intersection 関数は、すべての入力ジオメトリの交差の集約である 1 つのジオメトリを返します。

ST_Aggr_Intersection では複数のジオメトリの交差を特定できるのに対し、ST_Intersection では、2 つのジオメトリの間での交差しか特定できません。 たとえば、特定の校区、電話サービス、高速インターネット プロバイダーなど、いくつかの特定のサービスの対象となっており、特定の委員が代表者となっている物件を検索する場合は、それらすべての領域の交差を特定する必要があります。 それらの領域の 2 つだけについて交差を特定しても、必要なすべての情報を得ることはできないため、同じクエリですべての領域を評価できるよう ST_Aggr_Intersection 関数を使用します。

さらなる例として、2 つのフィーチャクラスのラインとポイントの交差を特定する場合、各関数によって以下が返されます。

- ST_Intersection - 各交差について ST_Point ジオメトリが 1 つ返されます。
- ST_Aggr_Intersection - すべての交差ポイントで構成される 1 つの ST_MultiPoint ジオメトリが返されます (1 つのポイント フィーチャと 1 つのライン フィーチャしか交差していない場合には、ST_Point ジオメトリが返されます)。

ST_Aggr_Union 関数を使用すると、提供したすべてのジオメトリの和 (ユニオン) である 1 つのジオメトリが返されます。

各入力ジオメトリはタイプが同じである必要があります。たとえば、ST_LineString と ST_LineString の和や、ST_Polygon と ST_Polygon の和を取得することはできますが、ST_LineString フィーチャクラスと ST_Polygon フィ

ーチャクラスの和を取得することはできません。

通常、集約ユニオンにより得られるジオメトリはコレクションです。 たとえば、空いているすべての 0.5 エーカー未満の土地区画の集約ユニオンを求める場合には、この条件を満たすすべての土地区画が隣接している場合を除き、ジオメトリとしてマルチポリゴンが返されます。すべての土地区画が隣接している場合は、1 つのポリゴンが返されます。

最短距離

これまでの関数は、新しいジオメトリを返すものでした。 ST_Distance 関数は空間処理を実行する (2 つのジオメトリ間の最短距離を評価する) ものであり、新しいジオメトリは返しません。

パラメトリックの円、楕円、および扇形

ST_Geometry 関数を使用し、ST_Geometry 列のパラメトリックの円、楕円、または扇形を作成および検索できます。

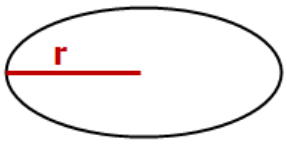
パラメトリックの円、楕円、および扇形は、座標値や角度、半径など、特定のパラメーターによって定義されるポリゴンです。データベースには、特定の頂点やラインの代わりに、このパラメーターが格納されます。シェープを定義するパラメーターを格納することで、パラメトリックのシェープは、ポリゴン表現として格納するよりも正確で格納領域が少なくなります。パラメトリックのシェープを使用すると、Z 座標とメジャー (M) 値をパラメーターとして含めることも可能です。

円の作成時には以下の 7 つのパラメーターが想定されます。

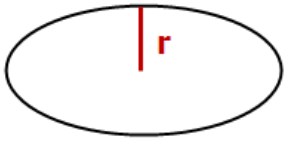
- 円の中心点の X 座標値
- 円の中心点の Y 座標値
- 円の中心点の Z 座標値
- M 値
- 作成する円の半径
- 円を定義するのに使用されるポイントの数
  指定できるポイントの数は最低でも 9 です。ポイントの数を指定しない場合は、デフォルト値として 50 が使用されます。これらのポイントはシェープと一緒に格納されませんが、円を生成してシェープを整合チェックするときに生成されます。
- 円を空間に配置するのに使用される空間参照 ID (SRID)

楕円の作成時には以下の 9 つのパラメーターが想定されます。

- 楕円の中心点の X 座標値
- 楕円の中心点の Y 座標値
- 楕円の中心点の Z 座標値
- M 値
- 楕円の赤道半径
  赤道半径は楕円の長半径です。赤道半径として指定する値は、極半径より大きくする必要があります。



- 楕円の極半径
  極半径は楕円の短半径です。極半径として指定する値は、0.0 より大きくする必要があります。



- 楕円の回転角度

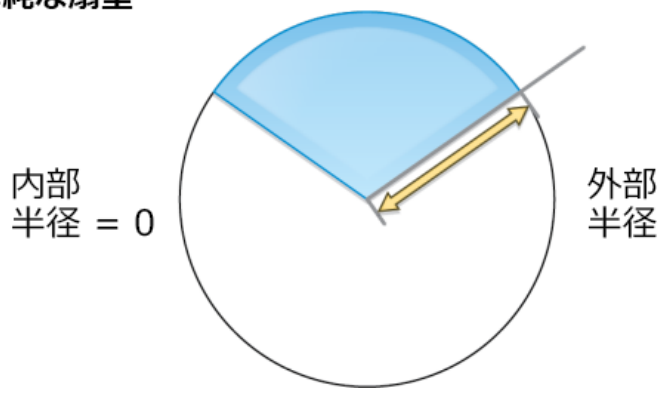
回転角度として指定する値は度単位で指定し、0.0 より大きく、360 より小さくする必要があります。回転の向きは時計回りです。

- 楕円を定義するのに使用されるポイントの数
  指定できるポイントの数は最低でも 9 です。ポイントの数を指定しない場合は、デフォルトで 50 ポイントが使用されます。これらのポイントはシェープと一緒に格納されませんが、楕円を生成してシェープを整合チェックするときに生成されます。
- 楕円を空間に配置するのに使用される SRID

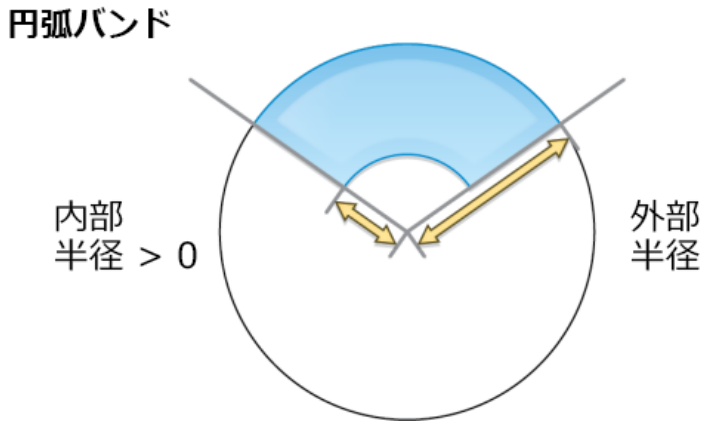
扇形の作成時には以下の 10 のパラメーターが想定されます。

- 扇形を定義する円の中心点の X 座標値
- 扇形を定義する円の中心点の Y 座標値
- 扇形を定義する円の中心点の Z 座標値
- M 値
- 扇形の開始角度
  開始角度は扇形の始点を定義します。0 度から反時計回りに計測した角度で表します。
- 扇形の終了角度
  終了角度は扇形の終点を定義します。0 度から反時計回りに計測した角度で表します。
- 外径
  外径は、円の中心から扇形の最も外側の点までの距離を定義します。
- 内径
  内径は、円の中心から扇形の最も内側の点までの距離を定義します。したがって、扇形の開始位置を定義します。内径が 0 の場合は単純な扇形になります。

単純な扇型



内径が 0 より大きい場合、扇形は厳密には厚みを持つ円弧になります。



- 扇形を定義するのに使用されるポイントの数
  指定できるポイントの数は最低でも 9 です。ポイントの数を指定しない場合は、デフォルトで 80 ポイントが使用されます。これらのポイントはシェープと一緒に格納されませんが、扇形を生成してシェープを整合チェックするときに生成されます。
- 扇形を空間に配置するのに使用される SRID

赤道半径と極半径および外形と内径を含むすべての半径は、SRID で指定された座標参照によって決まる単位によって定義されます。

パラメトリックの円、楕円、または扇形を作成する構文と例については、ST_Geometry 関数をご参照ください。

ST_Aggr_ConvexHull

注意:
Oracle と SQLite のみ

定義

ST_Aggr_ConvexHull は、すべての入力ジオメトリのユニオンから得られる凸包のジオメトリを 1 つ作成します。実際には、ST_Aggr_ConvexHull は ST_ConvexHull(ST_Aggr_Union(ジオメトリ)) に相当します。

構文

Oracle

```
sde.st_aggr_convexhull (geometry sde.st_geometry)
```

SQLite

```
st_aggr_convexhull (geometry st_geometry)
```

戻り値のタイプ

Oracle

ST_Geometry

SQLite

Geometryblob

例

この例では、service_territories テーブルを作成し、すべてのジオメトリを集約する SELECT ステートメントを実行して、すべてのシェープのユニオンから得られる凸包を表す単一のジオメトリが生成されます。

Oracle

```
CREATE TABLE service_territories
 (ID integer not null, UNITS number, SHAPE sde.st_geometry);

INSERT INTO service_territories (id, units, shape) VALUES (
 1,
 1250,
 sde.st_polygon ('polygon ((20 30, 30 30, 30 40, 20 40, 20 30))', 4326)
);

INSERT INTO service_territories (id, units, shape) VALUES (
 2,
 875,
 sde.st_polygon ('polygon ((30 30, 30 50, 50 50, 50 30, 30 30))', 4326)
);
```

```
INSERT INTO service_territories (id, units, shape) VALUES (
 3,
 1700,
 sde.st_polygon ('polygon ((40 40, 40 60, 60 60, 60 40, 40 40))', 4326)
);

SELECT sde.st_astext(sde.st_aggr_convexhull(shape)) CONVEX_HULL
 FROM service_territories
 WHERE units >= 1000;

CONVEX_HULL

POLYGON  (( 20.00000000 40.00000000, 20.00000000 30.00000000, 30.00000000 30.00000000,
60.00000000 40.00000000, 60.00000000 60.00000000, 40.00000000 60.00000000, 20.00000000
40.00000000))
```

SQLite

```
CREATE TABLE service_territories (
  ID integer primary key autoincrement not null,
  UNITS numeric
);

SELECT AddGeometryColumn(
 NULL,
 'service_territories',
 'shape',
 4326,
 'polygon',
 'xy',
 'null'
);

INSERT INTO service_territories (units, shape) VALUES (
 1250,
 st_polygon ('polygon ((20 30, 30 30, 30 40, 20 40, 20 30))', 4326)
);

INSERT INTO service_territories (units, shape) VALUES (
 875,
 st_polygon ('polygon ((30 30, 30 50, 50 50, 50 30, 30 30))', 4326)
);

INSERT INTO service_territories (units, shape) VALUES (
 1700,
 st_polygon ('polygon ((40 40, 40 60, 60 60, 60 40, 40 40))', 4326)
);

SELECT st_astext(st_aggr_convexhull(shape)) AS "CONVEX HULL"
 FROM service_territories
 WHERE units >= 1000;

CONVEX HULL

POLYGON  (( 20.00000000 40.00000000, 20.00000000 30.00000000, 30.00000000 30.00000000,
60.00000000 40.00000000, 60.00000000 60.00000000, 40.00000000 60.00000000, 20.00000000
40.00000000))
```

ST_Aggr_Intersection

注意:
Oracle と SQLite のみ

定義

ST_Aggr_Intersection は、すべての入力ジオメトリのインターセクト (交差) がユニオン (結合) された単一のジオメトリを返します。

構文

Oracle

```
sde.st_aggr_intersection (geometry1 sde.st_geometry)
```

SQLite

```
st_aggr_intersection (geometry1 geometryblob)
```

戻り値のタイプ

Oracle

ST_Geometry

SQLite

Geometryblob

例

この例では、生物学者が 3 つの野生生物生息地のインターセクトを見つけようとしています。

Oracle

最初に、生息地を格納するテーブルを作成します。

```
CREATE TABLE habitats (
 id integer not null,
 shape sde.st_geometry
);
```

次に、テーブルに 3 つのポリゴンを挿入します。

```
INSERT INTO habitats (id, shape) VALUES (
 1,
 sde.st_polygon ('polygon ((5 5, 12 5, 12 10, 5 10, 5 5))', 4326)
);
```

```
INSERT INTO habitats (id, shape) VALUES (
 2,
 sde.st_polygon ('polygon ((10 8, 14 8, 14 15, 10 15, 10 8))', 4326)
);

INSERT INTO habitats (id, shape) VALUES (
 3,
 sde.st_polygon ('polygon ((6 8, 20 8, 20 20, 6 20, 6 8))', 4326)
);
```

最後に、生息地のインターセクトを選択します。

```
SELECT sde.st_astext(sde.st_aggr_intersection(shape)) AGGR_SHAPES
 FROM habitats;

AGGR_SHAPES

POLYGON  (( 10.00000000 8.00000000, 12.00000000 8.00000000, 12.00000000 10.00000000,
10.00000000 10.00000000, 10.00000000 8.00000000))
```

SQLite

最初に、生息地を格納するテーブルを作成します。

```
CREATE TABLE habitats (
  id integer primary key autoincrement not null
);

SELECT AddGeometryColumn(
 NULL,
 'habitats',
 'shape',
 4326,
 'polygon',
 'xy',
 'null'
);
```

次に、テーブルに 3 つのポリゴンを挿入します。

```
INSERT INTO habitats (shape) VALUES (
 st_polygon ('polygon ((5 5, 12 5, 12 10, 5 10, 5 5))', 4326)
);

INSERT INTO habitats (shape) VALUES (
 st_polygon ('polygon ((10 8, 14 8, 14 15, 10 15, 10 8))', 4326)
);

INSERT INTO habitats (shape) VALUES (
 st_polygon ('polygon ((6 8, 20 8, 20 20, 6 20, 6 8))', 4326)
);
```

最後に、生息地のインターセクトを選択します。

```
SELECT st_astext(st_aggr_intersection(shape))
```

```
  AS "AGGR_SHAPES"
  FROM habitats;

AGGR_SHAPES

POLYGON  (( 10.00000000 8.00000000, 12.00000000 8.00000000, 12.00000000 10.00000000,
10.00000000 10.00000000, 10.00000000 8.00000000))
```

ST_Aggr_Union

定義

ST_Aggr_Union は、すべての入力ジオメトリがユニオン (結合) された単一のジオメトリを返します。

構文

Oracle および PostgreSQL

```
sde.st_aggr_union(geometry sde.st_geometry)
```

SQLite

```
st_aggr_union(geometry geometryblob)
```

戻り値のタイプ

Oracle および PostgreSQL

ST_Geometry

SQLite

Geometryblob

例

マーケティング アナリストは、売り上げが 1,000 ユニットを超えるすべての到達圏からなるジオメトリを 1 つ作成する必要があります。この例では、service_territories1 テーブルを作成し、売り上げ数の値を入力しています。SELECT ステートメントで st_aggr_union を使用して、売り上げ数が 1,000 ユニット以上であるすべてのジオメトリがユニオン (結合) されたマルチポリゴンを返します。

Oracle および PostgreSQL

```
--Create and populate tables.
CREATE TABLE service_territories1 (
 ID integer not null,
 UNITS number,
 SHAPE sde.st_geometry);
INSERT INTO service_territories1 (id, units, shape) VALUES (
 1,
 1250,
 sde.st_polygon ('polygon ((20 30, 30 30, 30 40, 20 40, 20 30))', 4326)
);

INSERT INTO service_territories1 (id, units, shape) VALUES (
 2,
 875,
 sde.st_polygon ('polygon ((30 30, 30 50, 50 50, 50 30, 30 30))', 4326)
);

INSERT INTO service_territories1 (id, units, shape) VALUES (
```

```
 3,
 1700,
 sde.st_polygon ('polygon ((40 40, 40 60, 60 60, 60 40, 40 40))', 4326)
);
```

```
--Union of all geometries for which sales numbers are >= 1,000 units.
SELECT sde.st_astext(sde.st_aggr_union(shape)) UNION_SHAPE
  FROM service_territories1
  WHERE units >= 1000;
UNION_SHAPE
MULTIPOLYGON  ((( 20.00000000 30.00000000, 30.00000000 30.00000000, 30.00000000
40.00000000, 20.00000000 40.00000000, 20.00000000 30.00000000)),(( 40.00000000
40.00000000,
60.00000000 40.00000000, 60.00000000 60.00000000, 40.00000000 60.00000000,
40.00000000 40.00000000)))
```

SQLite

```
--Create table, add geometry column to it, and populate table.
CREATE TABLE service_territories1 (
 id integer primary key autoincrement not null,
 units number
);
SELECT AddGeometryColumn(
 NULL,
 'service_territories1',
 'shape',
 4326,
 'polygon',
 'xy',
 'null'
);
INSERT INTO service_territories1 (units, shape) VALUES (
 1250,
 st_polygon ('polygon ((20 30, 30 30, 30 40, 20 40, 20 30))', 4326)
);

INSERT INTO service_territories1 (units, shape) VALUES (
 875,
 st_polygon ('polygon ((30 30, 30 50, 50 50, 50 30, 30 30))', 4326)
);

INSERT INTO service_territories1 (units, shape) VALUES (
 1700,
 st_polygon ('polygon ((40 40, 40 60, 60 60, 60 40, 40 40))', 4326)
);
```

```
--Union of all geometries for which sales numbers are >= 1,000 units.
SELECT st_astext(st_aggr_union(shape))
 AS "UNION_SHAPE"
 FROM service_territories1
 WHERE units >= 1000;
UNION_SHAPE
MULTIPOLYGON ((( 40.00000000 40.00000000, 60.00000000 40.00000000, 60.00000000 6
0.00000000, 40.00000000 60.00000000, 40.00000000 40.00000000)),(( 20.00000000 30
.00000000, 30.00000000 30.00000000, 30.00000000 40.00000000, 20.00000000 40.0000
0000, 20.00000000 30.00000000)))
```

ST_Area

定義

ST_Area は、ポリゴンまたはマルチポリゴンの面積を返します。

構文

Oracle および PostgreSQL

```
sde.st_area (polygon sde.st_geometry)
sde.st_area (multipolygon sde.st_geometry)
```

SQLite

```
st_area (polygon st_geometry)
st_area (polygon st_geometry, unit_name)
```

戻り値のタイプ

Double precision

例

都市エンジニアは、建物の面積のリストを必要としています。リストを作成するために、GIS 技術者は建物 ID と各建物の面積を選択します。

建物フットプリントは bfp テーブルに保存されます。

都市エンジニアの要求に応えるために、技術者は building_id という一意のキーを選択して、bfp テーブルから各建物の面積を取得します。

Oracle

```
--Create and populate table.
CREATE TABLE bfp (
 building_id integer not null,
 footprint sde.st_geometry);

INSERT INTO BFP (building_id, footprint) VALUES (
 1,
 sde.st_polygon ('polygon ((0 0, 0 10, 10 10, 10 0, 0 0))', 4326)
);

INSERT INTO BFP (building_id, footprint) VALUES (
 2,
 sde.st_polygon ('polygon ((20 0, 30 20, 40 0, 20 0))', 4326)
);

INSERT INTO BFP (building_id, footprint) VALUES (
 3,
 sde.st_polygon ('polygon ((20 30, 25 35, 30 30, 20 30))', 4326)
);
```

```
--Get area of geometries.
SELECT building_id, sde.st_area (footprint) Area
 FROM BFP;

BUILDING_ID       Area

          1        100
          2        200
          3         25
```

PostgreSQL

```
--Create and populate table.
CREATE TABLE bfp (
 building_id serial,
 footprint sde.st_geometry);

INSERT INTO bfp (footprint) VALUES (
 sde.st_polygon ('polygon ((0 0, 0 10, 10 10, 10 0, 0 0))', 4326)
);

INSERT INTO bfp (footprint) VALUES (
 sde.st_polygon ('polygon ((20 0, 30 20, 40 0, 20 0))', 4326)
);

INSERT INTO bfp (footprint) VALUES (
 sde.st_polygon ('polygon ((20 30, 25 35, 30 30, 20 30))', 4326)
);
```

```
--Get area of geometries.
SELECT building_id, sde.st_area (footprint)
 AS Area
 FROM bfp;

building_id       area
```

```
          1        100
          2        200
          3         25
```

SQLite

```
--Create table, add geometry column to it, and populate the table.
CREATE TABLE bfp (
 building_id integer primary key autoincrement not null
);

SELECT AddGeometryColumn(
 NULL,
 'bfp',
 'footprint',
 4326,
 'polygon',
 'xy',
 'null'
);

INSERT INTO bfp (footprint) VALUES (
 st_polygon ('polygon ((0 0, 0 10, 10 10, 10 0, 0 0))', 4326)
);

INSERT INTO bfp (footprint) VALUES (
 st_polygon ('polygon ((20 0, 30 20, 40 0, 20 0))', 4326)
);

INSERT INTO bfp (footprint) VALUES (
 st_polygon ('polygon ((20 30, 25 35, 30 30, 20 30))', 4326)
);
```

```
--Get area of geometries.
SELECT building_id, st_area (footprint)
 AS "area"
 FROM bfp;

building_id       area

          1        100.0
          2        200.0
          3         25.0
```

ST_AsBinary

定義

ST_AsBinary は、ジオメトリ オブジェクトを入力として、その WKB 表現を返します。

構文

Oracle および PostgreSQL

```
sde.st_asbinary (geometry sde.st_geometry)
```

SQLite

```
st_asbinary (geometry geometryblob)
```

戻り値のタイプ

Oracle および PostgreSQL

ST_Geometry

SQLite

Geometryblob

例

この例では、レコード 1111 の WKB 列に、レコード 1100 の GEOMETRY 列の内容を入力しています。

Oracle

```
CREATE TABLE sample_points (
 id integer not null,
 geometry sde.st_geometry,
 wkb blob
);

INSERT INTO SAMPLE_POINTS (id, geometry) VALUES (
 1100,
 sde.st_geometry ('point (10 20)', 4326)
);

INSERT INTO SAMPLE_POINTS (id, wkb) VALUES (
 1111,
 (SELECT sde.st_asbinary (geometry) FROM sample_points WHERE id = 1100)
);

SELECT id, sde.st_astext (sde.st_geomfromwkb (wkb, 4326))
 FROM SAMPLE_POINTS
 WHERE id = 1111;

ID              Point
1111     POINT (10.00000000 20.00000000)
```

PostgreSQL

```
CREATE TABLE sample_points (
 id serial,
 geometry sde.st_geometry,
 wkb bytea);

INSERT INTO sample_points (geometry) VALUES (
 sde.st_point (10, 20, 4326)
);

INSERT INTO sample_points (wkb) VALUES (
 (SELECT sde.st_asbinary (geometry) FROM sample_points WHERE id = 1100)
);

SELECT id, sde.st_astext (sde.st_geomfromwkb (wkb, 4326))
 FROM sample_points
 WHERE id = 1111;

ID              st_astext
1111     POINT (10 20)
```

SQLite

```
CREATE TABLE sample_points (
 id integer primary key autoincrement not null,
 wkb blob
);

SELECT AddGeometryColumn(
 NULL,
 'sample_points',
 'geometry',
 4326,
 'point',
 'xy',
 'null'
);

INSERT INTO sample_points (geometry) VALUES (
 st_point (10, 20, 4326)
);

INSERT INTO sample_points (wkb) VALUES (
 (SELECT st_asbinary (geometry) FROM sample_points WHERE id = 1)
);

SELECT id, st_astext (st_geomfromwkb (wkb, 4326))
 FROM sample_points
 WHERE id = 2;

ID              st_astext
2        POINT (10.00000000 20.00000000)
```

ST_AsText

定義

ST_AsText は、ジオメトリを入力として、その WKT 表現を返します。

構文

Oracle および PostgreSQL

```
sde.st_astext (geometry sde.st_geometry)
```

SQLite

```
st_astext (geometry geometryblob)
```

戻り値のタイプ

Oracle

CLOB

PostgreSQL および SQLite

Text

例

ST_AsText 関数は、hazardous_sites のロケーション ポイントをテキスト表現に変換します。

Oracle

```
CREATE TABLE hazardous_sites (
 site_id integer not null,
 name varchar(40),
 loc sde.st_geometry);

INSERT INTO HAZARDOUS_SITES (site_id, name, loc) VALUES (
 102,
 'W. H. KleenareChemical Repository',
 sde.st_geometry ('point (1020.12 324.02)', 4326)
);

SELECT site_id, name, sde.st_astext (loc) Location
 FROM HAZARDOUS_SITES;

SITE_ID      NAME                              Location

   102    W. H. KleenareChemical Repository   POINT (1020.12000000 324.02000000)
```

PostgreSQL

```
CREATE TABLE hazardous_sites (
 site_id serial,
 name varchar(40),
 loc sde.st_geometry);

INSERT INTO hazardous_sites (name, loc) VALUES (
 'W. H. KleenareChemical Repository',
 sde.st_point ('point (1020.12 324.02)', 4326)
);

SELECT site_id, name, sde.st_astext (loc)
 AS location
 FROM hazardous_sites;

site_id      name                              location

   102    W. H. KleenareChemical Repository   POINT (1020.12000001 324.01999999)
```

SQLite

```
CREATE TABLE hazardous_sites (
 site_id integer primary key autoincrement not null,
 name varchar(40)
);

SELECT AddGeometryColumn(
 NULL,
 'hazardous_sites',
 'loc',
 4326,
 'point',
 'xy',
 'null'
);

INSERT INTO hazardous_sites (name, loc) VALUES (
 'W. H. KleenareChemical Repository',
 st_point ('point (1020.12 324.02)', 4326)
);

SELECT site_id, name, st_astext (loc)
 FROM hazardous_sites;

   1    W. H. KleenareChemical Repository   POINT (1020.12000000 324.02000000)
```

ST_Boundary

定義

ST_Boundary はジオメトリを入力として、結合した境界をジオメトリ オブジェクトとして返します。

構文

Oracle および PostgreSQL

```
sde.st_boundary (geometry sde.st_geometry)
```

SQLite

```
st_boundary (geometry geometryblob)
```

戻り値のタイプ

Oracle および PostgreSQL

ST_Geometry

SQLite

Geometryblob

例

この例では、タイプとジオメトリの 2 つの列を持つ boundaries テーブルが作成されています。 後続の INSERT ステートメントは、各サブクラス ジオメトリに対して 1 つのレコードを追加します。 ST_Boundary 関数は、ジオメトリ列に格納された各サブクラスの境界を取得します。 結果として生成されるジオメトリのディメンションは、入力ジオメトリよりも常に 1 小さくなります。 ポイントとマルチポイントは、常に空のジオメトリである境界 (ディメンション -1) になります。 ラインストリングとマルチストリングは、マルチポイント境界 (ディメンション 0) を返します。 ポリゴンまたはマルチポリゴンは、常にマルチラインストリング境界 (ディメンション 1) を返します。

Oracle

```
CREATE TABLE boundaries (
 geotype varchar(20),
 geometry sde.st_geometry
);

INSERT INTO BOUNDARIES VALUES (
 'Point',
 sde.st_pointfromtext ('point (10.02 20.01)', 4326)
);

INSERT INTO BOUNDARIES VALUES (
 'Linestring',
 sde.st_linefromtext ('linestring (10.02 20.01, 10.32 23.98, 11.92 25.64)', 4326)
);
```

```
INSERT INTO BOUNDARIES VALUES (
 'Polygon',
 sde.st_polyfromtext ('polygon ((10.02 20.01, 11.92 35.64, 25.02 34.15, 19.15 33.94,
10.02 20.01))', 4326)
);

INSERT INTO BOUNDARIES VALUES (
 'Multipoint',
 sde.st_mpointfromtext ('multipoint ((10.02 20.01), (10.32 23.98), (11.92 25.64))',
4326)
);

INSERT INTO BOUNDARIES VALUES (
 'Multilinestring',
 sde.st_mlinefromtext ('multilinestring ((10.02 20.01, 10.32 23.98, 11.92 25.64), (9.55
23.75, 15.36 30.11))', 0)
);

INSERT INTO BOUNDARIES VALUES (
 'Multipolygon',
 sde.st_mpolyfromtext ('multipolygon (((10.02 20.01, 11.92 35.64, 25.02 34.15, 19.15
33.94, 10.02 20.01), (51.71 21.73,73.36 27.04,71.52 32.87, 52.43 31.90, 51.71
21.73)))', 4326)
);

SELECT geotype, sde.st_astext (sde.st_boundary (geometry)) "The boundary"
 FROM BOUNDARIES;

GEOTYPE            The boundary

Point              POINT EMPTY
Linestring         MULTIPOINT((10.02000000 20.01000000), (11.92000000 25.64000000))
Polygon            MULTILINESTRING ((10.02000000 20.01000000, 19.15000000
33.94000000,25.02000000 34.15000000, 11.92000000 35.64000000, 10.02000000 20.01000000))
Multipoint         POINT EMPTY
Multilinestring    MULTIPOINT ((9.55000000 23.75000000), (10.02000000 20.01000000),
(11.92000000 25.64000000), (15.36000000 30.11000000))
Multipolygon       MULTILINESTRING((51.71000000 21.73000000, 73.36000000 27.04000000,
71.52000000 32.87000000, 52.43000000 31.90000000, 51.71000000 21.73000000),
(10.02000000 20.01000000, 19.15000000 33.94000000, 25.02000000 34.15000000, 11.92000000
35.64000000, 10.02000000 20.01000000))
```

PostgreSQL

```
CREATE TABLE boundaries (
 geotype varchar(20),
 geometry st_geometry
);

INSERT INTO boundaries VALUES (
 'Point',
 st_point ('point (10.02 20.01)', 4326)
);

INSERT INTO boundaries VALUES (
 'Linestring',
 st_linestring ('linestring (10.02 20.01, 10.32 23.98, 11.92 25.64)', 4326)
);

INSERT INTO boundaries VALUES (
```

```
 'Polygon',
 st_polygon ('polygon ((10.02 20.01, 11.92 35.64, 25.02 34.15, 19.15 33.94, 10.02
20.01))', 4326)
);

INSERT INTO boundaries VALUES (
'Multipoint',
st_multipoint ('multipoint (10.02 20.01, 10.32 23.98, 11.92 25.64)', 0)
);

INSERT INTO boundaries VALUES (
 'Multilinestring',
 st_multilinestring ('multilinestring ((10.02 20.01, 10.32 23.98, 11.92 25.64), (9.55
23.75, 15.36 30.11))', 4326)
);

INSERT INTO boundaries VALUES (
 'Multipolygon',
 st_multipolygon ('multipolygon (((10.02 20.01, 11.92 35.64, 25.02 34.15, 19.15 33.94,
10.02 20.01),
(51.71 21.73, 73.36 27.04, 71.52 32.87, 52.43 31.90, 51.71 21.73)))', 4326)
);

SELECT geotype, st_astext (st_boundary (geometry))
 AS "The boundary"
 FROM boundaries;

geotype            The boundary

Point              EMPTY
Linestring         MULTIPOINT(10.02000000 20.01000000, 11.92000000 25.64000000)
Polygon            LINESTRING ((10.02000000 20.01000000, 19.15000000
33.94000000,25.02000000 34.15000000, 11.92000000 35.64000000, 10.02000000
20.01000000))
Multipoint         EMPTY
Multilinestring    MULTIPOINT (9.55000000 23.75000000, 10.02000000 20.01000000,
11.92000000 25.64000000, 15.36000000 30.11000000)
Multipolygon       MULTILINESTRING((51.71000000 21.73000000, 73.36000000 27.04000000,
71.52000000 32.87000000, 52.43000000 31.90000000,
51.71000000 21.73000000), (10.02000000 20.01000000, 19.15000000 33.94000000,
25.02000000 34.15000000, 11.92000000 35.64000000,
10.02000000 20.01000000))
```

SQLite

```
CREATE TABLE boundaries (
 geotype varchar(20)
);

SELECT AddGeometryColumn (
 NULL,
 'boundaries',
 'geometry',
 4326,
 'geometry',
 'xy',
 'null'
);

INSERT INTO boundaries VALUES (
 'Point',
```

```
 st_point ('point (10.02 20.01)', 4326)
);

INSERT INTO boundaries VALUES (
 'Linestring',
 st_linestring ('linestring (10.02 20.01, 10.32 23.98, 11.92 25.64)', 4326)
);

INSERT INTO boundaries VALUES (
 'Polygon',
 st_polygon ('polygon ((10.02 20.01, 11.92 35.64, 25.02 34.15, 19.15 33.94, 10.02
20.01))', 4326)
);

INSERT INTO boundaries VALUES (
 'Multipoint',
 st_multipoint ('multipoint ((10.02 20.01), (10.32 23.98), (11.92 25.64))', 4326)
);

INSERT INTO boundaries VALUES (
 'Multilinestring',
 st_multilinestring ('multilinestring ((10.02 20.01, 10.32 23.98, 11.92 25.64), (9.55
23.75, 15.36 30.11))', 4326)
);

INSERT INTO boundaries VALUES (
 'Multipolygon',
 st_multipolygon ('multipolygon (((10.02 20.01, 11.92 35.64, 25.02 34.15, 19.15 33.94,
10.02 20.01),
(51.71 21.73, 73.36 27.04, 71.52 32.87, 52.43 31.90, 51.71 21.73)))', 4326)
);

SELECT geotype, st_astext (st_boundary (geometry))
 FROM boundaries;

Point              EMPTY
Linestring         MULTIPOINT((10.02000000 20.01000000), (11.92000000 25.64000000))
Polygon            LINESTRING ((10.02000000 20.01000000, 19.15000000
33.94000000,25.02000000 34.15000000, 11.92000000 35.64000000, 10.02000000
20.01000000))
Multipoint         EMPTY
Multilinestring    MULTIPOINT ((9.55000000 23.75000000), (10.02000000 20.01000000),
(11.92000000 25.64000000), (15.36000000 30.11000000))
Multipolygon       MULTILINESTRING((51.71000000 21.73000000, 73.36000000 27.04000000,
71.52000000 32.87000000, 52.43000000 31.90000000,
51.71000000 21.73000000), (10.02000000 20.01000000, 19.15000000 33.94000000,
25.02000000 34.15000000, 11.92000000 35.64000000,
10.02000000 20.01000000))
```

ST_Buffer

定義

ST_Buffer は、ジオメトリ オブジェクトと距離を入力として、ソース オブジェクトの周囲にバッファーを作成したジオメトリ オブジェクトを返します。

構文

Unit_name は、バッファー距離の計測単位 (たとえば、メートル、キロメートル、フィート、マイル) です。Projected coordinate system tables.pdf の最初の表をご参照ください。これには「座標系、投影法、座標変換」からアクセスできます。

Oracle

```
sde.st_buffer (geometry sde.st_geometry, distance double_precision)
sde.st_buffer (geometry sde.st_geometry, distance double, varchar2 unit_name)
```

PostgreSQL

```
sde.st_buffer (geometry sde.st_geometry, distance double_precision)
sde.st_buffer (geometry sde.st_geometry, distance double, text unit_name)
```

SQLite

```
st_buffer (geometry geometryblob, distance double_precision)
st_buffer (geometry geometryblob, distance double, text unit_name)
```

戻り値のタイプ

Oracle および PostgreSQL

ST_Geometry

SQLite

Geometryblob

例

この例では、sensitive_areas と hazardous_sites という 2 つのテーブルを作成します。テーブルに値を入力し、ST_Buffer を使用して hazardous_sites テーブル内のポリゴンの周囲にバッファーを作成し、これらのバッファーが sensitive_areas ポリゴンと重なる領域を調べます。

Oracle

```
CREATE TABLE sensitive_areas (
 id integer,
 zone sde.st_geometry
);
```

```
CREATE TABLE hazardous_sites (
 site_id integer,
 name varchar(40),
 location sde.st_geometry
);

INSERT INTO SENSITIVE_AREAS VALUES (
 1,
 sde.st_polygon ('polygon ((20 30, 30 30, 30 40, 20 40, 20 30))', 4326)
);

INSERT INTO SENSITIVE_AREAS VALUES (
 2,
 sde.st_polygon ('polygon ((30 30, 30 50, 50 50, 50 30, 30 30))', 4326)
);

INSERT INTO SENSITIVE_AREAS VALUES (
3,
sde.st_polygon ('polygon ((40 40, 40 60, 60 60, 60 40, 40 40))', 0)
);

INSERT INTO HAZARDOUS_SITES VALUES (
 102,
 'W. H. KleenareChemical Repository',
 sde.st_pointfromtext ('point (60 60)', 4326)
);

SELECT sa.id "Sensitive Areas", hs.name "Hazardous Sites"
 FROM SENSITIVE_AREAS sa, HAZARDOUS_SITES hs
 WHERE sde.st_overlaps (sa.zone, sde.st_buffer (hs.location, .01)) = 1;
```

PostgreSQL

```
CREATE TABLE sensitive_areas (
 id serial,
 zone sde.st_geometry
);

CREATE TABLE hazardous_sites (
 site_id serial,
 name varchar(40),
 location sde.st_geometry
);

INSERT INTO sensitive_areas (zone) VALUES (
 sde.st_polygon ('polygon ((20 30, 30 30, 30 40, 20 40, 20 30))', 4326)
);

INSERT INTO sensitive_areas (zone) VALUES (
 sde.st_polygon ('polygon ((30 30, 30 50, 50 50, 50 30, 30 30))', 4326)
);

INSERT INTO sensitive_areas (zone) VALUES (
 sde.st_polygon ('polygon ((40 40, 40 60, 60 60, 60 40, 40 40))', 4326)
);

INSERT INTO hazardous_sites (name, location) VALUES (
 'W. H. KleenareChemical Repository',
 sde.st_point ('point (60 60)', 4326)
);
```

```
SELECT sa.id AS "Sensitive Areas", hs.name AS "Hazardous Sites"
 FROM sensitive_areas sa, hazardous_sites hs
 WHERE sde.st_overlaps (sa.zone, sde.st_buffer (hs.location, .01)) = 't';

Sensitive Areas               Hazardous Sites
             3               W.H. KleenareChemical Repository
```

SQLite

```
CREATE TABLE sensitive_areas (
 id integer primary key autoincrement not null
);

SELECT AddGeometryColumn (
 NULL,
 'sensitive_areas',
 'zone',
 4326,
 'polygon',
 'xy',
 'null'
);

CREATE TABLE hazardous_sites (
 site_id integer primary key autoincrement not null,
 name varchar(40)
);

SELECT AddGeometryColumn (
 NULL,
 'hazardous_sites',
 'location',
 4326,
 'point',
 'xy',
 'null'
);

INSERT INTO sensitive_areas (zone) VALUES (
 st_polygon ('polygon ((20 30, 30 30, 30 40, 20 40, 20 30))', 4326)
);

INSERT INTO sensitive_areas (zone) VALUES (
 st_polygon ('polygon ((30 30, 30 50, 50 50, 50 30, 30 30))', 4326)
);

INSERT INTO sensitive_areas (zone) VALUES (
 st_polygon ('polygon ((40 40, 40 60, 60 60, 60 40, 40 40))', 4326)
);

INSERT INTO hazardous_sites (name, location) VALUES (
 'W. H. KleenareChemical Repository',
 st_point ('point (60 60)', 4326)
);

SELECT sa.id AS "Sensitive Areas", hs.name AS "Hazardous Sites"
 FROM sensitive_areas sa, hazardous_sites hs
 WHERE st_overlaps (sa.zone, st_buffer (hs.location, .01)) = 1;

Sensitive Areas               Hazardous Sites
```

```
             3                W.H. KleenareChemical Repository
```

ST_Centroid

定義

ST_Centroid は、ポリゴン、マルチポリゴン、またはマルチラインストリングを入力として、ジオメトリのエンベロープの中心点を返します。つまり、中心点とは、ジオメトリの最小および最大 XY 範囲の中間点です。

構文

Oracle および PostgreSQL

```
sde.st_centroid (polygon sde.st_geometry)
sde.st_centroid (multipolygon sde.st_geometry)
sde.st_centroid (multilinestring sde.st_geometry)
```

SQLite

```
st_centroid (polygon geometryblob)
st_centroid (multipolygon geometryblob)
st_centroid (multilinestring geometryblob)
```

戻り値のタイプ

ST_Point

例

都市 GIS 技術者は、建物のマルチポリゴンを、建物の密集したグラフィックス内で 1 つのポイントとして表示したいと考えています。建物は、bfp テーブルに格納されています。このテーブルは、各データベースで示すステートメントで作成および入力されます。

Oracle

```
--Create and populate table
CREATE TABLE bfp (
 building_id integer,
 footprint sde.st_geometry);
INSERT INTO bfp VALUES (
 1,
 sde.st_polygon ('polygon ((0 0, 0 10, 10 10, 10 0, 0 0))', 4326)
);
INSERT INTO bfp VALUES (
 2,
 sde.st_polygon ('polygon ((20 0, 30 20, 40 0, 20 0))', 4326)
);
INSERT INTO bfp VALUES (
 3,
 sde.st_polygon ('polygon ((20 30, 25 35, 30 30, 20 30))', 4326)
);
```

```
--The ST_Centroid function returns the centroid of each building footprint
multipolygon.
```

```
--The ST_AsText function converts each centroid point into a text representation
recognized by the application.
SELECT building_id,
 sde.st_astext (sde.st_centroid (footprint)) Centroid
 FROM bfp;
BUILDING_ID                 Centroid
   1                    POINT  (5.00000000 5.00000000)
   2                    POINT  (30.00000000 10.00000000)
   3                    POINT  (25.00000000 32.50000000)
```

PostgreSQL

```
--Create and populate table
CREATE TABLE bfp (
 building_id serial,
 footprint sde.st_geometry);
INSERT INTO bfp (footprint) VALUES (
 sde.st_polygon ('polygon ((0 0, 0 10, 10 10, 10 0, 0 0))', 4326)
);
INSERT INTO bfp (footprint) VALUES (
 sde.st_polygon ('polygon ((20 0, 30 20, 40 0, 20 0))', 4326)
);
INSERT INTO bfp (footprint) VALUES (
 sde.st_polygon ('polygon ((20 30, 25 35, 30 30, 20 30))', 4326)
);
```

```
--The ST_Centroid function returns the centroid of each building footprint
multipolygon.
--The ST_AsText function converts each centroid point into a text representation
recognized by the application.
SELECT building_id, sde.st_astext (sde.st_centroid (footprint))
 AS centroid
 FROM bfp;
building_id                 centroid
   1                    POINT  (5 5)
   2                    POINT  (30 10)
   3                    POINT  (25 33)
```

SQLite

```
--Create table, add geometry column, and populate table
CREATE TABLE bfp (
 building_id integer primary key autoincrement not null
);
SELECT AddGeometryColumn (
 NULL,
 'bfp',
 'footprint',
 4326,
 'polygon',
 'xy',
 'null'
);
INSERT INTO bfp (footprint) VALUES (
 st_polygon ('polygon ((0 0, 0 10, 10 10, 10 0, 0 0))', 4326)
);
INSERT INTO bfp (footprint) VALUES (
```

```
 st_polygon ('polygon ((20 0, 30 20, 40 0, 20 0))', 4326)
);
INSERT INTO bfp (footprint) VALUES (
 st_polygon ('polygon ((20 30, 25 35, 30 30, 20 30))', 4326)
);
```

```
--The ST_Centroid function returns the centroid of each building footprint
multipolygon.
--The ST_AsText function converts each centroid point into a text representation
recognized by the application.
SELECT building_id, st_astext (st_centroid (footprint))
 AS "centroid"
 FROM bfp;
building_id              centroid
   1                     POINT  (5.00000000 5.00000000)
   2                     POINT  (30.00000000 10.0000000)
   3                     POINT  (25.00000000 32.5000000)
```

ST_Contains

定義

ST_Contains は、2 つのジオメトリ オブジェクトを入力として、最初のオブジェクトが 2 番目のオブジェクトを完全に含む場合は 1 (Oracle および SQLite) または t (PostgreSQL)、それ以外の場合は 0 (Oracle および SQLite) または f (PostgreSQL) を返します。

構文

Oracle および PostgreSQL

```
sde.st_contains (geometry1 sde.st_geometry, geometry2 sde.st_geometry)
```

SQLite

```
st_contains (geometry1 geometryblob, geometry2 geometryblob)
```

戻り値のタイプ

Boolean

例

以下の例では、2 つのテーブルが作成されます。buildingfootprints には都市の建物、lots にはその区画が含まれます。都市エンジニアは、すべての建物がその土地区画内にあることを確認したいと考えています。

都市エンジニアは、ST_Intersects および ST_Contains を使用して、1 つの区画に完全には収まっていない建物を選択します。

Oracle

```
--Create tables and insert values.
CREATE TABLE bfp (
 building_id integer,
 footprint sde.st_geometry
);

CREATE TABLE lots (
 lot_id integer,
 lot sde.st_geometry
);

INSERT INTO BFP (building_id, footprint) VALUES (
 1,
 sde.st_polygon ('polygon ((0 0, 0 10, 10 10, 10 0, 0 0))', 4326)
);

INSERT INTO BFP (building_id, footprint) VALUES (
 2,
 sde.st_polygon ('polygon ((20 0, 20 10, 30 10, 30 0, 20 0))', 4326)
);

INSERT INTO BFP (building_id, footprint) VALUES (
```

```
 3,
 sde.st_polygon ('polygon ((40 0, 40 10, 50 10, 50 0, 40 0))', 4326)
);

INSERT INTO LOTS (lot_id, lot) VALUES (
 1,
 sde.st_polygon ('polygon ((-1 -1, -1 11, 11 11, 11 -1, -1 -1))', 4326)
);

INSERT INTO LOTS (lot_id, lot) VALUES (
 2,
 sde.st_polygon ('polygon ((19 -1, 19 11, 29 9, 31 -1, 19 -1))', 4326)
);

INSERT INTO LOTS (lot_id, lot) VALUES (
 3,
 sde.st_polygon ('polygon ((39 -1, 39 11, 51 11, 51 -1, 39 -1))', 4326)
);
```

```
--Select the buildings that are not completely contained within one lot.
SELECT UNIQUE (building_id)
 FROM BFP, LOTS
 WHERE sde.st_intersects (lot, footprint) = 1
 AND sde.st_contains (lot, footprint) = 0;

BUILDING_ID

          2
```

PostgreSQL

```
--Create tables and insert values.
CREATE TABLE bfp (
 building_id serial,
 footprint st_geometry);

CREATE TABLE lots
 (lot_id serial,
 lot st_geometry);

INSERT INTO bfp (footprint) VALUES (
 st_polygon ('polygon ((0 0, 0 10, 10 10, 10 0, 0 0))', 4326)
);

INSERT INTO bfp (footprint) VALUES (
 st_polygon ('polygon ((20 0, 20 10, 30 10, 30 0, 20 0))', 4326)
);

INSERT INTO bfp (footprint) VALUES (
 st_polygon ('polygon ((40 0, 40 10, 50 10, 50 0, 40 0))', 4326)
);

INSERT INTO lots (lot) VALUES (
 st_polygon ('polygon ((-1 -1, -1 11, 11 11, 11 -1, -1 -1))', 4326)
);

INSERT INTO lots (lot) VALUES (
 st_polygon ('polygon ((19 -1, 19 11, 29 9, 31 -1, 19 -1))', 4326)
);
```

```
INSERT INTO lots (lot) VALUES (
 st_polygon ('polygon ((39 -1, 39 11, 51 11, 51 -1, 39 -1))', 4326)
);
```

```
--Select the buildings that are not completely contained within one lot.
SELECT DISTINCT (building_id)
 FROM bfp, lots
 WHERE st_intersects (lot, footprint) = 't'
 AND st_contains (lot, footprint) = 'f';

building_id

          2
```

SQLite

```
--Create tables, add geometry columns, and insert values.
CREATE TABLE bfp (
 building_id integer primary key autoincrement not null
);

SELECT AddGeometryColumn (
 NULL,
 'bfp',
 'footprint',
 4326,
 'polygon',
 'xy',
 'null'
);

CREATE TABLE lots
 (lot_id integer primary key autoincrement not null
);

SELECT AddGeometryColumn (
 NULL,
 'lots',
 'lot',
 4326,
 'polygon',
 'xy',
 'null'
);

INSERT INTO bfp (footprint) VALUES (
 st_polygon ('polygon ((0 0, 0 10, 10 10, 10 0, 0 0))', 4326)
);

INSERT INTO bfp (footprint) VALUES (
 st_polygon ('polygon ((20 0, 20 10, 30 10, 30 0, 20 0))', 4326)
);

INSERT INTO bfp (footprint) VALUES (
 st_polygon ('polygon ((40 0, 40 10, 50 10, 50 0, 40 0))', 4326)
);

INSERT INTO lots (lot) VALUES (
```

```
 st_polygon ('polygon ((-1 -1, -1 11, 11 11, 11 -1, -1 -1))', 4326)
);

INSERT INTO lots (lot) VALUES (
 st_polygon ('polygon ((19 -1, 19 11, 29 9, 31 -1, 19 -1))', 4326)
);

INSERT INTO lots (lot) VALUES (
 st_polygon ('polygon ((39 -1, 39 11, 51 11, 51 -1, 39 -1))', 4326)
);
```

```
--Select the buildings that are not completely contained within one lot.
SELECT DISTINCT (building_id)
 FROM bfp, lots
 WHERE st_intersects (lot, footprint) = 1
 AND st_contains (lot, footprint) = 0;

building_id

2
```

ST_ConvexHull

定義

ST_ConvexHull は、ST_Geometry オブジェクトの凸包を返します。

構文

Oracle および PostgreSQL

```
sde.st_convexhull (geometry1 sde.st_geometry)
```

SQLite

```
st_convexhull (geometry1 geometryblob)
```

戻り値のタイプ

Oracle および PostgreSQL

ST_Geometry

SQLite

Geometryblob

例

この例では、id、spatial_type、geometry という 3 つの列を持つ sample_geometries テーブルを作成します。spatial_type フィールドには、geometry 列に作成されたジオメトリのタイプが格納されます。 3 つのフィーチャ (ラインストリング、ポリゴン、マルチポイント) がテーブルに挿入されます。

ST_ConvexHull 関数が含まれている SELECT ステートメントが各ジオメトリの凸包を返します。

Oracle

```
--Create table and insert three sample geometries.
CREATE TABLE sample_geometries (
 id integer,
 spatial_type varchar(18),
 geometry sde.st_geometry
);

INSERT INTO sample_geometries (id, spatial_type, geometry) VALUES (
 1,
 'ST_LineString',
 sde.st_geometry ('linestring (20 20, 30 30, 20 40, 30 50)', 4326)
);

INSERT INTO sample_geometries (id, spatial_type, geometry) VALUES (
 2,
 'ST_Polygon',
 sde.st_geometry ('polygon ((30 30, 25 35, 15 50, 35 80, 40 85, 80 90, 70 75, 65 70, 55
```

```
50, 75 40, 60 30, 30 30))', 4326)
);

INSERT INTO sample_geometries (id, spatial_type, geometry) VALUES (
 3,
 'ST_MultiPoint',
 sde.st_geometry ('multipoint ((20 20), (30 30), (20 40), (30 50))', 4326)
);
```

```
--Find the convex hull of each geometry subtype.
SELECT id, spatial_type, sde.st_astext (sde.st_convexhull (geometry)) CONVEXHULL
 FROM  SAMPLE_GEOMETRIES;

ID    SPATIAL_TYPE       CONVEXHULL

 1    ST_LineString      POLYGON ((20.00000000 40.00000000,
                           20.00000000 20.00000000, 30.00000000
                           30.00000000, 30.00000000 50.00000000,
                           20.00000000 40.00000000))

 2    ST_Polygon         POLYGON ((15.00000000 50.00000000,
                           25.00000000 35.00000000, 30.00000000
                           30.00000000, 60.00000000 30.00000000,
                           75.00000000 40.00000000, 80.00000000
                           90.00000000, 40.00000000 85.00000000,
                           35.00000000 80.00000000, 15.00000000
                           50.00000000))

 3    ST_MultiPoint      POLYGON ((20.00000000 40.00000000,
                           20.00000000 20.00000000, 30.00000000
                           30.00000000, 30.00000000 50.00000000,
                           20.00000000 40.00000000))
```

PostgreSQL

```
--Create table and insert three sample geometries.
CREATE TABLE sample_geometries (
 id integer,
 spatial_type varchar(18),
 geometry sde.st_geometry
);

INSERT INTO sample_geometries (id, spatial_type, geometry) VALUES (
 1,
 'ST_LineString',
 sde.st_geometry ('linestring (20 20, 30 30, 20 40, 30 50)', 4326)
);

INSERT INTO sample_geometries (id, spatial_type, geometry) VALUES (
 2,
 'ST_Polygon',
 sde.st_geometry ('polygon ((30 30, 25 35, 15 50, 35 80, 40 85, 80 90, 70 75, 65 70, 55
50, 75 40, 60 30, 30 30))', 4326)
);

INSERT INTO sample_geometries (id, spatial_type, geometry) VALUES (
 3,
 'ST_MultiPoint',
 sde.st_geometry ('multipoint (20 20, 30 30, 20 40, 30 50)', 4326)
```

```
);
```

```
--Find the convex hull of each geometry subtype.
SELECT id, spatial_type, st_astext (sde.st_convexhull (geometry))
 AS CONVEXHULL
 FROM sample_geometries;

id    spatial_type       convexhull

 1    ST_LineString      POLYGON (( 20 40, 20 20, 30 30, 30 50,
                           20 40))

 2    ST_Polygon         POLYGON (( 15 50, 25 35, 30 30, 60 30,
                           75 40, 80 90, 40 85, 35 80, 15 50))

 3    ST_MultiPoint      POLYGON (( 20 40, 20 20, 30 30, 30 50,
                           20 40))
```

SQLite

```
--Create table and insert three sample geometries.
CREATE TABLE sample_geometries (
 id integer primary key autoincrement not null,
 spatial_type varchar(18)
);

SELECT AddGeometryColumn(
 NULL,
 'sample_geometries',
 'geometry',
 4326,
 'geometry',
 'xy',
 'null'
);

INSERT INTO sample_geometries (spatial_type, geometry) VALUES (
 'ST_LineString',
 st_geometry ('linestring (20 20, 30 30, 20 40, 30 50)', 4326)
);

INSERT INTO sample_geometries (spatial_type, geometry) VALUES (
 'ST_Polygon',
 st_geometry ('polygon ((30 30, 25 35, 15 50, 35 80, 40 85, 80 90, 70 75, 65 70, 55 50,
75 40, 60 30, 30 30))', 4326)
);

INSERT INTO sample_geometries (spatial_type, geometry) VALUES (
 'ST_MultiPoint',
 st_geometry ('multipoint ((20 20), (30 30), (20 40), (30 50))', 4326)
);
```

```
--Find the convex hull of each geometry subtype.
SELECT id, spatial_type, st_astext (st_convexhull (geometry))
 AS CONVEXHULL
 FROM sample_geometries;
```

```
id    spatial_type       CONVEXHULL

 1    ST_LineString      POLYGON ((20.00000000 40.00000000,
                           20.00000000 20.00000000, 30.00000000
                           30.00000000, 30.00000000 50.00000000,
                           20.00000000 40.00000000))

 2    ST_Polygon         POLYGON ((15.00000000 50.00000000,
                           25.00000000 35.00000000, 30.00000000
                           30.00000000, 60.00000000 30.00000000,
                           75.00000000 40.00000000, 80.00000000
                           90.00000000, 40.00000000 85.00000000,
                           35.00000000 80.00000000, 15.00000000
                           50.00000000))

 3    ST_MultiPoint      POLYGON ((20.00000000 40.00000000,
                           20.00000000 20.00000000, 30.00000000
                           30.00000000, 30.00000000 50.00000000,
                           20.00000000 40.00000000))
```

ST_CoordDim

定義

ST_CoordDim は、ジオメトリ列の座標値のディメンション (次元数) を返します。

構文

Oracle および PostgreSQL

```
sde.st_coorddim (geometry1 sde.st_geometry)
```

SQLite

```
st_coorddim (geometry1 geometryblob)
```

戻り値のタイプ

Integer

2 = XY 座標

3 = XYZ または XYM 座標

4 = XYZM 座標

例

この例では、geotype および g1 列を持つ coorddim_test テーブルを作成します。geotype 列は、g1 ジオメトリ列に格納されるジオメトリ サブクラスおよびディメンションの名前を格納します。

SELECT ステートメントは、geotype 列に格納されているサブクラス名と、そのジオメトリの座標のディメンションをリストします。

Oracle

```
--Create test table.
CREATE TABLE coorddim_test (
 geotype varchar(20),
 g1 sde.st_geometry
);
```

```
--Insert values to the test table.
INSERT INTO COORDDIM_TEST VALUES (
 'Point',
 sde.st_geometry ('point (60.567222 -140.404)', 4326)
);

INSERT INTO COORDDIM_TEST VALUES (
 'Point Z',
 sde.st_geometry ('point Z (60.567222 -140.404 5959)', 4326)
);
```

```
INSERT INTO COORDDIM_TEST VALUES (
 'Point M',
 sde.st_geometry ('point M (60.567222 -140.404 5250)', 4326)
);

INSERT INTO COORDDIM_TEST VALUES (
 'Point ZM',
 sde.st_geometry ('point ZM (60.567222 -140.404 5959 5250)', 4326)
);
```

```
--Determine the dimensionality of each feature.
SELECT geotype, sde.st_coorddim (g1) coordinate_dimension
 FROM COORDDIM_TEST;

GEOTYPE               coordinate_dimension

Point                           2
Point Z                         3
Point M                         3
Point ZM                        4
```

PostgreSQL

```
--Create test table.
CREATE TABLE coorddim_test (
 geotype varchar(20),
 g1 sde.st_geometry
);
```

```
--Insert values to the test table.
INSERT INTO coorddim_test VALUES (
 'Point',
 st_point ('point (60.567222 -140.404)', 4326)
);

INSERT INTO coorddim_test VALUES (
 'Point Z',
 st_point ('point z (60.567222 -140.404 5959)', 4326)
);

INSERT INTO coorddim_test VALUES (
 'Point M',
 st_point ('point m (60.567222 -140.404 5250)', 4326)
);

INSERT INTO coorddim_test VALUES (
 'Point ZM',
 st_point ('point zm (60.567222 -140.404 5959 5250)', 4326)
);
```

```
--Determine the dimensionality of each feature.
SELECT geotype, st_coorddim (g1)
 AS coordinate_dimension
 FROM coorddim_test;
```

```
geotype              coordinate_dimension

Point                          2
Point Z                        3
Point M                        3
Point ZM                       4
```

SQLite

```
--Create test tables and add geometry columns.
CREATE TABLE coorddim_test (
 geotype varchar(20)
);

SELECT AddGeometryColumn(
 NULL,
 'coorddim_test',
 'g1',
 4326,
 'pointzm',
 'xyzm',
 'null'
);

CREATE TABLE coorddim_test2 (
 geotype varchar(20)
);

SELECT AddGeometryColumn(
 NULL,
 'coorddim_test2',
 'g1',
 4326,
 'pointz',
 'xyz',
 'null'
);

CREATE TABLE coorddim_test3 (
 geotype varchar(20)
);

SELECT AddGeometryColumn(
 NULL,
 'coorddim_test3',
 'g1',
 4326,
 'pointm',
 'xym',
 'null'
);

CREATE TABLE coorddim_test4 (
 geotype varchar(20)
);

SELECT AddGeometryColumn(
 NULL,
 'coorddim_test4',
 'g1',
 4326,
```

```
 'point',
 'xy',
 'null'
);
```

```
--Insert values to the test table.
INSERT INTO coorddim_test4 VALUES (
 'Point',
 st_point ('point (60.567222 -140.404)', 4326)
);

INSERT INTO coorddim_test2 VALUES (
 'Point Z',
 st_point ('point z (60.567222 -140.404 5959)', 4326)
);

INSERT INTO coorddim_test3 VALUES (
 'Point M',
 st_point ('point m (60.567222 -140.404 5250)', 4326)
);

INSERT INTO coorddim_test VALUES (
 'Point ZM',
 st_point ('point zm (60.567222 -140.404 5959 5250)', 4326)
);
```

```
--Determine the dimensionality of features in each table.
SELECT geotype, st_coorddim (g1)
 AS coordinate_dimension
 FROM coorddim_test;

geotype              coordinate_dimension

Point ZM                       4


SELECT geotype, st_coorddim (g1)
 AS coordinate_dimension
 FROM coorddim_test2;

geotype              coordinate_dimension

Point Z                        3


SELECT geotype, st_coorddim (g1)
 AS coordinate_dimension
 FROM coorddim_test3;

geotype              coordinate_dimension

Point M                        3


SELECT geotype, st_coorddim (g1)
 AS coordinate_dimension
 FROM coorddim_test4;

geotype              coordinate_dimension
```

```
Point                          2
```

ST_Crosses

定義

ST_Crosses は、2 つの ST_Geometry オブジェクトを入力として、そのインターセクトの結果であるジオメトリ オブジェクトのディメンションが、ソース オブジェクトの最大ディメンションより 1 つ少ない場合は 1 (Oracle および SQLite) または t (PostgreSQL) を返します。インターセクト オブジェクトには、両方のソース ジオメトリの内部にあり、どちらのソース オブジェクトとも等しくないポイントが含まれる必要があります。それ以外の場合は 0 (Oracle および SQLite) または f (PostgreSQL) を返します。

構文

```
sde.st_crosses (geometry1 sde.st_geometry, geometry2 sde.st_geometry)
```

Oracle および PostgreSQL

```
sde.st_crosses (geometry1 sde.st_geometry, geometry2 sde.st_geometry)
```

SQLite

```
st_crosses (geometry1 geometryblob, geometry2 geometryblob)
```

戻り値のタイプ

Boolean

例

国の行政機関が、国内にあるすべての有害廃棄物貯蔵施設は、水路の特定半径内にあってはならないという新しい規制を検討しています。国の GIS マネージャーは、waterways テーブルにラインストリングとして格納された水路と河川の正確な表現を持っていますが、それぞれの有害廃棄物貯蔵施設については 1 つのポイントの位置しか持っていません。

GIS マネージャーは、規制案に違反している既存施設を国の行政官に警告する必要があるかどうかを判断するために、hazardous_sites の位置のバッファーを作成して、このバッファー ポリゴンを横断する川や河川がないか確認する必要があります。cross 述語は、バッファーされた hazardous_sites ポイントを水路と比較し、水路が国の規制案の半径を横断するレコードのみを返します。

Oracle

```
--Define tables and insert values.
CREATE TABLE waterways (
 id integer,
 name varchar(128),
 water sde.st_geometry
);

CREATE TABLE hazardous_sites (
 site_id integer,
 name varchar(40),
```

```
 location sde.st_geometry
);

INSERT INTO waterways VALUES (
 2,
 'Zanja',
 sde.st_geometry ('linestring (40 50, 50 40)', 4326)
);

INSERT INTO waterways VALUES (
 3,
 'Keshequa',
 sde.st_geometry ('linestring (20 20, 60 60)', 4326)
);

INSERT INTO hazardous_sites VALUES (
 4,
 'StorIt',
 sde.st_point ('point (60 60)', 4326)
);

INSERT INTO hazardous_sites VALUES (
 5,
 'Glowing Pools',
 sde.st_point ('point (30 30)', 4326)
);
```

```
--Buffer hazardous waste sites and find if any buffers cross a waterway.
SELECT UNIQUE (ww.name) "River or stream", hs.name "Hazardous sites"
 FROM WATERWAYS ww, HAZARDOUS_SITES hs
 WHERE sde.st_crosses (sde.st_buffer (hs.location, .01), ww.water) = 1;

River or stream                                Hazardous sites

Keshequa                                                    StorIt
Keshequa           Glowing Pools
```

PostgreSQL

```
--Define tables and insert values.
CREATE TABLE waterways (
 id serial,
 name varchar(128),
 water sde.st_geometry
);

CREATE TABLE hazardous_sites (
 site_id integer,
 name varchar(40),
 location sde.st_geometry
);

INSERT INTO waterways (name, water) VALUES (
 'Zanja',
 sde.st_geometry ('linestring (40 50, 50 40)', 4326)
);

INSERT INTO waterways (name, water) VALUES (
 'Keshequa',
```

```
 sde.st_geometry ('linestring (20 20, 60 60)', 4326)
);

INSERT INTO hazardous_sites (name, location) VALUES (
 'StorIt',
 sde.st_point ('point (60 60)', 4326)
);

INSERT INTO hazardous_sites (name, location) VALUES (
 'Glowing Pools',
 sde.st_point ('point (30 30)', 4326)
);
```

```
--Buffer hazardous waste sites and find if any buffers cross a waterway.
SELECT DISTINCT (ww.name) AS "River or stream", hs.name AS "Hazardous sites"
 FROM waterways ww, hazardous_sites hs
 WHERE sde.st_crosses (sde.st_buffer (hs.location, .01), ww.water) = 't';

River or stream                          Hazardous sites

Keshequa                                                 StorIt
Keshequa             Glowing Pools
```

SQLite

```
--Define tables and insert values.
CREATE TABLE waterways (
 id integer primary key autoincrement not null,
 name varchar(128)
);

SELECT AddGeometryColumn(
 NULL,
 'waterways',
 'water',
 4326,
 'linestring',
 'xy',
 'null'
);

CREATE TABLE hazardous_sites (
 site_id integer primary key autoincrement not null,
 name varchar(40)
);

SELECT AddGeometryColumn(
 NULL,
 'hazardous_sites',
 'location',
 4326,
 'point',
 'xy',
 'null'
);

INSERT INTO waterways (name, water) VALUES (
 'Zanja',
 st_geometry ('linestring (40 50, 50 40)', 4326)
```

```
);

INSERT INTO waterways (name, water) VALUES (
 'Keshequa',
 st_geometry ('linestring (20 20, 60 60)', 4326)
);

INSERT INTO hazardous_sites (name, location) VALUES (
 'StorIt',
 st_point ('point (60 60)', 4326)
);

INSERT INTO hazardous_sites (name, location) VALUES (
 'Glowing Pools',
 st_point ('point (30 30)', 4326)
);
```

```
--Buffer hazardous waste sites and find if any buffers cross a waterway.
SELECT DISTINCT (ww.name) AS "River or stream", hs.name AS "Hazardous sites"
 FROM waterways ww, hazardous_sites hs
 WHERE st_crosses (st_buffer (hs.location, .01), ww.water) = 1;

River or stream                          Hazardous sites

Keshequa                                                StorIt
Keshequa             Glowing Pools
```

ST_Curve

注意:
Oracle と SQLite のみ

定義

ST_Curve は、WKT 表現から曲線フィーチャを構築します。

構文

Oracle

```
sde.st_curve (wkt clob, srid integer)
```

SQLite

```
st_curve (wkt text, srid int32)
```

戻り値のタイプ

ST_LineString

例

この例では、曲線ジオメトリを持つテーブルを作成し、テーブルに値を挿入し、テーブルからフィーチャを 1 つ選択しています。

Oracle

```
CREATE TABLE curve_test (
 id integer,
 geometry sde.st_curve
);

INSERT INTO CURVE_TEST VALUES (
 1910,
 sde.st_curve ('linestring (33 2, 34 3, 35 6)', 4326)
);

SELECT id, sde.st_astext (geometry) CURVE
 FROM CURVE_TEST;

ID     CURVE

1110   LINESTRING (33.00000000 2.00000000, 34.00000000 3.00000000,
              35.00000000 6.00000000)
```

SQLite

```
CREATE TABLE curve_test (
 id integer primary key autoincrement not null
);

SELECT AddGeometryColumn(
 NULL,
 'curve_test',
 'geometry',
 4326,
 'linestring',
 'xy',
 'null'
);

INSERT INTO CURVE_TEST (geometry) VALUES (
 st_curve ('linestring (33 2, 34 3, 35 6)', 4326)
);

SELECT id, st_astext (geometry)
 AS curve
 FROM curve_test;

id     curve

1   LINESTRING (33.00000000 2.00000000, 34.00000000 3.00000000,
       35.00000000 6.00000000)
```

ST_Difference

定義

ST_Difference は、2 つのジオメトリ オブジェクトを入力として、ソース オブジェクトと一致しないジオメトリ オブジェクトを返します。

構文

Oracle および PostgreSQL

```
sde.st_difference (geometry1 sde.st_geometry, geometry2 sde.st_geometry)
```

SQLite

```
st_difference (geometry1 geometryblob, geometry2 geometryblob)
```

戻り値のタイプ

Oracle および PostgreSQL

ST_Geometry

SQLite

Geometryblob

例

この例では、都市エンジニアは、建物に占有されていない都市の区画の合計面積を知る必要があるため、建物の面積を除いた後の区画の合計面積を求めています。

都市エンジニアは、建物と区画テーブルを lot_id で等価結合し、区画から建物を引いた一致しない部分の面積の合計を取得しています。

Oracle

```
--Create tables and insert values
CREATE TABLE footprints (
 building_id integer,
 footprint sde.st_geometry
);

CREATE TABLE lots (
 lot_id integer,
 lot sde.st_geometry
);

INSERT INTO footprints (building_id, footprint) VALUES (
 1,
 sde.st_polygon ('polygon ((0 0, 0 10, 10 10, 10 0, 0 0))', 4326)
);
```

```
INSERT INTO footprints (building_id, footprint) VALUES (
 2,
 sde.st_polygon ('polygon ((20 0, 20 10, 30 10, 30 0, 20 0))', 4326)
);

INSERT INTO footprints (building_id, footprint) VALUES (
 3,
 sde.st_polygon ('polygon ((40 0, 40 10, 50 10, 50 0, 40 0))', 4326)
);

INSERT INTO lots (lot_id, lot) VALUES (
 1,
 sde.st_polygon ('polygon ((-1 -1, -1 11, 11 11, 11 -1, -1 -1))', 4326)
);

INSERT INTO lots (lot_id, lot) VALUES (
 2,
 sde.st_polygon ('polygon ((19 -1, 19 11, 29 9, 31 -1, 19 -1))', 4326)
);

INSERT INTO lots (lot_id, lot) VALUES (
 3,
 sde.st_polygon ('polygon ((39 -1, 39 11, 51 11, 51 -1, 39 -1))', 4326)
);
```

```
SELECT SUM (sde.st_area (sde.st_difference (lot, footprint)))
  FROM FOOTPRINTS bf, LOTS
  WHERE bf.building_id = lots.lot_id;

SUM(ST_AREA(ST_DIFFERENCE(LOT,FOOTPRINT)))

114
```

PostgreSQL

```
--Create tables and insert values
CREATE TABLE footprints (
 building_id integer,
 footprint sde.st_geometry
);

CREATE TABLE lots (
 lot_id integer,
 lot sde.st_geometry
);

INSERT INTO footprints (building_id, footprint) VALUES (
 1,
 sde.st_polygon ('polygon ((0 0, 0 10, 10 10, 10 0, 0 0))', 4326)
);

INSERT INTO footprints (building_id, footprint) VALUES (
 2,
 sde.st_polygon ('polygon ((20 0, 20 10, 30 10, 30 0, 20 0))', 4326)
);

INSERT INTO footprints (building_id, footprint) VALUES (
 3,
 sde.st_polygon ('polygon ((40 0, 40 10, 50 10, 50 0, 40 0))', 4326)
```

```
);

INSERT INTO lots (lot_id, lot) VALUES (
 1,
 sde.st_polygon ('polygon ((-1 -1, -1 11, 11 11, 11 -1, -1 -1))', 4326)
);

INSERT INTO lots (lot_id, lot) VALUES (
 2,
 sde.st_polygon ('polygon ((19 -1, 19 11, 29 9, 31 -1, 19 -1))', 4326)
);

INSERT INTO lots (lot_id, lot) VALUES (
 3,
 sde.st_polygon ('polygon ((39 -1, 39 11, 51 11, 51 -1, 39 -1))', 4326)
);
```

```
SELECT SUM (sde.st_area (sde.st_difference (lot, footprint)))
 FROM footprints bf, lots
 WHERE bf.building_id = lots.lot_id;

sum

114
```

SQLite

```
--Create tables, add geometry columns, and insert values
CREATE TABLE footprints (
 building_id integer primary key autoincrement not null
);

SELECT AddGeometryColumn (
 NULL,
 'footprints',
 'footprint',
 4326,
 'polygon',
 'xy',
 'null'
);

CREATE TABLE lots (
 lot_id integer primary key autoincrement not null
);

SELECT AddGeometryColumn (
 NULL,
 'lots',
 'lot',
 4326,
 'polygon',
 'xy',
 'null'
);

INSERT INTO footprints (footprint) VALUES (
 st_polygon ('polygon ((0 0, 0 10, 10 10, 10 0, 0 0))', 4326)
);
```

```
INSERT INTO footprints (footprint) VALUES (
 st_polygon ('polygon ((20 0, 20 10, 30 10, 30 0, 20 0))', 4326)
);

INSERT INTO footprints (footprint) VALUES (
 st_polygon ('polygon ((40 0, 40 10, 50 10, 50 0, 40 0))', 4326)
);

INSERT INTO lots (lot) VALUES (
 st_polygon ('polygon ((-1 -1, -1 11, 11 11, 11 -1, -1 -1))', 4326)
);

INSERT INTO lots (lot) VALUES (
 st_polygon ('polygon ((19 -1, 19 11, 29 9, 31 -1, 19 -1))', 4326)
);

INSERT INTO lots (lot) VALUES (
 st_polygon ('polygon ((39 -1, 39 11, 51 11, 51 -1, 39 -1))', 4326)
);
```

```
SELECT SUM (st_area (st_difference (lot, footprint)))
 FROM footprints bf, lots
 WHERE bf.building_id = lots.lot_id;

sum

114.0
```

ST_Dimension

定義

ST_Dimension はジオメトリ オブジェクトのディメンションを返します。 ここでは、ディメンションとは長さと幅を指します。 たとえば、ポイントには長さも幅もないので、ディメンションは 0 になります。一方で、ラインには長さはあるが幅はないので、ディメンションは 1 になります。

構文

Oracle および PostgreSQL

```
sde.st_dimension (geometry1 sde.st_geometry)
```

SQLite

```
st_dimension (geometry1 geometryblob)
```

戻り値のタイプ

Integer

例

dimension_test テーブルは、ジオタイプ列と g1 列で作成されます。 ジオタイプ列には、g1 ジオメトリ列に格納されているサブクラスの名前が格納されます。

SELECT ステートメントは、ジオタイプ列に格納されているサブクラス名を、そのジオタイプのディメンションとともにリストします。

Oracle

```
CREATE TABLE dimension_test (
 geotype varchar(20),
 g1 sde.st_geometry
);

INSERT INTO DIMENSION_TEST VALUES (
 'Point',
 sde.st_pointfromtext ('point (10.02 20.01)', 4326)
);

INSERT INTO DIMENSION_TEST VALUES (
 'Linestring',
 sde.st_linefromtext ('linestring (10.02 20.01, 10.32 23.98, 11.92 25.64)', 4326)
);

INSERT INTO DIMENSION_TEST VALUES (
 'Polygon',
 sde.st_polyfromtext ('polygon ((10.02 20.01, 11.92 35.64, 25.02 34.15, 19.15 33.94,
10.02 20.01))', 4326)
);

INSERT INTO DIMENSION_TEST VALUES (
```

```
 'Multipoint',
 sde.st_mpointfromtext ('multipoint ((10.02 20.01), (10.32 23.98), (11.92 25.64))',
4326)
);

INSERT INTO DIMENSION_TEST VALUES (
 'Multilinestring',
 sde.st_mlinefromtext ('multilinestring ((10.02 20.01, 10.32 23.98, 11.92 25.64), (9.55
23.75, 15.36 30.11))', 4326)
);

INSERT INTO DIMENSION_TEST VALUES (
 'Multipolygon',
 sde.st_mpolyfromtext ('multipolygon (((10.02 20.01, 11.92 35.64, 25.02 34.15,
19.15 33.94, 10.02 20.01), (51.71 21.73, 73.36 27.04, 71.52 32.87,
52.43 31.90, 51.71 21.73)))', 4326)
);
```

```
SELECT geotype, sde.st_dimension (g1) Dimension
 FROM DIMENSION_TEST;

GEOTYPE                    Dimension

Point                       0
Linestring                  1
Polygon                     2
Multipoint                  0
Multilinestring             1
Multipolygon                2
```

PostgreSQL

```
CREATE TABLE dimension_test (
 geotype varchar(20),
 g1 sde.st_geometry
);

INSERT INTO dimension_test VALUES (
 'Point',
 sde.st_point ('point (10.02 20.01)', 4326)
);

INSERT INTO dimension_test VALUES (
 'Linestring',
 sde.st_linestring ('linestring (10.02 20.01, 10.32 23.98, 11.92 25.64)', 4326)
);

INSERT INTO dimension_test VALUES (
 'Polygon',
 sde.st_polygon ('polygon ((10.02 20.01, 11.92 35.64, 25.02 34.15, 19.15 33.94, 10.02
20.01))', 4326)
);

INSERT INTO dimension_test VALUES (
 'Multipoint',
 sde.st_multipoint ('multipoint (10.02 20.01, 10.32 23.98, 11.92 25.64)', 4326)
);

INSERT INTO dimension_test VALUES (
```

```
 'Multilinestring',
 sde.st_multilinestring ('multilinestring ((10.02 20.01, 10.32 23.98, 11.92 25.64),
(9.55 23.75, 15.36 30.11))', 4326)
);

INSERT INTO dimension_test VALUES (
 'Multipolygon',
 sde.st_multipolygon ('multipolygon (((10.02 20.01, 11.92 35.64, 25.02 34.15,
19.15 33.94, 10.02 20.01), (51.71 21.73, 73.36 27.04, 71.52 32.87,
52.43 31.90, 51.71 21.73)))', 4326)
);
```

```
SELECT geotype, sde.st_dimension (g1)
 AS Dimension
 FROM dimension_test;

geotype                  dimension

Point                       0
Linestring                  1
Polygon                     2
Multipoint                  0
Multilinestring             1
Multipolygon                2
```

SQLite

```
CREATE TABLE dimension_test (
 geotype varchar(20)
);

SELECT AddGeometryColumn (
 NULL,
 'dimension_test',
 'g1',
 4326,
 'geometry',
 'xy',
 'null'
);

INSERT INTO dimension_test VALUES (
 'Point',
 st_point ('point (10.02 20.01)', 4326)
);

INSERT INTO dimension_test VALUES (
 'Linestring',
 st_linestring ('linestring (10.02 20.01, 10.32 23.98, 11.92 25.64)', 4326)
);

INSERT INTO dimension_test VALUES (
 'Polygon',
 st_polygon ('polygon ((10.02 20.01, 11.92 35.64, 25.02 34.15, 19.15 33.94, 10.02
20.01))', 4326)
);

INSERT INTO dimension_test VALUES (
 'Multipoint',
```

```
 st_multipoint ('multipoint ((10.02 20.01), (10.32 23.98), (11.92 25.64))', 4326)
);

INSERT INTO dimension_test VALUES (
 'Multilinestring',
 st_multilinestring ('multilinestring ((10.02 20.01, 10.32 23.98, 11.92 25.64), (9.55
23.75, 15.36 30.11))', 4326)
);

INSERT INTO dimension_test VALUES (
 'Multipolygon',
 st_multipolygon ('multipolygon (((10.02 20.01, 11.92 35.64, 25.02 34.15,
19.15 33.94, 10.02 20.01), (51.71 21.73, 73.36 27.04, 71.52 32.87,
52.43 31.90, 51.71 21.73)))', 4326)
);
```

```
SELECT geotype, st_dimension (g1)
 AS "Dimension"
 FROM dimension_test;

geotype                  Dimension

Point                       0
Linestring                  1
Polygon                     2
Multipoint                  0
Multilines                  1
Multipolyg                  2
```

ST_Disjoint

定義

ST_Disjoint は、2 つのジオメトリを入力として、2 つのジオメトリのインターセクトが空のセットを生成する場合は 1 (Oracle および SQLite) または t (PostgreSQL)、それ以外の場合は 0 (Oracle および SQLite) または f (PostgreSQL) を返します。

構文

Oracle および PostgreSQL

```
sde.st_disjoint (geometry1 sde.st_geometry, geometry2 sde.st_geometry)
```

SQLite

```
st_disjoint (geometry1 geometryblob, geometry2 geometryblob)
```

戻り値のタイプ

Boolean

例

この例では、2 つのテーブル (distribution_areas および factories) が作成され、それぞれに値が挿入されます。次に、工場の周囲にバッファーが作成され、st_disjoint を使用して分布エリアをまたがっていない工場のバッファーを特定します。

ヒント:

ST_Intersects と ST_Disjoint は反対の結果を返すため、関数の結果が 0 と等しくなるようにすることで、このクエリで ST_Intersects 関数を代わりに使用することができます。ST_Intersects 関数は、クエリを評価するときに空間インデックスを使用しますが、ST_Disjoint 関数は使用しません。

Oracle

```
--Create tables and insert values.
CREATE TABLE distribution_areas (
 id integer,
 areas sde.st_geometry
);

CREATE TABLE factories (
 id integer,
 loc sde.st_geometry
);

INSERT INTO distribution_areas (id, areas) VALUES (
 1,
 sde.st_geometry ('polygon ((20 30, 30 30, 30 40, 20 40, 20 30))', 4326)
);
```

```
INSERT INTO distribution_areas (id, areas) VALUES (
 2,
 sde.st_geometry ('polygon ((30 30, 30 50, 50 50, 50 30, 30 30))', 4326)
);

INSERT INTO distribution_areas (id, areas) VALUES (
 3,
 sde.st_geometry ('polygon ((40 40, 40 60, 60 60, 60 40, 40 40))', 4326)
);

INSERT INTO factories (id,loc) VALUES (
 4,
 sde.st_geometry ('point (60 60)', 4326)
);

INSERT INTO factories (id,loc) VALUES (
 5,
 sde.st_geometry ('point (30 30)', 4326)
);
```

```
--Buffer factories and find which buffers are separate from distribution areas.
SELECT da.id
 FROM DISTRIBUTION_AREAS da, FACTORIES f
 WHERE sde.st_disjoint ((sde.st_buffer (f.loc, .001)), da.areas) = 1;
```

PostgreSQL

```
--Create tables and insert values.
CREATE TABLE distribution_areas (
 id serial,
 areas sde.st_geometry
);

CREATE TABLE factories (
 id serial,
 loc sde.st_geometry
);

INSERT INTO distribution_areas (areas) VALUES (
 sde.st_geometry ('polygon ((20 30, 30 30, 30 40, 20 40, 20 30))', 4326)
);

INSERT INTO distribution_areas (areas) VALUES (
 sde.st_geometry ('polygon ((30 30, 30 50, 50 50, 50 30, 30 30))', 4326)
);

INSERT INTO distribution_areas (areas) VALUES (
 sde.st_geometry ('polygon ((40 40, 40 60, 60 60, 60 40, 40 40))', 4326)
);

INSERT INTO factories (loc) VALUES (
 sde.st_geometry ('point (60 60)', 4326)
);

INSERT INTO factories (loc) VALUES (
 sde.st_geometry ('point (30 30)', 4326)
);
```

```
--Buffer factories and find which buffers are separate from distribution areas.
SELECT da.id
 FROM distribution_areas da, factories f
 WHERE sde.st_disjoint ((sde.st_buffer (f.loc, .001)), da.areas) = 't';
```

SQLite

```
--Create tables and insert values.
CREATE TABLE distribution_areas (
 id integer primary key autoincrement not null
);

SELECT AddGeometryColumn (
 NULL,
 'distribution_areas',
 'areas',
 4326,
 'polygon',
 'xy',
 'null'
);

CREATE TABLE factories (
 id integer primary key autoincrement not null
);
```

```
SELECT AddGeometryColumn (
 NULL,
 'factories',
 'loc',
 4326,
 'point',
 'xy',
 'null'
);

INSERT INTO distribution_areas (areas) VALUES (
 st_geometry ('polygon ((20 30, 30 30, 30 40, 20 40, 20 30))', 4326)
);

INSERT INTO distribution_areas (areas) VALUES (
 st_geometry ('polygon ((30 30, 30 50, 50 50, 50 30, 30 30))', 4326)
);

INSERT INTO distribution_areas (areas) VALUES (
 st_geometry ('polygon ((40 40, 40 60, 60 60, 60 40, 40 40))', 4326)
);

INSERT INTO factories (loc) VALUES (
 st_geometry ('point (60 60)', 4326)
);

INSERT INTO factories (loc) VALUES (
 st_geometry ('point (30 30)', 4326)
);
```

```
--Buffer factories and find which buffers are separate from distribution areas.
SELECT da.id
 FROM distribution_areas da, factories f
 WHERE st_disjoint ((st_buffer (f.loc, .001)), da.areas) = 1;

id

1
2
3
```

ST_Distance

定義

ST_Distance は、2 つのジオメトリ間の距離を返します。距離は、2 つのジオメトリの最も近い頂点から測定されます。

構文

Oracle および PostgreSQL

```
sde.st_distance (geometry1 sde.st_geometry, geometry2 sde.st_geometry)
```

```
sde.st_distance (geometry1 sde.st_geometry, geometry2 sde.st_geometry, unit_name text)
```

SQLite

```
st_distance (geometry1 geometryblob, geometry2 geometryblob)
```

```
st_distance (geometry1 geometryblob, geometry2 geometryblob, unit_name text)
```

有効な単位名は次のとおりです。

ミリメートル	インチ	ヤード	リンク
センチメートル	インチ (US)	ヤード（US）	リンク（US）
デシメートル	フィート	ヤード（Clarke）	リンク（Clarke）
メートル	フィート (US)	ヤード（Sears）	リンク（Sears）
メートル（German）	フィート (Clarke)	ヤード（Sears_1922_Truncated）	リンク（Sears_1922_Truncated）
キロメートル	フィート (Sears)	ヤード（Benoit_1895_A）	リンク（Benoit_1895_B）
50 キロメートル	フィート (Sears_1922_Truncated)	ヤード（Indian）	チェーン
150 キロメートル	フィート (Benoit_1895_A)	ヤード（Indian_1937）	チェーン（US）
US Vara	フィート (1865)	ヤード（Indian_1962）	チェーン（Clarke）
スムート	フィート (Indian)	ヤード（Indian_1975）	チェーン（Sears）
	フィート (Indian_1937)	ファゾム	チェーン（Sears_1922_Truncated）
	フィート (Indian_1962)	マイル（US）	チェーン（Benoit_1895_A）
	フィート (Indian_1975)	法定マイル	ロッド

	フィート (Gold_Coast)	海里	ロッド（US）
	フィート (British_1936)	海里（US）	
		海里（UK）	

戻り値のタイプ

Double precision

例

2 つのテーブル（study1 および zones）が作成され、レコードが追加されます。ST_Distance 関数を使用して、各サブエリアの境界と、study1 エリアにある利用コード 400 のポリゴン間の距離を測定します。このシェープには 3 つのゾーンがあるため、3 レコードが返されます。

単位を指定しない場合、ST_Distance はデータの投影の単位を使用します。最初の例では、10 進の度になります。最後の 2 つの例では、キロメートルが指定されています。このため、距離はキロメートル単位で返されます。

Oracle および PostgreSQL

```
--Create tables and insert values.
CREATE TABLE zones (
 sa_id integer,
 usecode integer,
 shape sde.st_geometry
);
CREATE TABLE study1 (
 code integer unique,
 shape sde.st_geometry
);
INSERT INTO zones (sa_id, usecode, shape) VALUES (
 1,
 400,
 sde.st_polygon ('polygon ((0 0, 0 10, 10 10, 10 0, 0 0))', 4326)
);
INSERT INTO zones (sa_id, usecode, shape) VALUES (
 2,
 400,
 sde.st_polygon ('polygon ((12 3, 12 6, 15 6, 15 3, 12 3))', 4326)
);
INSERT INTO zones (sa_id, usecode, shape) VALUES (
 3,
 400,
 sde.st_polygon ('polygon ((20 0, 20 10, 30 10, 30 0, 20 0))', 4326)
);
INSERT INTO zones (sa_id, usecode, shape) VALUES (
 4,
 402,
 sde.st_polygon ('polygon ((40 0, 40 10, 50 10, 50 0, 40 0))', 4326)
);
INSERT INTO study1 (code, shape) VALUES (
 400,
 sde.st_polygon ('polygon ((-1 -1, -1 11, 11 11, 19 11, 31 11, 31 -1, 19 -1, 11 -1, -1
-1))', 4326)
);
INSERT INTO study1 (code, shape) VALUES (
 402,
 sde.st_polygon ('polygon ((39 -1, 39 11, 51 11, 51 -1, 39 -1))', 4326)
);
```

```
--Oracle SELECT statement without units
SELECT UNIQUE s.code, z.sa_id, sde.st_distance(z.shape, sde.st_boundary(s.shape))
DISTANCE
 FROM zones z, study1 s
 WHERE z.usecode = s.code AND s.code = 400
 ORDER BY DISTANCE;
CODE       SA_ID      DISTANCE
---------- ---------- ----------
400                       1                                                1
400                              3                                         1
400                              3                                         4
--PostgreSQL SELECT statement without units
SELECT DISTINCT s.code, z.sa_id, sde.st_distance(z.shape, sde.st_boundary(s.shape))
 AS Distance
 FROM zones z, study1 s
 WHERE z.usecode = s.code AND s.code = 400
 ORDER BY Distance;
code             sa_id                distance
400            1                                  1
```

```
400            3                                    1
400                   2                                    4
--Oracle SELECT statement with values returned in kilometers
SELECT UNIQUE s.code, z.sa_id, sde.st_distance(z.shape, sde.st_boundary(s.shape),
'kilometer') DISTANCE
 FROM zones z, study1 s
 WHERE z.usecode = s.code AND s.code = 400
 ORDER BY DISTANCE;
      CODE      SA_ID   DISTANCE
---------- ---------- ----------
       400          1 109.639196
       400          3 109.639196
       400          2 442.300258
--PostgreSQL SELECT statement with values returned in kilometers
SELECT DISTINCT s.code, z.sa_id, sde.st_distance(z.shape, sde.st_boundary(s.shape),
'kilometer')
 AS Distance
 FROM zones z, study1 s
 WHERE z.usecode = s.code AND s.code = 400
 ORDER BY Distance;
code             sa_id                distance
400       1      109.63919620267
400       3      109.63919620267
400       2      442.300258454087
```

SQLite

```
--Create tables, add geometry columns, and insert values.
CREATE TABLE zones (
 sa_id integer primary key autoincrement not null,
 usecode integer
);
SELECT AddGeometryColumn (
 NULL,
 'zones',
 'shape',
 4326,
 'polygon',
 'xy',
 'null'
);
CREATE TABLE study1 (
 code integer unique
);
SELECT AddGeometryColumn (
 NULL,
 'study1',
 'shape',
 4326,
 'polygon',
 'xy',
 'null'
);
INSERT INTO zones (usecode, shape) VALUES (
 400,
 st_polygon ('polygon ((0 0, 0 10, 10 10, 10 0, 0 0))', 4326)
);
INSERT INTO zones (usecode, shape) VALUES (
 400,
 st_polygon ('polygon ((12 3, 12 6, 15 6, 15 3, 12 3))', 4326)
);
```

```
INSERT INTO zones (usecode, shape) VALUES (
 400,
 st_polygon ('polygon ((20 0, 20 10, 30 10, 30 0, 20 0))', 4326)
);
INSERT INTO zones (usecode, shape) VALUES (
 402,
 st_polygon ('polygon ((40 0, 40 10, 50 10, 50 0, 40 0))', 4326)
);
INSERT INTO study1 (code, shape) VALUES (
 400,
 st_polygon ('polygon ((-1 -1, -1 11, 11 11, 19 11, 31 11, 31 -1, 19 -1, 11 -1, -1
-1))', 4326)
);
INSERT INTO study1 (code, shape) VALUES (
 402,
 st_polygon ('polygon ((39 -1, 39 11, 51 11, 51 -1, 39 -1))', 4326)
);
```

```
--SQLite SELECT statement without units
SELECT DISTINCT s.code, z.sa_id, st_distance(z.shape, st_boundary(s.shape))
 AS "Distance(km)"
 FROM zones z, study1 s
 WHERE z.usecode = s.code AND s.code = 400
 ORDER BY "Distance(km)";
code             sa_id                distance
400            1                                  1
400            3                                  1
400                  2                                    4
--SQLite SELECT statement with units
SELECT DISTINCT s.code, z.sa_id, st_distance(z.shape, st_boundary(s.shape),
"kilometer")
 AS "Distance(km)"
 FROM zones z, study1 s
 WHERE z.usecode = s.code AND s.code = 400
 ORDER BY "Distance(km)";
code                 sa_id      Distance(km)
400                      1                                        109.63919620267
400                              3
109.63919620267
400                              2
442.30025845408
```

ST_DWithin

定義

ST_DWithin は入力として 2 つのジオメトリを取得し、互いが指定した距離内にある場合に true を返し、そうでない場合に false を返します。 ジオメトリの空間参照系は、指定した距離に適用される計測単位を決定します。 そのため、ST_DWithin に指定するジオメトリは、同じ座標投影と空間参照 ID (SRID) を使用する必要があります。

構文

Oracle および PostgreSQL

```
sde.st_dwithin (st_geometry geometry1, st_geometry geometry2, double_precision distance);
```

SQLite

```
st_dwithin (geometryblob geometry1, geometryblob geometry2, double_precision distance);
```

戻り値のタイプ

Boolean

例

次の例では、2 つのテーブルを作成し、それらにフィーチャを挿入します。 次に、ST_DWithin 関数を 2 つの異なる SELECT ステートメントで使用します。1 つは最初のテーブル内のポイントが 2 つ目のテーブル内のポリゴンから 100 メートル以内にあるかどうかを判定するため、もう 1 つは互いに 300 メートル以内にあるフィーチャを判定するために使用します。

Oracle

```
--Create table to store points.
CREATE TABLE dwithin_test_pt (id INT, geom sde.st_geometry);

--Create table to store polygons.
CREATE TABLE dwithin_test_poly (id INT, geom sde.st_geometry);

--Insert features into each table.

INSERT INTO dwithin_test_pt
 VALUES
 (
  1,
  sde.st_geometry('point (1 2)', 4326)
 )
;

INSERT INTO dwithin_test_pt
 VALUES
 (
  2,
  sde.st_geometry('point (10.02 20.01)', 4326)
 )
;

INSERT INTO dwithin_test_poly
 VALUES
```

```
 (
  1,
  sde.st_geometry('polygon ((10.02 20.01, 11.92 35.64, 25.02 34.15, 19.15 33.94, 10.02
20.01))', 4326)
 )
;

INSERT INTO dwithin_test_poly
 VALUES
 (
  2,
  sde.st_geometry('polygon ((101.02 200.01, 111.92 350.64, 250.02 340.15, 190.15
330.94, 101.02 200.01))', 4326)
 )
;
```

次に、ST_DWithin を使用して、各テーブル内で互いに 100 メートル以内にあるフィーチャとそうでないフィーチャを判定します。 このステートメントには、フィーチャ間の実際の距離を表示する ST_Distance 関数が含まれています。

```
--Determine which features in the point table are within 100 meters of the features in
the polygon table.
SELECT pt.id, poly.id, sde.st_distance(pt.geom, poly.geom) distance_meters,
sde.st_dwithin(pt.geom, poly.geom, 100) DWithin
 FROM dwithin_test_pt pt, dwithin_test_poly poly;
```

ステートメントは、以下を返します。

```
ID     ID   DISTANCE_METERS   DWITHIN
1      1         20.1425048         1
1      2          221.83769         0
2      1                  0         1
2      2         201.695315         0
```

次の例では、ST_DWithin を使用して、互いに 300 メートル以内にあるフィーチャを検索します。

```
--Determine which features in the point table are within 300 meters of the features in
the polygon table.
SELECT pt.id, poly.id, sde.st_distance(pt.geom, poly.geom) distance_meters,
sde.st_dwithin(pt.geom, poly.geom, 300) DWithin
 FROM dwithin_test_pt pt, dwithin_test_poly poly;
```

Oracle でデータに対して実行した場合、2 つ目の SELECT ステートメントは以下を返します。

```
ID     ID   DISTANCE_METERS    DWITHIN
1      1         20.1425048          1
1      2          221.83769          1
2      1                  0          1
2      2         201.695315          1
```

PostgreSQL

```
--Create table to store points.
CREATE TABLE dwithin_test_pt (id INT, geom sde.st_geometry);

--Create table to store polygons.
CREATE TABLE dwithin_test_poly (id INT, geom sde.st_geometry);

--Insert features into each table.

INSERT INTO dwithin_test_pt
 VALUES
 (
  1,
  sde.st_geometry('point (1 2)', 4326)
 )
;

INSERT INTO dwithin_test_pt
 VALUES
 (
  2,
  sde.st_geometry('point (10.02 20.01)', 4326)
 )
;

INSERT INTO dwithin_test_poly
 VALUES
 (
  1,
  sde.st_geometry('polygon ((10.02 20.01, 11.92 35.64, 25.02 34.15, 19.15 33.94, 10.02
20.01))', 4326)
 )
;

INSERT INTO dwithin_test_poly
 VALUES
 (
  2,
  sde.st_geometry('polygon ((101.02 200.01, 111.92 350.64, 250.02 340.15, 190.15
330.94, 101.02 200.01))', 4326)
 )
;
```

次に、ST_DWithin を使用して、各テーブル内で互いに 100 メートル以内にあるフィーチャとそうでないフィーチャを判定します。 このステートメントには、フィーチャ間の実際の距離を表示する ST_Distance 関数が含まれています。

```
--Determine which features in the point table are within 100 meters of the features in
the polygon table.
SELECT pt.id, poly.id, sde.st_distance(pt.geom, poly.geom) distance_meters,
sde.st_dwithin(pt.geom, poly.geom, 100) DWithin
 FROM dwithin_test_pt pt, dwithin_test_poly poly;
```

ステートメントは、以下を返します。

```
id     id     distance_meters        dwithin
1      1      20.1425048094819       t
```

```
1      2      221.837689538996       f
2      1      0                      t
2      2      201.69531476958        f
```

次の例では、ST_DWithin を使用して、互いに 300 メートル以内にあるフィーチャを検索します。

```
--Determine which features in the point table are within 300 meters of the features in
the polygon table.
SELECT pt.id, poly.id, sde.st_distance(pt.geom, poly.geom) distance_meters,
sde.st_dwithin(pt.geom, poly.geom, 300) DWithin
 FROM dwithin_test_pt pt, dwithin_test_poly poly;
```

この 2 つ目の選択ステートメントは、以下を返します。

```
id     id     distance_meters        dwithin
1      1      20.1425048094819       t
1      2      221.837689538996       t
2      1      0                      t
2      2      201.69531476958        t
```

SQLite

```
--Create table to store points.
CREATE TABLE dwithin_test_pt (
 id integer not null
);

SELECT AddGeometryColumn(
 NULL,
 'dwithin_test_pt',
 'geom',
 4326,
 'point',
 'xy',
 'null'
);

--Create table to store polygons.
CREATE TABLE dwithin_test_poly (
 id integer not null
);

SELECT AddGeometryColumn(
 NULL,
 'dwithin_test_poly',
 'geom',
 4326,
 'polygon',
 'xy',
 'null'
);

--Insert features into each table.

INSERT INTO dwithin_test_pt
 VALUES
 (
```

```
  1,
  st_geometry('point (1 2)', 4326)
 )
;

INSERT INTO dwithin_test_pt
 VALUES
 (
  2,
  st_geometry('point (10.02 20.01)', 4326)
 )
;

INSERT INTO dwithin_test_poly
 VALUES
 (
  1,
  st_geometry('polygon ((10.02 20.01, 11.92 35.64, 25.02 34.15, 19.15 33.94, 10.02
20.01))', 4326)
 )
;

INSERT INTO dwithin_test_poly
 VALUES
 (
  2,
  st_geometry('polygon ((101.02 200.01, 111.92 350.64, 250.02 340.15, 190.15 330.94,
101.02 200.01))', 4326)
 )
;
```

次に、ST_DWithin を使用して、各テーブル内で互いに 100 メートル以内にあるフィーチャとそうでないフィーチャを判定します。 このステートメントには、フィーチャ間の実際の距離を表示する ST_Distance 関数が含まれています。

```
--Determine which features in the point table are within 100 meters of the features in
the polygon table.
SELECT pt.id, poly.id, st_distance(pt.geom, poly.geom) distance_meters,
st_dwithin(pt.geom, poly.geom, 100) DWithin
 FROM dwithin_test_pt pt, dwithin_test_poly poly;
```

ステートメントは、以下を返します。

```
1|1|20.1425048094819|1
1|2|221.837689538996|0
2|1|0.0|1
2|2|201.69531476958|0
```

次の例では、ST_DWithin を使用して、互いに 300 メートル以内にあるフィーチャを検索します。

```
--Determine which features in the point table are within 300 meters of the features in
the polygon table.
SELECT pt.id, poly.id, st_distance(pt.geom, poly.geom) distance_meters,
st_dwithin(pt.geom, poly.geom, 300) DWithin
 FROM dwithin_test_pt pt, dwithin_test_poly poly;
```

この 2 つ目の選択ステートメントは、以下を返します。

```
1|1|20.1425048094819|1
1|2|221.837689538996|1
2|1|0.0|1
2|2|201.69531476958|1
```

ST_EndPoint

定義

ST_EndPoint は、ラインストリングの最後のポイントを返します。

構文

Oracle および PostgreSQL

```
sde.st_endpoint (line1 sde.st_geometry)
```

SQLite

```
st_endpoint (line1 geometryblob)
```

戻り値のタイプ

ST_Point

例

endpoint_test テーブルは、各行を一意に識別する gid 整数列と、ラインストリングを格納する ln1 ST_LineString 列を持ちます。

INSERT ステートメントは、ラインストリングを endpoint_test テーブルに挿入します。最初のラインストリングには Z 座標またはメジャー値がありませんが、2 番目のラインストリングにはあります。

クエリは、gid 列と、ST_EndPoint 関数によって生成された ST_Point ジオメトリをリストします。

Oracle

```
--Create table and insert values.
CREATE TABLE endpoint_test (
 gid integer,
 ln1 sde.st_geometry
);

INSERT INTO ENDPOINT_TEST VALUES (
 1,
 sde.st_linefromtext ('linestring (10.02 20.01, 23.73 21.92, 30.10 40.23)', 4326)
);

INSERT INTO ENDPOINT_TEST VALUES (
 2,
 sde.st_linefromtext ('linestring zm(10.02 20.01 5.0 7.0, 23.73 21.92 6.5 7.1,30.10
40.23 6.9 7.2)', 4326)
);
```

```
--Find the end point of each line.
SELECT gid, sde.st_astext (sde.st_endpoint (ln1)) Endpoint
 FROM ENDPOINT_TEST;
```

```
GID         Endpoint

 1    POINT (30.10 40.23)
 2    POINT ZM (30.10 40.23 6.9 7.2)
```

PostgreSQL

```
--Create table and insert values.
CREATE TABLE endpoint_test (
 gid integer,
 ln1 sde.st_geometry
);

INSERT INTO endpoint_test VALUES (
 1,
st_linestring ('linestring (10.02 20.01, 23.73 21.92, 30.10 40.23)', 4326)
);

INSERT INTO endpoint_test VALUES (
 2,
 st_linestring ('linestring zm(10.02 20.01 5.0 7.0, 23.73 21.92 6.5 7.1,30.10 40.23 6.9
7.2)', 4326)
);
```

```
--Find the end point of each line.
SELECT gid, st_astext (st_endpoint (ln1))
 AS endpoint
 FROM endpoint_test;

gid          endpoint

 1    POINT (30.10 40.23)
 2    POINT ZM (30.10 40.23 6.9 7.2)
```

SQLite

```
--Create table, add geometry column, and insert values.
CREATE TABLE endpoint_test (
 gid integer primary key autoincrement not null
);

SELECT AddGeometryColumn (
 NULL,
 'endpoint_test',
 'ln1',
 4326,
 'linestringzm',
 'xyzm',
 'null'
);

INSERT INTO endpoint_test (ln1) VALUES (
 st_linestring ('linestring (10.02 20.01, 23.73 21.92, 30.10 40.23)', 4326)
);

INSERT INTO endpoint_test (ln1) VALUES (
 st_linestring ('linestring zm(10.02 20.01 5.0 7.0, 23.73 21.92 6.5 7.1,30.10 40.23 6.9
```

```
7.2)', 4326)
);
```

```
--Find the end point of each line.
SELECT gid, st_astext (st_endpoint (ln1))
 AS "endpoint"
 FROM endpoint_test;

gid          endpoint

 1    POINT (30.10000000 40.23000000)
 2    POINT ZM (30.10000000 40.23000000 6.90000000 7.20000000)
```

ST_Entity

定義

ST_Entity は、ジオメトリ オブジェクトの空間エンティティ タイプを返します。空間エンティティ タイプは、ジオメトリ オブジェクトのエンティティ メンバー フィールドに格納された値です。

構文

Oracle と PostgreSQL

```
sde.st_entity (geometry1 sde.st_geometry)
```

SQLite

```
st_entity (geometry1 geometryblob)
```

戻り値のタイプ

次のエンティティ タイプを表す値 (Oracle) または整数 (SQLite および PostgreSQL) を返します。

0	nil シェープ
1	ポイント
2	ライン (スパゲッティ ラインを含む)
4	ラインストリング
8	エリア
257	マルチポイント
258	マルチライン (スパゲッティ ラインを含む)
260	マルチラインストリング
264	マルチエリア

例

次の例は、テーブルを作成して、テーブルにさまざまなジオメトリを挿入します。ST_Entity が実行され、テーブル内の各レコードのジオメトリ サブタイプを返します。

Oracle

```
CREATE TABLE sample_geos (
 id integer,
 geometry sde.st_geometry
);
INSERT INTO sample_geos (id, geometry) VALUES (
 1901,
 sde.st_geometry ('point (1 2)', 4326)
);
INSERT INTO sample_geos (id, geometry) VALUES (
```

```
 1902,
 sde.st_geometry ('linestring (33 2, 34 3, 35 6)', 4326)
);
INSERT INTO sample_geos (id, geometry) VALUES (
 1903,
 sde.st_geometry ('polygon ((3 3, 4 6, 5 3, 3 3))', 4326)
);
SELECT sde.st_entity (geometry) entity, UPPER (sde.st_geometrytype (geometry)) TYPE
 FROM sample_geos;
```

SELECT ステートメントは、以下の値を返します。

```
 ENTITY        TYPE
    1    ST_POINT
    4    ST_LINESTRING
    8    ST_POLYGON
```

PostgreSQL

```
CREATE TABLE sample_geos (
 id integer,
 geometry sde.st_geometry
);
INSERT INTO sample_geos (id, geometry) VALUES (
 1900,
 sde.st_geometry ('Point Empty', 4326)
);
INSERT INTO sample_geos (id, geometry) VALUES (
 1901,
 sde.st_geometry ('point (1 2)', 4326)
);
INSERT INTO sample_geos (id, geometry) VALUES (
 1902,
 sde.st_geometry ('linestring (33 2, 34 3, 35 6)', 4326)
);
INSERT INTO sample_geos (id, geometry) VALUES (
 1903,
 sde.st_geometry ('polygon ((3 3, 4 6, 5 3, 3 3))', 4326)
);
INSERT INTO sde.entity_test (id, geometry) VALUES (
 1904,
 sde.st_geometry ('multipoint (10.01 20.03, 10.52 40.11, 30.29 41.56, 31.78 10.74)',
4326)
);
INSERT INTO sde.entity_test (id, geometry) VALUES (
 1905,
 sde.st_geometry ('multilinestring ((10.01 20.03, 10.52 40.11, 30.29 41.56,31.78
10.74), (20.93 20.81, 21.52 40.10))', 4326)
);
INSERT INTO sde.entity_test (id, geometry) VALUES (
 1906,
 sde.st_geometry ('multipolygon (((3 3, 4 6, 5 3, 3 3), (8 24, 9 25, 1 28, 8 24), (13
33, 7 36, 1 40, 10 43, 13 33)))', 4326)
);
SELECT id AS "id",
 sde.st_entity (geometry) AS "entity",
 sde.st_geometrytype (geometry) AS "geom_type"
 FROM sample_geos;
```

SELECT ステートメントは、以下の値を返します。

```
 id       entity     geom_type
    1900     0          "ST_GEOMETRY"
    1901     1          "ST_POINT"
    1902     4          "ST_LINESTRING"
    1903     8          "ST_POLYGON"
    1904     257        "ST_MULTIPOINT"
    1905     260        "ST_MULTILINESTRING"
    1906     264        "ST_MULTIPOLYGON"
```

SQLite

```
CREATE TABLE sample_geos (
 id integer primary key autoincrement not null
);
SELECT AddGeometryColumn (
 NULL,
 'sample_geos',
 'geometry',
 4326,
 'geometry',
 'xy',
 'null'
);
INSERT INTO sample_geos (geometry) VALUES (
 st_geometry ('point (1 2)', 4326)
);
INSERT INTO sample_geos (geometry) VALUES (
 st_geometry ('linestring (33 2, 34 3, 35 6)', 4326)
);
INSERT INTO sample_geos (geometry) VALUES (
 st_geometry ('polygon ((3 3, 4 6, 5 3, 3 3))', 4326)
);
SELECT st_entity (geometry) AS "entity",
 st_geometrytype (geometry) AS "type"
 FROM sample_geos;
```

SELECT ステートメントは、以下の値を返します。

```
entity    type
1         ST_POINT
4         ST_LINESTRING
8         ST_POLYGON
```

ST_Envelope

定義

ST_Envelope は、ジオメトリ オブジェクトの最小境界四角形をポリゴンとして返します。

📖 **詳細:**
この関数は、ST_Envelope がポリゴンを返すことを定めた Open Geospatial Consortium (OGC) シンプル フィーチャ仕様に準拠します。 ポイント ジオメトリや水平線、垂直線といった特殊なケースを扱うため、ST_Envelope 関数はこれらの形状周辺のポリゴンを返します。これは、ジオメトリの空間参照系の XY 縮尺係数に基づいて計算された、小さなエンベロープ許容値です。 この許容値を最小 x と y から引き、最大 x と y 座標に足して、これらの形状周辺のポリゴンを返します。

構文

Oracle および PostgreSQL

```
sde.st_envelope (geometry1 sde.st_geometry)
```

SQLite

```
st_envelope (geometry1 geometryblob)
```

戻り値のタイプ

Oracle および PostgreSQL

ST_Geometry

SQLite

Geometryblob

例

envelope_test テーブルのジオタイプ列には、g1 列に格納されているジオメトリ サブクラスの名前が格納されます。 INSERT ステートメントは、各ジオメトリ サブクラスを envelope_test テーブルに挿入します。

次に、ST_Envelope 関数を実行し、各ジオメトリ周辺のポリゴン エンベロープを返します。

Oracle

```
--Create table and insert values.
CREATE TABLE envelope_test (
 geotype varchar(20),
 g1 sde.st_geometry
);

INSERT INTO ENVELOPE_TEST VALUES (
'Point',
```

```
sde.st_geometry ('point (-1509734.232 -36684.757)', 102004)
);

INSERT INTO ENVELOPE_TEST VALUES (
'Linestring',
sde.st_geometry ('linestring (-1511144.181 -37680.015, -1509734.232 -38841.149,
-1508656.036 -39753.469)', 102004)
);

INSERT INTO ENVELOPE_TEST VALUES (
'Polygon',
sde.st_geometry ('polygon ((-1506333.768 -36435.943, -1504343.252 -36767.695,
-1502684.489 -35357.747, -1506333.768 -36435.943))', 102004)
);

INSERT INTO ENVELOPE_TEST VALUES (
'Multipoint',
sde.st_geometry ('multipoint ((-1493229.539 -40665.789), (-1494141.859 -40831.665),
(-1495800.622 -42739.242))', 102004)
);

INSERT INTO ENVELOPE_TEST VALUES (
'Multilinestring',
sde.st_geometry ('multilinestring ((-1504757.943 -33201.355, -1507411.964 -35606.561),
(-1502518.613 -38094.706, -1499781.653 -37099.448, -1498952.272 -34694.241))', 102004)
);

INSERT INTO ENVELOPE_TEST VALUES (
'Multipolygon',
sde.st_geometry ('multipolygon (((-1492068.405 -47300.841, -1492814.848 -45725.016,
-1493975.983 -46471.459,
-1493478.354 -47798.47, -1492068.405 -47300.841), (-1497874.076 -48047.284,
-1498537.581 -50618.367, -1497210.571 -50037.8,
-1497874.076 -48047.284)))', 102004)
);
```

```
--Return the polygon envelope around each geometry in well-known text.
SELECT geotype geometry_type,
 sde.st_astext (sde.st_envelope (g1)) envelope
 FROM ENVELOPE_TEST;

GEOMETRY_TYPE    ENVELOPE

Point           |POLYGON (( -1509734.23220000 -36684.75720000, -1509734.23180000
-36684.75720000,
-1509734.23180000 -36684.75680000, -1509734.23220000 -36684.75680000, -1509734.23220000
-36684.75720000))

Linestring      |POLYGON (( -1511144.18100000 -39753.46900000, -1508656.03600000
-39753.46900000,
-1508656.03600000 -37680.01500000, -1511144.18100000 -37680.01500000, -1511144.18100000
-39753.46900000))

Polygon         |POLYGON (( -1506333.76800000 -36767.69500000, -1502684.48900000
-36767.69500000,
-1502684.48900000 -35357.74700000, -1506333.76800000 -35357.74700000, -1506333.76800000
-36767.69500000))

Multipoint      |POLYGON (( -1495800.62200000 -42739.24200000, -1493229.53900000
-42739.24200000,
-1493229.53900000 -40665.78900000, -1495800.62200000 -40665.78900000, -1495800.62200000
```

```
-42739.24200000))

Multilinestring |POLYGON (( -1507411.96400000 -38094.70600000, -1498952.27200000
-38094.70600000,
-1498952.27200000 -33201.35500000, -1507411.96400000 -33201.35500000, -1507411.96400000
-38094.70600000))

Multipolygon    |POLYGON (( -1498537.58100000 -50618.36700000, -1492068.40500000
-50618.36700000,
-1492068.40500000 -45725.01600000, -1498537.58100000 -45725.01600000, -1498537.58100000
-50618.36700000))
```

PostgreSQL

```
--Create table and insert values.
CREATE TABLE envelope_test (
 geotype varchar(20),
 g1 sde.st_geometry
);

INSERT INTO ENVELOPE_TEST VALUES (
'Point',
sde.st_geometry ('point (-1509734.232 -36684.757)', 102004)
);

INSERT INTO ENVELOPE_TEST VALUES (
'Linestring',
sde.st_geometry ('linestring (-1511144.181 -37680.015, -1509734.232 -38841.149,
-1508656.036 -39753.469)', 102004)
);

INSERT INTO ENVELOPE_TEST VALUES (
'Polygon',
sde.st_geometry ('polygon ((-1506333.768 -36435.943, -1504343.252 -36767.695,
-1502684.489 -35357.747, -1506333.768 -36435.943))', 102004)
);

INSERT INTO ENVELOPE_TEST VALUES (
'Multipoint',
sde.st_geometry ('multipoint (-1493229.539 -40665.789, -1494141.859 -40831.665,
-1495800.622 -42739.242)', 102004)
);

INSERT INTO ENVELOPE_TEST VALUES (
'Multilinestring',
sde.st_geometry ('multilinestring ((-1504757.943 -33201.355, -1507411.964 -35606.561),
(-1502518.613 -38094.706, -1499781.653 -37099.448, -1498952.272 -34694.241))', 102004)
);

INSERT INTO ENVELOPE_TEST VALUES (
'Multipolygon',
sde.st_geometry ('multipolygon (((-1492068.405 -47300.841, -1492814.848 -45725.016,
-1493975.983 -46471.459,
-1493478.354 -47798.47, -1492068.405 -47300.841), (-1497874.076 -48047.284,
-1498537.581 -50618.367, -1497210.571 -50037.8,
-1497874.076 -48047.284)))', 102004)
);
```

```
--Return the polygon envelope around each geometry in well-known text.
```

```
SELECT geotype AS geometry_type,
 sde.st_astext (sde.st_envelope (g1)) AS Envelope
 FROM envelope_test;

geometry_type     envelope

"Point"           |"POLYGON (( -1509734.23220000 -36684.75720000, -1509734.23180000
-36684.75720000,
-1509734.23180000 -36684.75680000, -1509734.23220000 -36684.75680000, -1509734.23220000
-36684.75720000))"

"Linestring"      |"POLYGON (( -1511144.18100000 -39753.46900000, -1508656.03600000
-39753.46900000,
-1508656.03600000 -37680.01500000, -1511144.18100000 -37680.01500000, -1511144.18100000
-39753.46900000))"

"Polygon"         |"POLYGON (( -1506333.76800000 -36767.69500000, -1502684.48900000
-36767.69500000,
-1502684.48900000 -35357.74700000, -1506333.76800000 -35357.74700000, -1506333.76800000
-36767.69500000))"

"Multipoint"      |"POLYGON (( -1495800.62200000 -42739.24200000, -1493229.53900000
-42739.24200000,
-1493229.53900000 -40665.78900000, -1495800.62200000 -40665.78900000, -1495800.62200000
-42739.24200000))"

"Multilinestring" |"POLYGON (( -1507411.96400000 -38094.70600000, -1498952.27200000
-38094.70600000,
-1498952.27200000 -33201.35500000, -1507411.96400000 -33201.35500000, -1507411.96400000
-38094.70600000))"

"Multipolygon"    |"POLYGON (( -1498537.58100000 -50618.36700000, -1492068.40500000
-50618.36700000,
-1492068.40500000 -45725.01600000, -1498537.58100000 -45725.01600000, -1498537.58100000
-50618.36700000))"
```

SQLite

```
--Create table and insert values.
CREATE TABLE envelope_test (
 geotype varchar(20)
);

SELECT AddGeometryColumn (
 NULL,
 'envelope_test',
 'g1',
 4326,
 'geometry',
 'xy',
 'null'
);

INSERT INTO ENVELOPE_TEST VALUES (
 'Point',
 st_geometry ('point (-1509734.232 -36684.757)', 102004)
);

INSERT INTO ENVELOPE_TEST VALUES (
 'Linestring',
 st_geometry ('linestring (-1511144.181 -37680.015, -1509734.232 -38841.149,
```

```
-1508656.036 -39753.469)', 102004)
);

INSERT INTO ENVELOPE_TEST VALUES (
 'Polygon',
 st_geometry ('polygon ((-1506333.768 -36435.943, -1504343.252 -36767.695, -1502684.489
-35357.747, -1506333.768 -36435.943))', 102004)
);

INSERT INTO ENVELOPE_TEST VALUES (
 'Multipoint',
 st_geometry ('multipoint ((-1493229.539 -40665.789), (-1494141.859 -40831.665),
(-1495800.622 -42739.242))', 102004)
);

INSERT INTO ENVELOPE_TEST VALUES (
 'Multilinestring',
 st_geometry ('multilinestring ((-1504757.943 -33201.355, -1507411.964 -35606.561),
(-1502518.613 -38094.706, -1499781.653 -37099.448, -1498952.272 -34694.241))', 102004)
);

INSERT INTO ENVELOPE_TEST VALUES (
 'Multipolygon',
 st_geometry ('multipolygon (((-1492068.405 -47300.841, -1492814.848 -45725.016,
-1493975.983 -46471.459,
-1493478.354 -47798.47, -1492068.405 -47300.841), (-1497874.076 -48047.284,
-1498537.581 -50618.367, -1497210.571 -50037.8,
-1497874.076 -48047.284)))', 102004)
);
```

```
--Return the polygon envelope around each geometry in well-known text.
SELECT geotype AS geometry_type,
 st_astext (st_envelope (g1)) AS "Envelope"
 FROM envelope_test;

geometry_type   Envelope

Point           POLYGON (( -1509734.23220000 -36684.75720000, -1509734.23180000
-36684.75720000,
-1509734.23180000 -36684.75680000, -1509734.23220000 -36684.75680000, -1509734.23220000
-36684.75720000))

Linestring      POLYGON (( -1511144.18100000 -39753.46900000, -1508656.03600000
-39753.46900000,
-1508656.03600000 -37680.01500000, -1511144.18100000 -37680.01500000, -1511144.18100000
-39753.46900000))

Polygon         POLYGON (( -1506333.76800000 -36767.69500000, -1502684.48900000
-36767.69500000,
-1502684.48900000 -35357.74700000, -1506333.76800000 -35357.74700000, -1506333.76800000
-36767.69500000))

Multipoint      POLYGON (( -1495800.62200000 -42739.24200000, -1493229.53900000
-42739.24200000,
-1493229.53900000 -40665.78900000, -1495800.62200000 -40665.78900000, -1495800.62200000
-42739.24200000))

Multilinestring POLYGON (( -1507411.96400000 -38094.70600000, -1498952.27200000
-38094.70600000,
-1498952.27200000 -33201.35500000, -1507411.96400000 -33201.35500000, -1507411.96400000
-38094.70600000))
```

```
Multipolygon    POLYGON (( -1498537.58100000 -50618.36700000, -1492068.40500000
-50618.36700000,
-1492068.40500000 -45725.01600000, -1498537.58100000 -45725.01600000, -1498537.58100000
-50618.36700000))
```

ST_EnvIntersects

注意:
Oracle と SQLite のみ

定義

ST_EnvIntersects は、2 つのジオメトリのエンベロープが交差する場合は 1 (TRUE) を返し、それ以外の場合は 0 (FALSE) を返します。

構文

Oracle

```
sde.st_envintersects (geometry1 sde.st_geometry, geometry2 sde.st_geometry)
sde.st_envintersects (geometry1 sde.st_geometry, minx number, miny number, maxx number,
maxy number)
```

SQLite

```
st_envintersects (geometry1 geometryblob, geometry2 geoemtryblob)
st_envintersects (geometry1 geoemtryblob, minx float64, miny float64, maxx float64,
maxy float64)
```

戻り値のタイプ

Boolean

例

この例では、定義済みのポリゴンとエンベロープが交差しているジオメトリを検索しています。

最初の SELECT ステートメントは、2 つのジオメトリのエンベロープおよびジオメトリ自身を比較して、フィーチャまたはエンベロープが交差しているかどうかを確認します。

2 番目の SELECT ステートメントは、エンベロープを使用して、SELECT ステートメントの WHERE 句で渡したエンベロープ内にあるフィーチャを検出します。

Oracle

```
--Define and populate the table.
CREATE TABLE sample_geoms (
 id integer,
 geometry sde.st_geometry);

INSERT INTO SAMPLE_GEOMS (id, geometry) VALUES (
 1,
 sde.st_geometry ('linestring (10 10, 50 50)', 4326)
);

INSERT INTO SAMPLE_GEOMS (id, geometry) VALUES (
 2,
```

```
 sde.st_geometry ('linestring (10 20, 50 60)', 4326)
);
```

```
--Find the intersection of the geometries and the geometries' envelopes.
SELECT a.id, b.id, sde.st_intersects (a.geometry, b.geometry) Intersects,
sde.st_envintersects (a.geometry, b.geometry) Envelope_Intersects
 FROM SAMPLE_GEOMS a, SAMPLE_GEOMS b
 WHERE a.id = 1 AND b.id=2;

ID      ID      INTERSECTS   ENVELOPE_INTERSECTS

1       2       0            1
```

```
--Find the geometries whose envelopes intersect the specified envelope.
SELECT id
 FROM SAMPLE_GEOMS
 WHERE sde.st_envintersects(geometry, 5, 5, 60, 65) = 1;

ID

1
2
```

SQLite

```
--Define and populate the table.
CREATE TABLE sample_geoms (
 id integer primary key autoincrement not null
);

SELECT AddGeometryColumn (
 NULL,
 'sample_geoms',
 'geometry',
 4326,
 'linestring',
 'xy',
 'null'
);

INSERT INTO SAMPLE_GEOMS (geometry) VALUES (
 st_geometry ('linestring (10 10, 50 50)', 4326)
);

INSERT INTO SAMPLE_GEOMS (geometry) VALUES (
 st_geometry ('linestring (10 20, 50 60)', 4326)
);
```

```
--Find the intersection of the geometries and the geometries' envelopes.
SELECT a.id AS aid, b.id AS bid, st_intersects (a.geometry, b.geometry) AS "Intersects",
 st_envintersects (a.geometry, b.geometry) AS "Envelope_Intersects"
 FROM SAMPLE_GEOMS a, SAMPLE_GEOMS b
 WHERE a.id = 1 AND b.id = 2;

aid     bid     Intersects   Envelope_Intersects
```

```
1       2       0            1
```

```
--Find the geometries whose envelopes intersect the specified envelope.
SELECT id
 FROM SAMPLE_GEOMS
 WHERE st_envintersects(geometry, 5, 5, 60, 65) = 1;

ID

1
2
```

ST_Equals

説明

ST_Equals は、2 つのジオメトリを比較し、同じ場合は 1 (Oracle および SQLite) または t (PostgreSQL)、それ以外の場合は 0 (Oracle および SQLite) または f (PostgreSQL) を返します。

構文

Oracle および PostgreSQL

```
sde.st_equals (geometry1 sde.st_geometry, geometry2 sde.st_geometry)
```

SQLite

```
st_equals (geometry1 geometryblob, geometry2 geometryblob)
```

戻り値のタイプ

ブール値

例

GIS 技術者は、studies テーブルの中に重複したデータがあるのではないかと疑っています。懸念を払拭するために、テーブルにクエリを実行して同じシェープのマルチポリゴンがあるかどうかを判定します。

studies テーブルが作成され、次のステートメントによって入力されます。id 列が分析範囲を一意に識別し、シェープ フィールドにその範囲のジオメトリが格納されます。

次に、studies テーブルは、equal 述語によって空間的に自身に結合し、同じ 2 つのマルチポリゴンを見つけた場合は 1 (Oracle および SQLite) または t (PostgreSQL) を返します。s1.id<>s2.id 条件は、ジオメトリを自身と比較しないようにします。

Oracle

```
CREATE TABLE studies (
 id integer unique,
 shape sde.st_geometry
);

INSERT INTO studies (id, shape) VALUES (
 1,
 sde.st_polygon ('polygon ((0 0, 0 10, 10 10, 10 0, 0 0))', 4326)
);

INSERT INTO studies (id, shape) VALUES (
 2,
 sde.st_polygon ('polygon ((20 0, 20 10, 30 10, 30 0, 20 0))', 4326)
);

INSERT INTO studies (id, shape) VALUES (
 3,
 sde.st_polygon ('polygon ((40 0, 40 10, 50 10, 50 0, 40 0))', 4326)
);
```

```
INSERT INTO studies (id, shape) VALUES (
 4,
 sde.st_polygon ('polygon ((0 0, 0 10, 10 10, 10 0, 0 0))', 4326)
);
```

```
SELECT UNIQUE (s1.id), s2.id
 FROM STUDIES s1, STUDIES s2
 WHERE sde.st_equals (s1.shape, s2.shape) = 1
 AND s1.id <> s2.id;

ID     ID

 4       1
 1       4
```

PostgreSQL

```
CREATE TABLE studies (
 id serial,
 shape st_geometry
);

INSERT INTO studies (shape) VALUES (
 st_polygon ('polygon ((0 0, 0 10, 10 10, 10 0, 0 0))', 4326)
);

INSERT INTO studies (shape) VALUES (
 st_polygon ('polygon ((20 0, 20 10, 30 10, 30 0, 20 0))', 4326)
);

INSERT INTO studies (shape) VALUES (
 st_polygon ('polygon ((40 0, 40 10, 50 10, 50 0, 40 0))', 4326)
);

INSERT INTO studies (shape) VALUES (
 st_polygon ('polygon ((0 0, 0 10, 10 10, 10 0, 0 0))', 4326)
);
```

```
SELECT DISTINCT (s1.id), s2.id
 FROM studies s1, studies s2
 WHERE st_equals (s1.shape, s2.shape) = 't'
 AND s1.id <> s2.id;

id    id

 1      4
 4      1
```

SQLite

```
CREATE TABLE studies (
 id integer primary key autoincrement not null
);
```

```
SELECT AddGeometryColumn (
 NULL,
 'studies',
 'shape',
 4326,
 'polygon',
 'xy',
 'null'
);

INSERT INTO studies (shape) VALUES (
 st_polygon ('polygon ((0 0, 0 10, 10 10, 10 0, 0 0))', 4326)
);

INSERT INTO studies (shape) VALUES (
 st_polygon ('polygon ((20 0, 20 10, 30 10, 30 0, 20 0))', 4326)
);

INSERT INTO studies (shape) VALUES (
 st_polygon ('polygon ((40 0, 40 10, 50 10, 50 0, 40 0))', 4326)
);

INSERT INTO studies (shape) VALUES (
 st_polygon ('polygon ((0 0, 0 10, 10 10, 10 0, 0 0))', 4326)
);
```

```
SELECT DISTINCT (s1.id), s2.id
 FROM studies s1, studies s2
 WHERE st_equals (s1.shape, s2.shape) = 1
 AND s1.id <> s2.id;

id    id

 1      4
 4      1
```

ST_Equalsrs

注意:
PostgreSQL のみ

定義

ST_Equalsrs は、2 つの異なるフィーチャクラスの空間参照系が同一であるかどうかを判定します。 空間参照系が同一である場合は t (true) が返されます。 空間参照系が同一でない場合は f (false) が返されます。

構文

```
sde.st_equalsrs (srid1 integer, srid2 integer)
```

戻り値のタイプ

Boolean

例

この例では、各フィーチャクラスの空間参照 ID (SRID) を検出した後、ST_Equalsrs を使用して SRID が同じ空間参照系を表しているかどうかを確認しています。

```
SELECT srid, table_name
FROM sde_layers
WHERE table_name = 'transmains' OR table_name = 'streets';
```

srid	table_name
2	streets
6	transmains

sde_layers クエリの結果

続いて ST_Equalsrs を使用して、これら 2 つの SRID で識別された空間参照系が同一であるかどうかを判定します。

```
SELECT sde.st_equalsrs(2,6) ;

 st_equalsrs

-------------

f

(1 row)
```

ST_ExteriorRing

定義

ST_ExteriorRing は、ポリゴンの外部リングをラインストリングとして返します。

構文

```
sde.st_exteriorring (polygon1 sde.st_geometry)
```

Oracle および PostgreSQL

```
sde.st_exteriorring (polygon1 sde.st_geometry)
```

SQLite

```
st_exteriorring (polygon1 geometryblob)
```

戻り値のタイプ

ST_LineString

例

複数の島で鳥の生息数を調査している鳥類学者は、目的の鳥類の餌場が海岸線に限定されることを知っています。島の扶養能力を計算する上で、鳥類学者は島の周長を知る必要があります。島の中には、非常に広大で複数の湖がある島もあります。しかし、湖の縁は攻撃的な別の鳥類が独占しています。そのため、鳥類学者が必要なのは、島の外部リングの周長だけです。

islands テーブルの ID と name 列は各島を識別します。land ポリゴン列は島のジオメトリを格納します。

ST_ExteriorRing 関数は、各島のポリゴンの外部リングをラインストリングとして抽出します。ラインストリングの長さは、ST_Length 関数によって計算されます。ラインストリングの長さは、SUM 関数によって合計されます。

島の外部リングは、各島が海と共有する生態環境インターフェイスを表します。

Oracle

```
--Create the table and insert two polygons.
CREATE TABLE islands (
 id integer,
 name varchar(32),
 land sde.st_geometry
);

INSERT INTO islands VALUES (
1,
'Bear',
sde.st_polygon ('polygon ((40 120, 90 120, 90 150, 40 150, 40 120),(50 130, 60 130, 60
140, 50 140, 50 130),
(70 130, 80 130, 80 140, 70 140, 70 130))', 4326)
);
```

```
INSERT INTO islands VALUES (
 2,
 'Johnson',
sde.st_polygon ('polygon ((10 10, 50 10, 10 30, 10 10))', 4326)
);
```

```
--Extract the exterior ring from each island and find its length.
SELECT SUM (sde.st_length (sde.st_exteriorring (land)))
 FROM ISLANDS;

SUM(ST_LENGTH(ST_EXTERIORRING(LAND)))

                            264.72136
```

PostgreSQL

```
--Create the table and insert two polygons.
CREATE TABLE islands (
 id serial,
 name varchar(32),
 land sde.st_geometry
);

INSERT INTO islands (name, land) VALUES (
 'Bear',
 sde.st_polygon ('polygon ((40 120, 90 120, 90 150, 40 150, 40 120),(50 130, 60 130, 60
140, 50 140, 50 130),
(70 130, 80 130, 80 140, 70 140, 70 130))', 4326)
);

INSERT INTO islands (name, land) VALUES (
 'Johnson',
 sde.st_polygon ('polygon ((10 10, 50 10, 10 30, 10 10))', 4326)
);
```

```
--Extract the exterior ring from each island and find its length.
SELECT SUM (sde.st_length (sde.st_exteriorring (land)))
 FROM islands;

sum

264.721359549996
```

SQLite

```
--Create the table and insert two polygons.
CREATE TABLE islands (
 id integer primary key autoincrement not null,
 name varchar(32)
);

SELECT AddGeometryColumn (
 NULL,
 'islands',
```

```
 'land',
 4326,
 'polygon',
 'xy',
 'null'
);

INSERT INTO islands (name, land) VALUES (
 'Bear',
 st_polygon ('polygon ((40 120, 90 120, 90 150, 40 150, 40 120),(50 130, 60 130, 60
140, 50 140, 50 130),
(70 130, 80 130, 80 140, 70 140, 70 130))', 4326)
);

INSERT INTO islands (name, land) VALUES (
 'Johnson',
 st_polygon ('polygon ((10 10, 50 10, 10 30, 10 10))', 4326)
);
```

```
--Extract the exterior ring from each island and find its length.
SELECT SUM (st_length (st_exteriorring (land)))
 FROM islands;

sum

264.721359549996
```

ST_GeomCollection

注意:
Oracle と PostgreSQL のみ

定義

ST_GeomCollection は、WKT 表現からジオメトリ コレクションを構築します。

構文

Oracle

```
sde.st_multilinestring (wkt clob, srid integer)
sde.st_multipoint (wkt clob, srid integer)
sde.st_multipolygon (wkt clob, srid integer)
```

PostgreSQL

```
sde.st_multilinestring (wkt, srid integer)
sde.st_multilinestring (esri_shape bytea, srid integer)
sde.st_multipoint (wkt, srid integer)
sde.st_multipoint (esri_shape bytea, srid integer)
sde.st_multipolygon (wkt, srid integer)
sde.st_multipolygon (esri_shape bytea, srid integer)
```

戻り値のタイプ

ST_GeomCollection

例

Oracle

geomcoll_test テーブルを作成し、このテーブルにジオメトリを挿入します。

```
CREATE TABLE geomcoll_test (id integer, geometry sde.st_geometry);

INSERT INTO geomcoll_test (id, geometry) VALUES (
1901,
sde.st_multipoint ('multipoint ((1 2), (4 3), (5 6))', 0)
);

INSERT INTO geomcoll_test (id, geometry) VALUES (
1902,
sde.st_multilinestring ('multilinestring ((33 2, 34 3, 35 6),
(28 4, 29 5, 31 8, 43 12), (39 3, 37 4, 36 7))', 0)
);

INSERT INTO geomcoll_test (id, geometry) VALUES (
1903,
sde.st_multipolygon ('multipolygon (((3 3, 4 6, 5 3, 3 3),
(8 24, 9 25, 1 28, 8 24), (13 33, 7 36, 1 40, 10 43, 13 33)))', 0)
);
```

geomcoll_test テーブルからジオメトリ コレクションを選択します。

```
SELECT id, sde.st_astext (geometry) Geomcollection
FROM GEOMCOLL_TEST;

        ID     GEOMCOLLECTION

      1901     MULTIPOINT ((1.00000000 2.00000000), (4.00000000 3.00000000),
                      (5.00000000 6.00000000))

      1902     MULTILINESTRING ((33.00000000 2.00000000, 34.00000000
                  3.00000000, 35.00000000 6.00000000),(28.00000000 4.00000000,
                  29.00000000 5.00000000, 31.00000000 8.00000000, 43.00000000
                   12.00000000),(39.00000000 3.00000000, 37.00000000
                   4.00000000, 36.00000000 7.00000000))

      1903     MULTIPOLYGON (((13.00000000 33.00000000, 10.00000000
                  43.00000000, 1.00000000 40.00000000, 7.00000000 36.00000000,
                   13.00000000 33.00000000)),((8.00000000 24.00000000,
                   9.00000000 25.00000000, 1.00000000 28.00000000, 8.00000000
                   24.00000000)), ((3.00000000 3.00000000,5.00000000
                   3.00000000, 4.00000000 6.00000000,3.00000000 3.00000000)))
```

PostgreSQL

geomcoll_test テーブルを作成し、このテーブルにジオメトリを挿入します。

```
CREATE TABLE geomcoll_test (id integer, geometry sde.st_geometry);

INSERT INTO geomcoll_test (id, geometry) VALUES (
1901,
sde.st_multipoint ('multipoint (1 2, 4 3, 5 6)', 0)
);

INSERT INTO geomcoll_test (id, geometry) VALUES (
1902,
sde.st_multilinestring ('multilinestring ((33 2, 34 3, 35 6),
(28 4, 29 5, 31 8, 43 12), (39 3, 37 4, 36 7))', 0)
);

INSERT INTO geomcoll_test (id, geometry) VALUES (
1903,
sde.st_multipolygon ('multipolygon (((3 3, 4 6, 5 3, 3 3),
(8 24, 9 25, 1 28, 8 24), (13 33, 7 36, 1 40, 10 43, 13 33)))', 0)
);
```

geomcoll_test テーブルからジオメトリ コレクションを選択します。

```
SELECT id, sde.st_astext (geometry)
AS geomcollection
FROM geomcoll_test;

        id     geomcollection

1901     MULTIPOINT (1 2, 4 3, 5 6)

1902     MULTILINESTRING ((33 2, 34        3, 35 6),(28 4,
29 5, 31 8, 43 12),(39 3, 37 4, 36 7))
```

```
1903     MULTIPOLYGON (((13 33, 10 43, 1 40, 7 36,
13 33)),((8 24, 9 25, 1 28, 8 24)), 3 3, 5 3, 4 6, 3 3)))
```

ST_GeomCollFromWKB

注意:
PostgreSQL のみ

説明

ST_GeomCollFromWKB は、WKB 表現からジオメトリ コレクションを構築します。

構文

```
sde.st_geomcollfromwkb (wkb bytea, srid integer)
```

戻り値のタイプ

ST_GeomCollection

例

注意:
読みやすいように改行が挿入されています。ステートメントをコピーする際は改行を削除してください。

gcoll_test というテーブルを作成します。

```
CREATE TABLE gcoll_test (pkey integer, shape sde.st_geomcollection );
```

テーブルに値を挿入します。

```
INSERT INTO gcoll_test VALUES
(1,
st_geomcollfromwkb (sde.st_asbinary(sde.st_geomcollection
('multipoint(20 20, 30 30, 20 40, 30 50)', 0)), 0));

INSERT INTO gcoll_test VALUES
(2,
sde.st_geomcollfromwkb (sde.st_asbinary(sde.st_geomcollection
('multilinestring ((10.02 20.01, 10.32 23.98, 11.92 25.64),
(9.55 23.75,15.36 30.11),(10 10,20 20,30 30,40 40, 90 90))', 0)), 0));

INSERT INTO gcoll_test VALUES
(3,
sde.st_geomcollfromwkb (sde.st_asbinary(sde.st_geomcollection
('multipolygon(((0 0,4 0,4 4,0 4,0 0),(1 1,2 1,2 2,1 2,1 1)),
((-1 -1,-1 -2,-2 -2,-2 -1,-1 -1)))', 0)), 0));
```

gcoll_test テーブルからジオメトリを選択します。

```
SELECT pkey, sde.st_astext(shape) from gcoll_test;

 pkey   st_astext
```

```
   1          MULTIPOINT ( 20 20, 30 30, 20 40, 30 50)

   3           MULTIPOLYGON ((( 0 0, 4 0, 4 4, 0 4, 0 0), (1 1, 1 2,
2 2, 2 1, 1 1)), ((-1 -1, -2 -1, -2 -2, -1 -2, -1 -1))
```

ST_Geometry

定義

ST_Geometry は、WKT 表現からジオメトリを構築します。

注意:
ArcGIS で使用する空間テーブルを作成する場合は、ST_Geometry サブタイプを指定するよりも、ジオメトリ スーパータイプ (たとえば、ST_Geometry) として列を作成することをお勧めします。

構文

Oracle

- ラインストリング、ポリゴン、およびポイント

```
sde.st_geometry (wkt clob, srid integer)
```

- 最適化ポイント (これは extproc エージェントを起動しないため、より高速にクエリを処理します)

```
sde.st_geometry (x, y, z, m, srid)
```

多数のポイント データをバッチで挿入する場合は、最適化ポイントの構築を使用します。

- パラメトリックの円

```
sde.st_geometry (x, y, z, m, radius, number_of_points, srid)
```

- パラメトリックの楕円

```
sde.st_geometry (x, y, z, m, semi_major_axis, semi_minor_axis, angle,
number_of_points, srid)
```

- パラメトリックの扇形

```
sde.st_geometry (x, y, z, m, startangle, endangle, outerradius, innerradius,
number_of_points, srid)
```

PostgreSQL

- ラインストリング、ポリゴン、およびポイント

```
sde.st_geometry (wkt, srid integer)
sde.st_geometry (esri_shape bytea, srid integer)
```

- パラメトリックの円

```
sde.st_geometry (x, y, z, m, radius, number_of_points, srid)
```

- パラメトリックの楕円

```
sde.st_geometry (x, y, z, m, semi_major_axis, semi_minor_axis, angle,
number_of_points, srid)
```

- パラメトリックの扇形

```
sde.st_geometry (x, y, z, m, startangle, endangle, outerradius, innerradius,
number_of_points, srid)
```

SQLite

- ラインストリング、ポリゴン、およびポイント

```
st_geometry (text  WKT_string,int32 srid)
```

- パラメトリックの円

```
st_geometry (x, y, z, m, radius, number_of_points, srid)
```

- パラメトリックの楕円

```
st_geometry (x, y, z, m, semi_major_axis, semi_minor_axis, angle_of_rotation,
number_of_points, srid)
```

- パラメトリックの扇形

```
st_geometry (x, y, z, m, start_angle, end_angle, outer_radius, inner_radius,
number_of_points, srid)
```

戻り値のタイプ

Oracle および PostgreSQL

ST_Geometry

SQLite

Geometryblob

例

ポイント、ラインストリング、およびポリゴン フィーチャの作成と検索

この例では、テーブル (geoms) を作成し、ポイント、ラインストリング、およびポリゴンの値を挿入しています。

Oracle

```
CREATE TABLE geoms (
```

```
 id integer,
 geometry sde.st_geometry
);
```

```
INSERT INTO GEOMS (id, geometry) VALUES (
 1901,
 sde.st_geometry ('point (1 2)', 4326)
);

--To insert the same point using optimized point construction:
INSERT INTO GEOMS (id, geometry) VALUES (
 1901,
 sde.st_geometry (1,2,null,null,4326)
);

INSERT INTO GEOMS (id, geometry) VALUES (
 1902,
 sde.st_geometry ('linestring (33 2, 34 3, 35 6)', 4326)
);

INSERT INTO GEOMS (id, geometry) VALUES (
 1903,
 sde.st_geometry ('polygon ((3 3, 4 6, 5 3, 3 3))', 4326)
);
```

PostgreSQL

```
CREATE TABLE geoms (
 id serial,
 geometry sde.st_geometry
);
```

```
INSERT INTO geoms (geometry) VALUES (
 sde.st_geometry ('point (1 2)', 4326)
);

INSERT INTO geoms (geometry) VALUES (
 sde.st_geometry ('linestring (33 2, 34 3, 35 6)', 4326)
);

INSERT INTO geoms (geometry) VALUES (
 sde.st_geometry ('polygon ((3 3, 4 6, 5 3, 3 3))', 4326)
);
```

SQLite

```
CREATE TABLE geoms (
 id integer primary key autoincrement not null
);

SELECT AddGeometryColumn (
 NULL,
 'geoms',
 'geometry',
```

```
 4326,
 'geometry',
 'xy',
 'null'
);
```

```
INSERT INTO geoms (geometry) VALUES (
 st_geometry ('point (1 2)', 4326)
);

INSERT INTO geoms (geometry) VALUES (
 st_geometry ('linestring (33 2, 34 3, 35 6)', 4326)
);

INSERT INTO geoms (geometry) VALUES (
 st_geometry ('polygon ((3 3, 4 6, 5 3, 3 3))', 4326)
);
```

パラメトリックの円の作成と検索

radii という名前のテーブルを作成し、円を挿入します。

Oracle

```
CREATE TABLE radii (
 id integer,
 geometry sde.st_geometry
);
```

```
INSERT INTO RADII (id, geometry) VALUES (
 1904,
 sde.st_geometry (10,10,NULL,NULL,25,50,4326)
);

INSERT INTO RADII (id, geometry) VALUES (
 1905,
 sde.st_geometry (5,15,NULL,NULL,10,20,4326)
);
```

PostgreSQL

```
CREATE TABLE radii (
 id serial,
 geometry sde.st_geometry
);
```

```
INSERT INTO radii (geometry) VALUES (
 sde.st_geometry (10,10,NULL,NULL,25,50,4326)
);

INSERT INTO radii (geometry) VALUES (
 sde.st_geometry (5,15,NULL,20,10,30,4326)
```

```
);
```

SQLite

```
CREATE TABLE radii (
 id integer primary key autoincrement not null
);

SELECT AddGeometryColumn (
 NULL,
 'radii',
 'geometry',
 4326,
 'geometry',
 'xy',
 'null'
);
```

```
INSERT INTO radii (geometry) VALUES (
 st_geometry (10,10,NULL,NULL,25,50,4326)
);

INSERT INTO radii (geometry) VALUES (
 st_geometry (5,15,NULL,20,10,30,4326)
);
```

パラメトリックの楕円の作成と検索

track という名前のテーブルを作成し、楕円を挿入します。

Oracle

```
CREATE TABLE track (
 id integer,
 geometry sde.st_geometry
);
```

```
INSERT INTO TRACK (id, geometry) VALUES (
 1907,
 sde.st_geometry (0,0,NULL,NULL,10,5,0,50,4326)
);

INSERT INTO TRACK (id, geometry) VALUES (
 1908,
 sde.st_geometry (4,19,10,20,10,5,0,40,4326)
);
```

PostgreSQL

```
CREATE TABLE track (
 id serial,
 geometry sde.st_geometry
```

```
);
```

```
INSERT INTO track (geometry) VALUES (
 sde.st_geometry (0,0,NULL,NULL,10,5,0,50,4326)
);

INSERT INTO track (geometry) VALUES (
 sde.st_geometry (4,19,10,20,10,5,0,40,4326)
);
```

SQLite

```
CREATE TABLE track (
 id integer primary key autoincrement not null
);

SELECT AddGeometryColumn (
 NULL,
 'track',
 'geometry',
 4326,
 'geometry',
 'xy',
 'null'
);
```

```
INSERT INTO track (geometry) VALUES (
 st_geometry (0,0,NULL,NULL,10,5,0,50,4326)
);

INSERT INTO track (geometry) VALUES (
 st_geometry (4,19,10,20,10,5,0,40,4326)
);
```

パラメトリックの扇形の作成と検索

pwedge という名前のテーブルを作成し、扇形を挿入します。

Oracle

```
CREATE TABLE pwedge (
 id integer,
 label varchar2(8),
 shape sde.st_geometry
);
```

```
INSERT INTO PWEDGE (id, label, shape) VALUES (
 1,
 'Wedge1',
 sde.st_geometry (10,30,NULL,NULL,45,145,5,2,60,4326)
);
```

PostgreSQL

```
CREATE TABLE pwedge (
 id serial,
 label varchar(8),
 shape sde.st_geometry
);
```

```
INSERT INTO pwedge (label, shape) VALUES (
 'Wedge',
 sde.st_geometry(10,30,NULL,NULL,45,145,5,2,60,4326)
);
```

SQLite

```
CREATE TABLE pwedge (
 id integer primary key autoincrement not null,
 label varchar(8)
);

SELECT AddGeometryColumn (
 NULL,
 'pwedge',
 'shape',
 4326,
 'geometry',
 'xy',
 'null'
);
```

```
INSERT INTO pwedge (label, shape) VALUES (
 'Wedge',
 st_geometry(10,30,NULL,NULL,45,145,5,2,60,4326)
);
```

ST_GeometryN

定義

ST_GeometryN は、コレクションと整数インデックスを入力として、コレクション内の n 番目の ST_Geometry オブジェクトを返します。

構文

Oracle および PostgreSQL

```
sde.st_geometryn (mpt1 sde.st_multipoint, index integer)
sde.st_geometryn (mln1 sde.st_multilinestring, index integer)
sde.st_geometryn (mpl1 sde.st_multipolygon, index integer)
```

SQLite

```
st_geometryn (mpt1 st_multipoint, index integer)
st_geometryn (mln1 st_multilinestring, index integer)
st_geometryn (mpl1 st_multipolygon, index integer)
```

戻り値のタイプ

Oracle および PostgreSQL

ST_Geometry

SQLite

Geometryblob

例

この例では、マルチポリゴンを作成します。次に、ST_GeometryN を使用して、マルチポリゴンの 2 番目のエレメントをリストします。

Oracle

```
CREATE TABLE districts (
 dist_id integer,
 shape sde.st_multipolygon
);

INSERT INTO DISTRICTS (dist_id, shape) VALUES (
 1,
 sde.st_multipolygon ('multipolygon (((-1 -1, -1 11, 11 11, 11 -1, -1 -1),
(19 -1, 19 11, 29 9, 31 -1, 19 -1), (39 -1, 39 11, 51 11, 51 -1, 39 -1)))', 4326)
);

SELECT sde.st_astext (sde.st_geometryn (shape, 2)) Second_Element
 FROM DISTRICTS;

Second_Element
```

```
POLYGON  ((-1.00000000 -1.00000000, 11.00000000 -1.00000000, 11.0000000 0 11.000
```

PostgreSQL

```
CREATE TABLE districts (
 dist_id serial,
 shape sde.st_geometry
);

INSERT INTO districts (shape) VALUES (
 sde.st_multipolygon ('multipolygon (((-1 -1, -1 11, 11 11, 11 -1, -1 -1),
(19 -1, 19 11, 29 9, 31 -1, 19 -1), (39 -1, 39 11, 51 11, 51 -1, 39 -1)))', 4326)
);

SELECT sde.st_astext (sde.st_geometryn (shape, 2)) AS Second_Element
 FROM districts;

second_element

POLYGON  ((39 -1, 51 -1, 51 11, 39 11, 39 -1))
```

SQLite

```
CREATE TABLE districts (
 dist_id integer primary key autoincrement not null
);

SELECT AddGeometryColumn (
 NULL,
 'districts',
 'shape',
 4326,
 'multipolygon',
 'xy',
 'null'
);

INSERT INTO districts (shape) VALUES (
 st_multipolygon ('multipolygon (((-1 -1, -1 11, 11 11, 11 -1, -1 -1),
(19 -1, 19 11, 29 9, 31 -1, 19 -1), (39 -1, 39 11, 51 11, 51 -1, 39 -1)))', 4326)
);

SELECT st_astext (st_geometryn (shape, 2))
 AS "Second_Element"
 FROM districts;

Second_Element

POLYGON  ((39.00000000 -1.00000000, 51.00000000 -1.00000000, 51.00000000 11.00000000,
39.00000000 11.00000000, 39.00000000 -1.00000000))
```

ST_GeometryType

定義

ST_GeometryType はジオメトリ オブジェクトを入力として、そのジオメトリ タイプ (ポイント、ライン、ポリゴン、マルチポイント) を文字列として返します。

構文

Oracle および PostgreSQL

```
sde.st_geometrytype (g1 sde.st_geometry)
```

SQLite

```
st_geometrytype (g1 geometryblob)
```

戻り値のタイプ

次のいずれかを含む Varchar(32) (Oracle と PostgreSQL) またはテキスト (SQLite)。

- ST_Point
- ST_LineString
- ST_Polygon
- ST_MultiPoint
- ST_MultiLineString
- ST_MultiPolygon

例

geometrytype_test テーブルには、g1 ジオメトリ列が含まれています。

INSERT ステートメントは、各ジオメトリ サブクラスを g1 列に挿入します。

SELECT クエリは、g1 ジオメトリ列に格納されている各サブクラスのジオメトリ タイプをリストします。

Oracle

```
CREATE TABLE geometrytype_test (g1 sde.st_geometry);

INSERT INTO geometrytype_test VALUES (
 sde.st_geometry ('point (10.02 20.01)', 4326)
);

INSERT INTO geometrytype_test VALUES (
 sde.st_geometry ('linestring (10.01 20.01, 10.01 30.01, 10.01 40.01)', 4326)
);

INSERT INTO geometrytype_test VALUES (
 sde.st_geometry ('polygon ((10.02 20.01, 11.92 35.64, 25.02 34.15,
19.15 33.94, 10.02 20.01))', 4326)
```

```
);

INSERT INTO geometrytype_test VALUES (
 sde.st_geometry ('multipoint ((10.02 20.01), (10.32 23.98), (11.92 25.64))', 4326)
);

INSERT INTO geometrytype_test VALUES (
 sde.st_geometry ('multilinestring ((10.02 20.01, 10.32 23.98,
11.92 25.64), (9.55 23.75, 15.36 30.11))', 4326)
);

INSERT INTO geometrytype_test VALUES (
 sde.st_geometry ('multipolygon (((10.02 20.01, 11.92 35.64, 25.02 34.15,
19.15 33.94, 10.02 20.01)), ((51.71 21.73, 73.36 27.04, 71.52 32.87,
52.43 31.90,51.71 21.73)))', 4326)
);
```

```
SELECT UPPER (sde.st_geometrytype (g1)) Geometry_type
FROM GEOMETRYTYPE_TEST;

Geometry_type

ST_POINT
ST_LINESTRING
ST_POLYGON
ST_MULTIPOINT
ST_MULTILINESTRING
ST_MULTIPOLYGON
```

PostgreSQL

```
CREATE TABLE geometrytype_test (g1 sde.st_geometry);

INSERT INTO geometrytype_test VALUES (
 sde.st_geometry ('point (10.02 20.01)', 4326)
);

INSERT INTO geometrytype_test VALUES (
 sde.st_geometry ('linestring (10.01 20.01, 10.01 30.01, 10.01 40.01)', 4326)
);

INSERT INTO geometrytype_test VALUES (
 sde.st_geometry ('polygon ((10.02 20.01, 11.92 35.64, 25.02 34.15,
19.15 33.94, 10.02 20.01))', 4326)
);

INSERT INTO geometrytype_test VALUES (
 sde.st_geometry ('multipoint (10.02 20.01, 10.32 23.98, 11.92 25.64)', 4326)
);

INSERT INTO geometrytype_test VALUES (
 sde.st_geometry ('multilinestring ((10.02 20.01, 10.32 23.98,
11.92 25.64), (9.55 23.75, 15.36 30.11))', 4326)
);

INSERT INTO geometrytype_test VALUES (
 sde.st_geometry ('multipolygon (((10.02 20.01, 11.92 35.64, 25.02 34.15,
19.15 33.94, 10.02 20.01)), ((51.71 21.73, 73.36 27.04, 71.52 32.87,
52.43 31.90,51.71 21.73)))', 4326)
```

```
);
```

```
SELECT (sde.st_geometrytype (g1))
 AS Geometry_type
 FROM geometrytype_test;

Geometry_type

ST_POINT
ST_LINESTRING
ST_POLYGON
ST_MULTIPOINT
ST_MULTILINESTRING
ST_MULTIPOLYGON
```

SQLite

```
CREATE TABLE geometrytype_test (id integer primary key autoincrement not null);

SELECT AddGeometryColumn (
 NULL,
 'geometrytype_test',
 'g1',
 4326,
 'geometry',
 'xy',
 'null'
);

INSERT INTO geometrytype_test (g1) VALUES (
 st_geometry ('point (10.02 20.01)', 4326)
);

INSERT INTO geometrytype_test (g1) VALUES (
 st_geometry ('linestring (10.01 20.01, 10.01 30.01, 10.01 40.01)', 4326)
);

INSERT INTO geometrytype_test (g1) VALUES (
 st_geometry ('polygon ((10.02 20.01, 11.92 35.64, 25.02 34.15,
19.15 33.94, 10.02 20.01))', 4326)
);

INSERT INTO geometrytype_test (g1) VALUES (
 st_geometry ('multipoint ((10.02 20.01), (10.32 23.98), (11.92 25.64))', 4326)
);

INSERT INTO geometrytype_test (g1) VALUES (
 st_geometry ('multilinestring ((10.02 20.01, 10.32 23.98,
11.92 25.64), (9.55 23.75, 15.36 30.11))', 4326)
);

INSERT INTO geometrytype_test (g1) VALUES (
 st_geometry ('multipolygon (((10.02 20.01, 11.92 35.64, 25.02 34.15,
19.15 33.94, 10.02 20.01)), ((51.71 21.73, 73.36 27.04, 71.52 32.87,
52.43 31.90,51.71 21.73)))', 4326)
);
```

```
SELECT (st_geometrytype (g1))
 AS "Geometry_type"
 FROM geometrytype_test;

Geometry_type

ST_POINT
ST_LINESTRING
ST_POLYGON
ST_MULTIPOINT
ST_MULTILINESTRING
ST_MULTIPOLYGON
```

ST_GeomFromCollection

注意:
PostgreSQL のみ

定義

ST_GeomFromCollection は一連の st_geometry 行を返します。各行はジオメトリと整数で構成されています。整数はセット内のジオメトリの位置を表します。

ST_GeomFromCollection 関数を使用して、マルチパート ジオメトリ内の個々のジオメトリにアクセスします。入力ジオメトリがコレクションまたはマルチパート ジオメトリ (ST_MultiLineString、ST_MultiPoint、ST_MultiPolygon など) の場合、ST_GeomFromCollection はコレクションの各コンポーネントのレコードを返し、パスはコレクション内のコンポーネントの位置を表します。

シンプル ジオメトリ (ST_Point、ST_LineString、ST_Polygon など) で ST_GeomFromCollection を使用した場合は、ジオメトリは 1 つだけなので、1 つのレコードと空のパスが返されます。

構文

```
sde.st_geomfromcollection (shape sde.st_geometry)
```

ジオメトリのみを返す場合は、`(sde.st_geomfromcollection (shape)).st_geo` を使用します。

ジオメトリの位置のみを返す場合は、`(sde.st_geomfromcollection (shape)).path[1]` を使用します。

戻り値のタイプ

ST_Geometry セット

例

この例では、1 つのフィーチャと 4 パートのシェープを含むマルチライン フィーチャクラス (ghanasharktracks) を作成します。

```
--Create the feature class.
CREATE TABLE ghanasharktracks (objectid integer, shape sde.st_geometry);
--Insert a multiline with four parts using SRID 4326.
INSERT INTO ghanasharktracks VALUES
 (1,
  sde.st_geometry('MULTILINESTRING Z (( 1 1 0, 1 6 0),(1 3 0, 3 3 0),(3 1 0, 3 3 0), (4
1 0, 4 6 0))',
  4326
  )
 );
```

フィールドにデータが含まれていることを確認するには、テーブルを検索します。シェープ フィールド上で ST_AsText を直接使用して、シェープの座標をテキスト表示します。マルチラインストリングのテキスト記述が返されます。

```
--View inserted feature. SELECT gst_orig.objectid, sde.st_astext(gst_orig.shape)
shapetext FROM ghanasharktracks gst_orig;
shapetext
------------------------------
```

```
"MULTILINESTRING Z (( 1.00000000 1.00000000 0.00000000, 1.00000000 6.00000000
0.00000000),(1.00000000 3.00000000 0.00000000, 3.00000000 3.00000000
0.00000000),(3.00000000 1.00000000 0.00000000, 3.00000000 3.00000000 0.00000000),
(4.00000000 1.00000000 0.00000000, 4.00000000 6.00000000 0.00000000))"
```

各ラインストリング ジオメトリを個別に返すには、ST_GeomFromCollection 関数を使用します。ジオメトリをテキスト表示するために、この例では、ST_AsText 関数と ST_GeomFromCollection 関数を使用しています。

```
--Return each linestring in the multilinestring
SELECT sde.st_astext((sde.st_geomfromcollection(gst.shape)).st_geo) shapetext,
((sde.st_geomfromcollection(gst.shape)).path[1]) path FROM ghanasharktracks gst;
shapetext
      path
------------------------------------------------------------------------------------
------------------
"LINESTRING Z ( 1.00000000 1.00000000 0.00000000, 1.00000000 6.00000000
0.00000000)"          1
"LINESTRING Z ( 1.00000000 3.00000000 0.00000000, 3.00000000 3.00000000
0.00000000)"          2
"LINESTRING Z ( 3.00000000 1.00000000 0.00000000, 3.00000000 3.00000000
0.00000000)"          3
"LINESTRING Z ( 4.00000000 1.00000000 0.00000000, 4.00000000 6.00000000
0.00000000)"          4
```

ST_GeomFromText

注意:
Oracle および SQLite でのみ使用されます。PostgreSQL では、ST_Geometry を使用してください。

定義

ST_GeomFromText は、WKT 表現と空間参照 ID を入力として、ジオメトリ オブジェクトを返します。

構文

Oracle

```
sde.st_geomfromtext (wkt clob, srid integer)
```

```
sde.st_geomfromtext (wkt clob)
```

SRID を指定しない場合、空間参照がデフォルトで 4326 に設定されます。

SQLite

```
st_geomfromtext (wkt text, srid int32)
```

```
st_geomfromtext (wkt text)
```

SRID を指定しない場合、空間参照がデフォルトで 4326 に設定されます。

戻り値のタイプ

Oracle

ST_Geometry

SQLite

Geometryblob

例

geometry_test テーブルは、各行を一意に識別する整数の gid 列と、ジオメトリを格納する g1 列を含みます。

INSERT ステートメントは、データを geometry_test テーブルの gid 列と g1 列に挿入します。 ST_GeomFromText 関数は、各ジオメトリのテキスト表現を、対応するインスタンス化可能なサブクラスに変換します。 最後の SELECT ステートメントで、データが g1 列に挿入されたことを確認します。

Oracle

```
CREATE TABLE geometry_test (
 gid smallint unique,
 g1 sde.st_geometry
);
```

```
INSERT INTO GEOMETRY_TEST VALUES (
 1,
 sde.st_geomfromtext ('point (10.02 20.01)', 4326)
);

INSERT INTO GEOMETRY_TEST VALUES (
 2,
 sde.st_geomfromtext('linestring (10.01 20.01, 10.01 30.01, 10.01 40.01)', 4326)
);

INSERT INTO GEOMETRY_TEST VALUES (
 3,
 sde.st_geomfromtext('polygon ((10.02 20.01, 11.92 35.64, 25.02 34.15,
19.15 33.94, 10.02 20.01))', 4326)
);

INSERT INTO GEOMETRY_TEST VALUES (
 4,
 sde.st_geomfromtext('multipoint ((10.02 20.01), (10.32 23.98), (11.92 25.64))', 4326)
);

INSERT INTO GEOMETRY_TEST VALUES (
 5,
 sde.st_geomfromtext ('multilinestring ((10.02 20.01, 10.32 23.98,
11.92 25.64), (9.55 23.75, 15.36 30.11))', 4326)
);

INSERT INTO GEOMETRY_TEST VALUES (
 6,
 sde.st_geomfromtext ('multipolygon (((10.02 20.01, 11.92 35.64,
25.02 34.15, 19.15 33.94, 10.02 20.01), (51.71 21.73, 73.36 27.04,
71.52 32.87, 52.43 31.90, 51.71 21.73)))', 4326)
);

SELECT sde.st_astext(g1)
 FROM GEOMETRY_TEST;

POINT ( 10.02000000 20.01000000)
LINESTRING ( 10.01000000 20.01000000, 10.01000000 30.01000000, 10.01000000 40.01000000)
POLYGON (( 10.02000000 20.01000000, 19.15000000 33.94000000, 25.02000000 34.15000000,
11.92000000 35.64000000, 10.02000000 20.01000000))
MULTIPOINT (( 10.02000000 20.01000000), (10.32000000 23.98000000), (11.92000000
25.64000000))
MULTILINESTRING (( 10.02000000 20.01000000, 10.32000000 23.98000000, 11.92000000
25.64000000),( 9.55000000 23.75000000, 15.36000000 30.11000000))
MULTIPOLYGON ((( 51.71000000 21.73000000, 73.36000000 27.04000000, 71.52000000
32.87000000, 52.43000000 31.90000000, 51.71000000 21.73000000)),(( 10.02000000
20.01000000, 19.15000000 33.94000000, 25.02000000 34.15000000, 11.92000000 35.64000000,
10.02000000 20.01000000)))
```

SQLite

```
CREATE TABLE geometry_test (
 gid integer primary key autoincrement not null
);

SELECT AddGeometryColumn (
 NULL,
 'geometry_test',
 'g1',
 4326,
 'geometry',
 'xy',
 'null'
);
```

```
INSERT INTO GEOMETRY_TEST (g1) VALUES (
 st_geomfromtext ('point (10.02 20.01)', 4326)
);

INSERT INTO GEOMETRY_TEST (g1) VALUES (
 st_geomfromtext('linestring (10.01 20.01, 10.01 30.01, 10.01 40.01)', 4326)
);

INSERT INTO GEOMETRY_TEST (g1) VALUES (
 st_geomfromtext('polygon ((10.02 20.01, 11.92 35.64, 25.02 34.15,
19.15 33.94, 10.02 20.01))', 4326)
);

INSERT INTO GEOMETRY_TEST (g1) VALUES (
 st_geomfromtext('multipoint ((10.02 20.01), (10.32 23.98), (11.92 25.64))', 4326)
);

INSERT INTO GEOMETRY_TEST (g1) VALUES (
 st_geomfromtext ('multilinestring ((10.02 20.01, 10.32 23.98,
11.92 25.64), (9.55 23.75, 15.36 30.11))', 4326)
);

INSERT INTO GEOMETRY_TEST (g1) VALUES (
 st_geomfromtext ('multipolygon (((10.02 20.01, 11.92 35.64,
25.02 34.15, 19.15 33.94, 10.02 20.01), (51.71 21.73, 73.36 27.04,
71.52 32.87, 52.43 31.90, 51.71 21.73)))', 4326)
);

SELECT st_astext(g1)
 FROM geometry_test;

POINT (10.02000000 20.01000000)
LINESTRING (10.01000000 20.01000000, 10.01000000 30.01000000, 10.01000000 40.01000000)
POLYGON ((10.02000000 20.01000000, 19.15000000 33.94000000, 25.02000000 34.15000000,
11.92000000 35.64000000, 10.02000000 20.01000000))
MULTIPOINT ((10.02000000 20.01000000), (10.32000000 23.98000000), (11.92000000
25.64000000))
MULTILINESTRING ((10.02000000 20.01000000, 10.32000000 23.98000000, 11.92000000
25.64000000),(9.55000000 23.75000000, 15.36000000 30.11000000))
MULTIPOLYGON (((51.71000000 21.73000000, 73.36000000 27.04000000, 71.52000000
32.87000000, 52.43000000 31.90000000, 51.71000000 21.73000000)),((10.02000000
20.01000000, 19.15000000 33.94000000, 25.02000000 34.15000000, 11.92000000 35.64000000,
10.02000000 20.01000000)))
```

ST_GeomFromWKB

定義

ST_GeomFromWKB は、WKB 表現と空間参照 ID を入力として、ジオメトリ オブジェクトを返します。

構文

Oracle

```
sde.st_geomfromwkb (wkb blob, srid integer)
```

```
sde.st_geomfromwkb (wkb blob)
```

SRID を指定しない場合、空間参照がデフォルトで 4326 に設定されます。

PostgreSQL

```
sde.st_geomfromwkb (wkb, srid integer)
```

```
sde.st_geomfromwkb (esri_shape bytea, srid integer)
```

SQLite

```
st_geomfromwkb (wkb blob, srid int32)
```

```
st_geomfromwkb (wkb blob)
```

SRID を指定しない場合、空間参照がデフォルトで 4326 に設定されます。

戻り値のタイプ

Oracle と PostgreSQL

ST_Geometry

SQLite

Geometryblob

例

以下の例では、読みやすいように結果の書式を再設定しています。結果で表示される間隔は、各自のオンライン表示によって異なります。以下のコードは、ST_GeomFromWKB 関数を使用して、WKB ライン表現からラインを作成して挿入する方法を示しています。次の例では、ID および WKB 表現による空間参照系 4326 のジオメトリを含

む sample_gs テーブルに、レコードを挿入しています。

Oracle

```
CREATE TABLE sample_gs (
 id integer,
 geometry sde.st_geometry,
 wkb blob
);
INSERT INTO sample_gs (id, geometry) VALUES (
 1901,
 sde.st_geomfromtext ('point (1 2)', 4326)
);
INSERT INTO sample_gs (id, geometry) VALUES (
 1902,
 sde.st_geomfromtext ('linestring (33 2, 34 3, 35 6)', 4326)
);
INSERT INTO sample_gs (id, geometry) VALUES (
 1903,
 sde.st_geomfromtext ('polygon ((3 3, 4 6, 5 3, 3 3))', 4326)
);

UPDATE sample_gs
 SET wkb = sde.st_asbinary (geometry)
 WHERE id = 1901;
UPDATE sample_gs
 SET wkb = sde.st_asbinary (geometry)
 WHERE id = 1902;
UPDATE sample_gs
 SET wkb = sde.st_asbinary (geometry)
 WHERE id = 1903;
SELECT id, sde.st_astext (sde.st_geomfromwkb (wkb, 4326))
 FROM sample_gs;
ID   GEOMETRY
1901 POINT (1.00000000 2.00000000)
1902 LINESTRING (33.00000000 2.00000000, 34.00000000 3.00000000, 35.00000000
6.00000000)
1903 POLYGON ((3.00000000 3.00000000, 5.00000000 3.00000000, 4.00000000 6.00000000,
3.00000000 3.00000000))
```

PostgreSQL

```
CREATE TABLE sample_gs (
 id integer,
 geometry sde.st_geometry,
 wkb bytea);
INSERT INTO sample_gs (id, geometry) VALUES (
 1901,
 sde.st_geometry ('point (1 2)', 4326)
);
INSERT INTO sample_gs (id, geometry) VALUES (
 1902,
 sde.st_geometry ('linestring (33 2, 34 3, 35 6)', 4326)
);
INSERT INTO sample_gs (id, geometry) VALUES (
 1903,
 sde.st_geometry ('polygon ((3 3, 4 6, 5 3, 3 3))', 4326)
);

UPDATE sample_gs
```

```
 SET wkb = sde.st_asshape (geometry)
 WHERE id = 1901;
UPDATE sample_gs
 SET wkb = sde.st_asshape (geometry)
 WHERE id = 1902;
UPDATE sample_gs
 SET wkb = sde.st_asshape (geometry)
 WHERE id = 1903;
SELECT id, sde.st_astext (sde.st_geomfromshape (wkb, 4326))
 FROM sample_gs;
id   st_astext
1901 POINT (1 2)
1902 LINESTRING (33 2, 34 3, 35 6)
1903 POLYGON ((3 3, 5 3, 4 6, 3 3))
```

SQLite

```
CREATE TABLE sample_gs (
 id integer primary key autoincrement not null,
 wkb blob
);
SELECT AddGeometryColumn (
 NULL,
 'sample_gs',
 'geometry',
 4326,
 'geometry',
 'xy',
 'null'
);
INSERT INTO sample_gs (geometry) VALUES (
 st_geomfromtext ('point (1 2)', 4326)
);
INSERT INTO sample_gs (geometry) VALUES (
 st_geomfromtext ('linestring (33 2, 34 3, 35 6)', 4326)
);
INSERT INTO sample_gs (geometry) VALUES (
 st_geomfromtext ('polygon ((3 3, 4 6, 5 3, 3 3))', 4326)
);

--Replace IDs with actual values.
UPDATE sample_gs
 SET wkb = st_asbinary (geometry)
 WHERE id = 1;
UPDATE sample_gs
 SET wkb = st_asbinary (geometry)
 WHERE id = 2;
UPDATE sample_gs
 SET wkb = st_asbinary (geometry)
 WHERE id = 3;
SELECT id, st_astext (st_geomfromwkb (wkb, 4326))
 FROM sample_gs;
ID   GEOMETRY
1    POINT (1.00000000 2.00000000)
2    LINESTRING (33.00000000 2.00000000, 34.00000000 3.00000000, 35.00000000
6.00000000)
3    POLYGON ((3.00000000 3.00000000, 5.00000000 3.00000000, 4.00000000 6.00000000,
3.00000000 3.00000000))
```

ST_GeoSize

注意:
PostgreSQL のみ

定義

ST_GeoSize は、ST_Geometry オブジェクトを受け取って、そのサイズをバイト単位で返します。

構文

```
st_geosize (st_geometry)
```

戻り値のタイプ

Integer

例

ST_GeomFromWKB の例で作成したフィーチャのサイズを、sample_geometries テーブルの geometry 列を検索して検出します。

```
SELECT st_geosize (geometry)
FROM sample_geometries;

st_geosize
        512
        592
        616
```

ST_InteriorRingN

定義

ST_InteriorRingN は、ポリゴンの n 番目の内部リングを ST_LineString として返します。

リングの順序は事前に定義できません。リングは、幾何学的な向きではなく、内部のジオメトリ検証ルーチンによって定義された規則に従って編成されるためです。インデックスがポリゴンが持つ内部リングの数を超えた場合、NULL 値が返されます。

構文

Oracle

```
sde.st_interiorringn (polygon1 sde.st_polygon, INDEX integer)
```

PostgreSQL

```
sde.st_interiorringn (polygon1 sde.st_polygon, ring_number integer)
```

SQLite

```
st_interiorringn (polygon1 sde.st_polygon, ring_number int32)
```

戻り値のタイプ

ST_LineString

例

sample_polys テーブルを作成し、レコードを追加します。次に、内部リングの ID とジオメトリを選択します。

Oracle

```
CREATE TABLE sample_polys (
 id integer,
 geometry sde.st_geometry
);

INSERT INTO sample_polys VALUES (
 1,
 sde.st_polygon ('polygon ((40 120, 90 120, 90 150, 40 150, 40 120), (50 130, 60 130,
60 140, 50 140, 50 130),
(70 130, 80 130, 80 140, 70 140, 70 130))', 4326)
);
```

```
SELECT id, sde.st_astext (sde.st_interiorringn (geometry, 2)) Interior_Ring
 FROM SAMPLE_POLYS;

ID INTERIOR_RING
```

```
1  LINESTRING (70.00000000 130.00000000, 70.00000000 140.00000000, 80.00000000
140.00000000, 80.00000000 130.00000000, 70.00000000 130.00000000)
```

PostgreSQL

```
CREATE TABLE sample_polys (
 id serial,
 geometry sde.st_geometry
);

INSERT INTO sample_polys (geometry) VALUES (
 sde.st_polygon ('polygon ((40 120, 90 120, 90 150, 40 150, 40 120), (50 130, 60 130,
60 140, 50 140, 50 130),
(70 130, 80 130, 80 140, 70 140, 70 130))', 4326)
);
```

```
SELECT id, sde.st_astext (st_interiorringn (geometry, 2))
 AS Interior_Ring
 FROM sample_polys;

id interior_ring

1  LINESTRING (70 130, 70 140, 80 140, 80 130, 70 130)
```

SQLite

```
CREATE TABLE sample_polys (
 id integer primary key autoincrement not null
);

SELECT AddGeometryColumn (
 NULL,
 'sample_polys',
 'geometry',
 4326,
 'polygon',
 'xy',
 'null'
);

INSERT INTO sample_polys (geometry) VALUES (
 st_polygon ('polygon ((40 120, 90 120, 90 150, 40 150, 40 120), (50 130, 60 130, 60
140, 50 140, 50 130),
(70 130, 80 130, 80 140, 70 140, 70 130))', 4326)
);
```

```
SELECT id, st_astext (st_interiorringn (geometry, 2))
 AS "Interior_Ring"
 FROM sample_polys;

id Interior_Ring

1  LINESTRING (70.00000000 130.00000000, 70.00000000 140.00000000, 80.00000000
140.00000000, 80.00000000 130.00000000, 70.00000000 130.00000000)
```

ST_Intersection

定義

ST_Intersection は、2 つのジオメトリ オブジェクトを入力として、インターセクト セットを二次元のジオメトリ オブジェクトとして返します。

構文

Oracle および PostgreSQL

```
sde.st_intersection (geometry1 sde.st_geometry, geometry2 sde.st_geometry)
```

SQLite

```
st_intersection (geometry1 geometryblob, geometry2 geometryblob)
```

戻り値のタイプ

Oracle および PostgreSQL

ST_Geometry

SQLite

Geometryblob

例

消防部長は、有害廃棄物汚染の可能性がある半径内に入る病院、学校、看護施設の領域を取得する必要があります。

病院、学校、看護施設は、次の CREATE TABLE ステートメントで作成される population テーブルに格納されます。ポリゴンとして定義された shape 列は、各保護区域の外周を格納します。

有害区域は、次の CREATE TABLE ステートメントで作成される waste_sites テーブルに格納されます。ポイントとして定義された site 列は、各有害区域の地理的な中心位置を格納します。

ST_Buffer 関数は、有害廃棄物区域の周囲にバッファーを作成します。ST_Intersection 関数は、バッファーされた有害廃棄物区域と保護区域のインターセクトからポリゴンを作成します。

Oracle

```
CREATE TABLE population (
 id integer,
 shape sde.st_geometry
);

CREATE TABLE waste_sites (
 id integer,
 site sde.st_geometry
);
```

```
INSERT INTO population VALUES (
 1,
 sde.st_geometry ('polygon ((.20 .30, .30 .30, .30 .40, .20 .40, .20 .30))', 4326)
);

INSERT INTO population VALUES (
 2,
 sde.st_geometry ('polygon ((.30 .30, .30 .50, .50 .50, .50 .30, .30 .30))', 4326)
);

INSERT INTO population VALUES (
 3,
 sde.st_geometry ('polygon ((.40 .40, .40 .60, .60 .60, .60 .40, .40 .40))', 4326)
);

INSERT INTO waste_sites VALUES (
 40,
 sde.st_geometry ('point (.60 .60)', 4326)
);

INSERT INTO waste_sites VALUES (
 50,
 sde.st_geometry ('point (.30 .30)', 4326)
);
```

```
SELECT sa.id, sde.st_astext (sde.st_intersection (sde.st_buffer (hs.site, .1),
sa.shape)) Intersection
 FROM population sa, waste_sites hs
 WHERE hs.id = 50
 AND sde.st_astext (sde.st_intersection (sde.st_buffer (hs.site, .01), sa.shape))
 NOT LIKE '%EMPTY%';

  ID  INTERSECTION

1     POLYGON (( 0.29000000 0.30000000, 0.30000000 0.30000000, 0.30000000
0.31000000, 0.29934597 0.30997859, 0.29869474 0.30991445, 0.29804910 0.30980785,
 0.29741181 0.30965926, 0.29678561 0.30946930, 0.29617317 0.30923880, 0.29557711
 0.30896873, 0.29500000 0.30866025, 0.29444430 0.30831470, 0.29391239 0.30793353
, 0.29340654 0.30751840, 0.29292893 0.30707107, 0.29248160 0.30659346, 0.2920664
7 0.30608761, 0.29168530 0.30555570, 0.29133975 0.30500000, 0.29103127 0.3044228
9, 0.29076121 0.30382683, 0.29053070 0.30321440, 0.29034074 0.30258819, 0.290192
15 0.30195090, 0.29008555 0.30130526, 0.29002141 0.30065403, 0.29000000 0.300000
00))

2     POLYGON (( 0.30000000 0.30000000, 0.31000000 0.30000000, 0.30997859
0.30065403, 0.30991445 0.30130526, 0.30980785 0.30195090, 0.30965926 0.30258819,
 0.30946930 0.30321440, 0.30923880 0.30382683, 0.30896873 0.30442289, 0.30866025
 0.30500000, 0.30831470 0.30555570, 0.30793353 0.30608761, 0.30751840 0.30659346
, 0.30707107 0.30707107, 0.30659346 0.30751840, 0.30608761 0.30793353, 0.3055557
0 0.30831470, 0.30500000 0.30866025, 0.30442289 0.30896873, 0.30382683 0.3092388
0, 0.30321440 0.30946930, 0.30258819 0.30965926, 0.30195090 0.30980785, 0.301305
26 0.30991445, 0.30065403 0.30997859, 0.30000000 0.31000000, 0.30000000 0.300000
00))
```

PostgreSQL

```
CREATE TABLE population (
```

```
 id serial,
 shape sde.st_geometry
);

CREATE TABLE waste_sites (
 id serial,
 site sde.st_geometry
);


INSERT INTO population (shape) VALUES (
 sde.st_geometry ('polygon ((.20 .30, .30 .30, .30 .40, .20 .40, .20 .30))', 4326)
);

INSERT INTO population (shape) VALUES (
 sde.st_geometry ('polygon ((.30 .30, .30 .50, .50 .50, .50 .30, .30 .30))', 4326)
);

INSERT INTO population (shape) VALUES (
 sde.st_geometry ('polygon ((.40 .40, .40 .60, .60 .60, .60 .40, .40 .40))', 4326)
);

INSERT INTO waste_sites (site) VALUES (
 sde.st_geometry ('point (.60 .60)', 4326)
);

INSERT INTO waste_sites (site) VALUES (
 sde.st_geometry ('point (.30 .30)', 4326)
);
```

```
--Replace hs.id with ID value of second record in waste_sites table if not 2.
SELECT sa.id, sde.st_astext (sde.st_intersection (sde.st_buffer (hs.site, .01),
sa.shape))
 AS Intersection
 FROM population sa, waste_sites hs
 WHERE hs.id = 2
 AND sde.st_astext (sde.st_intersection (sde.st_buffer (hs.site, .1),
sa.shape))::varchar
 NOT LIKE '%EMPTY%';

  id  intersection

1      POLYGON (( 0.29000000 0.30000000, 0.30000000 0.30000000, 0.30000000
0.31000000, 0.29934597 0.30997859, 0.29869474 0.30991445, 0.29804910 0.30980785,
 0.29741181 0.30965926, 0.29678561 0.30946930, 0.29617317 0.30923880, 0.29557711
 0.30896873, 0.29500000 0.30866025, 0.29444430 0.30831470, 0.29391239 0.30793353
, 0.29340654 0.30751840, 0.29292893 0.30707107, 0.29248160 0.30659346, 0.2920664
7 0.30608761, 0.29168530 0.30555570, 0.29133975 0.30500000, 0.29103127 0.3044228
9, 0.29076121 0.30382683, 0.29053070 0.30321440, 0.29034074 0.30258819, 0.290192
15 0.30195090, 0.29008555 0.30130526, 0.29002141 0.30065403, 0.29000000 0.300000
00))

2      POLYGON (( 0.30000000 0.30000000, 0.31000000 0.30000000, 0.30997859
0.30065403, 0.30991445 0.30130526, 0.30980785 0.30195090, 0.30965926 0.30258819,
 0.30946930 0.30321440, 0.30923880 0.30382683, 0.30896873 0.30442289, 0.30866025
 0.30500000, 0.30831470 0.30555570, 0.30793353 0.30608761, 0.30751840 0.30659346
, 0.30707107 0.30707107, 0.30659346 0.30751840, 0.30608761 0.30793353, 0.3055557
0 0.30831470, 0.30500000 0.30866025, 0.30442289 0.30896873, 0.30382683 0.3092388
0, 0.30321440 0.30946930, 0.30258819 0.30965926, 0.30195090 0.30980785, 0.301305
26 0.30991445, 0.30065403 0.30997859, 0.30000000 0.31000000, 0.30000000 0.300000
00))
```

SQLite

```
CREATE TABLE population (
 id integer primary key autoincrement not null
);

SELECT AddGeometryColumn (
 NULL,
 'population',
 'shape',
 4326,
 'polygon',
 'xy',
 'null'
);

CREATE TABLE waste_sites (
 id integer primary key autoincrement not null
);

SELECT AddGeometryColumn (
 NULL,
 'waste_sites',
 'site',
 4326,
 'point',
 'xy',
 'null'
);

INSERT INTO population (shape) VALUES (
 st_geometry ('polygon ((.20 .30, .30 .30, .30 .40, .20 .40, .20 .30))', 4326)
);

INSERT INTO population (shape) VALUES (
 st_geometry ('polygon ((.30 .30, .30 .50, .50 .50, .50 .30, .30 .30))', 4326)
);

INSERT INTO population (shape) VALUES (
 st_geometry ('polygon ((.40 .40, .40 .60, .60 .60, .60 .40, .40 .40))', 4326)
);

INSERT INTO waste_sites (site) VALUES (
 st_geometry ('point (.60 .60)', 4326)
);

INSERT INTO waste_sites (site) VALUES (
 st_geometry ('point (.30 .30)', 4326)
);
```

```
--Replace hs.id with ID value of second record in waste_sites table if not 2.
SELECT sa.id, st_astext (st_intersection (st_buffer (hs.site, .01), sa.shape))
 AS "Intersection"
 FROM population sa, waste_sites hs
 WHERE hs.id = 2
 AND st_astext (st_intersection (st_buffer (hs.site, .1), sa.shape))
 NOT LIKE '%EMPTY%';

  id  Intersection

1     POLYGON (( 0.29000000 0.30000000, 0.30000000 0.30000000, 0.30000000
```

```
0.31000000, 0.29934597 0.30997859, 0.29869474 0.30991445, 0.29804910 0.30980785,
 0.29741181 0.30965926, 0.29678561 0.30946930, 0.29617317 0.30923880, 0.29557711
 0.30896873, 0.29500000 0.30866025, 0.29444430 0.30831470, 0.29391239 0.30793353
, 0.29340654 0.30751840, 0.29292893 0.30707107, 0.29248160 0.30659346, 0.2920664
7 0.30608761, 0.29168530 0.30555570, 0.29133975 0.30500000, 0.29103127 0.3044228
9, 0.29076121 0.30382683, 0.29053070 0.30321440, 0.29034074 0.30258819, 0.290192
15 0.30195090, 0.29008555 0.30130526, 0.29002141 0.30065403, 0.29000000 0.300000
00))

2     POLYGON (( 0.30000000 0.30000000, 0.31000000 0.30000000, 0.30997859
0.30065403, 0.30991445 0.30130526, 0.30980785 0.30195090, 0.30965926 0.30258819,
 0.30946930 0.30321440, 0.30923880 0.30382683, 0.30896873 0.30442289, 0.30866025
 0.30500000, 0.30831470 0.30555570, 0.30793353 0.30608761, 0.30751840 0.30659346
, 0.30707107 0.30707107, 0.30659346 0.30751840, 0.30608761 0.30793353, 0.3055557
0 0.30831470, 0.30500000 0.30866025, 0.30442289 0.30896873, 0.30382683 0.3092388
0, 0.30321440 0.30946930, 0.30258819 0.30965926, 0.30195090 0.30980785, 0.301305
26 0.30991445, 0.30065403 0.30997859, 0.30000000 0.31000000, 0.30000000 0.300000
00))
```

ST_Intersects

定義

ST_Intersects は、2 つのジオメトリのインターセクトが空のセットを生成しない場合は 1 (Oracle および SQLite) または t (PostgreSQL)、それ以外の場合は 0 (Oracle および SQLite) または f (PostgreSQL) を返します。

構文

Oracle および PostgreSQL

```
sde.st_intersects (geometry1 sde.st_geometry, geometry2 sde.st_geometry)
```

SQLite

```
st_intersects (geometry1 geometryblob, geometry2 geometryblob)
```

戻り値のタイプ

Boolean

例

消防部長は、有害廃棄物区域の範囲内にある保護区域のリストを必要としています。

保護区域は、sensitive_areas テーブルに格納されます。ポリゴンとして定義された shape 列は、各保護区域の外周を格納します。

有害区域は、hazardous_sites テーブルに格納されます。ポイントとして定義された site 列は、各有害区域の地理的な中心位置を格納します。

SELECT クエリは、各有害区域の周囲にバッファー範囲を作成し、有害区域のバッファーと交差する保護区域のリストを返します。

Oracle

```
--Create and populate tables.
CREATE TABLE sensitive_areas (
 id integer,
 shape sde.st_geometry
);

CREATE TABLE hazardous_sites (
 id integer,
 site sde.st_geometry
);

INSERT INTO sensitive_areas VALUES (
 1,
 sde.st_geometry ('polygon ((20 30, 30 30, 30 40, 20 40, 20 30))', 4326)
);

INSERT INTO sensitive_areas VALUES (
 2,
```

```
 sde.st_geometry ('polygon ((30 30, 30 50, 50 50, 50 30, 30 30))', 4326)
);

INSERT INTO sensitive_areas VALUES (
 3,
 sde.st_geometry ('polygon ((40 40, 40 60, 60 60, 60 40, 40 40))', 4326)
);

INSERT INTO hazardous_sites VALUES (
 4,
 sde.st_geometry ('point (60 60)', 4326)
);

INSERT INTO hazardous_sites VALUES (
 5,
 sde.st_geometry ('point (30 30)', 4326)
);
```

```
--Create a buffer around the hazardous sites, then find the hazardous site buffers that
intersect sensitive areas.
SELECT sa.id SA_ID, hs.id HS_ID
 FROM SENSITIVE_AREAS sa, HAZARDOUS_SITES hs
 WHERE sde.st_intersects (sde.st_buffer (hs.site, .1), sa.shape) = 1
 ORDER BY sa.id;

SA_ID      HS_ID

    1          5
    2          5
    3          4
```

PostgreSQL

```
--Create and populate tables.
CREATE TABLE sensitive_areas (
 id serial,
 shape sde.st_geometry
);

CREATE TABLE hazardous_sites (
 id serial,
 site sde.st_geometry
);

INSERT INTO sensitive_areas (shape) VALUES (
 sde.st_geometry ('polygon ((20 30, 30 30, 30 40, 20 40, 20 30))', 4326)
);

INSERT INTO sensitive_areas (shape) VALUES (
 sde.st_geometry ('polygon ((30 30, 30 50, 50 50, 50 30, 30 30))', 4326)
);

INSERT INTO sensitive_areas (shape) VALUES (
 sde.st_geometry ('polygon ((40 40, 40 60, 60 60, 60 40, 40 40))', 4326)
);

INSERT INTO hazardous_sites (site) VALUES (
 sde.st_geometry ('point (60 60)', 4326)
);
```

```
INSERT INTO hazardous_sites (site) VALUES (
 sde.st_geometry ('point (30 30)', 4326)
);
```

```
--Create a buffer around the hazardous sites, then find the hazardous site buffers that
intersect sensitive areas.
SELECT sa.id AS sid, hs.id AS hid
 FROM sensitive_areas sa, hazardous_sites hs
 WHERE sde.st_intersects (sde.st_buffer (hs.site, .1), sa.shape) = 't'
 ORDER BY sa.id;

sid     hid

  1       2
  2       2
  3       1
```

SQLite

```
--Create and populate tables.
CREATE TABLE sensitive_areas (
 id integer primary key autoincrement not null
);

SELECT AddGeometryColumn (
 NULL,
 'sensitive_areas',
 'shape',
 4326,
 'polygon',
 'xy',
 'null'
);

CREATE TABLE hazardous_sites (
 id integer primary key autoincrement not null
);

SELECT AddGeometryColumn (
 NULL,
 'hazardous_sites',
 'site',
 4326,
 'point',
 'xy',
 'null'
);

INSERT INTO sensitive_areas (shape) VALUES (
 st_geometry ('polygon ((20 30, 30 30, 30 40, 20 40, 20 30))', 4326)
);

INSERT INTO sensitive_areas (shape) VALUES (
 st_geometry ('polygon ((30 30, 30 50, 50 50, 50 30, 30 30))', 4326)
);

INSERT INTO sensitive_areas (shape) VALUES (
 st_geometry ('polygon ((40 40, 40 60, 60 60, 60 40, 40 40))', 4326)
```

```
);

INSERT INTO hazardous_sites (site) VALUES (
 st_geometry ('point (60 60)', 4326)
);

INSERT INTO hazardous_sites (site) VALUES (
 st_geometry ('point (30 30)', 4326)
);
```

```
--Create a buffer around the hazardous sites, then find the hazardous site buffers that
intersect sensitive areas.
SELECT sa.id AS "sid", hs.id AS "hid"
 FROM sensitive_areas sa, hazardous_sites hs
 WHERE st_intersects (st_buffer (hs.site, .1), sa.shape) = 1
 ORDER BY sa.id;

sid     hid

  1       2
  2       2
  3       1
```

ST_Is3d

定義

ST_Is3d は ST_Geometry を入力パラメーターとして、指定されたジオメトリに Z 座標がある場合は 1 (Oracle および SQLite) または t (PostgreSQL)、そうでない場合は 0 (Oracle および SQLite) または f (PostgreSQL) を返します。

構文

Oracle および PostgreSQL

```
sde.st_is3d (geometry1 sde.st_geometry)
```

SQLite

```
st_is3d (geometry1 geometryblob)
```

戻り値のタイプ

Boolean

例

この例では、is3d_test tb を作成し、レコードを入力します。

次に、ST_Is3d を使用して、いずれかのレコードに Z 座標があるかどうかを確認します。

Oracle

```
CREATE TABLE is3d_test (
 id integer,
 geo sde.st_geometry
);

INSERT INTO IS3D_TEST VALUES (
 1902,
 sde.st_geometry ('polygon ((40 120, 90 120, 90 150, 40 150, 40 120))', 4326)
);

INSERT INTO IS3D_TEST VALUES (
 1903,
 sde.st_geometry ('multipoint m((10 10 5), (50 10 6), (10 30 8))' , 4326)
);

INSERT INTO IS3D_TEST VALUES (
 1904,
 sde.st_geometry ('linestring z(10 10 166, 20 10 168)', 4326)
);

INSERT INTO IS3D_TEST VALUES (
 1905,
 sde.st_geometry ('point zm(10 10 16 30)', 4326)
);
```

```
SELECT id, sde.st_is3d (geo) Is_3D
 FROM IS3D_TEST;

        ID      IS_3D

      1902          0
      1903          0
      1904          1
      1905          1
```

PostgreSQL

```
CREATE TABLE is3d_test (
 id integer,
 geo sde.st_geometry
);

INSERT INTO IS3D_TEST VALUES (
 1902,
 sde.st_geometry ('polygon ((40 120, 90 120, 90 150, 40 150, 40 120))', 4326)
);

INSERT INTO IS3D_TEST VALUES (
 1903,
 sde.st_geometry ('multipoint m(10 10 5, 50 10 6, 10 30 8)' , 4326)
);

INSERT INTO IS3D_TEST VALUES (
 1904,
 sde.st_geometry ('linestring z(10 10 166, 20 10 168)', 4326)
);
```

```
INSERT INTO IS3D_TEST VALUES (
 1905,
 sde.st_geometry ('point zm(10 10 16 30)', 4326)
);
```

```
SELECT id, sde.st_is3d (geo)
 AS Is_3D
 FROM is3d_test;

        id      is_3d

      1902          f
      1903          f
      1904          t
      1905          t
```

SQLite

```
CREATE TABLE is3d_test (
 id integer
);

SELECT AddGeometryColumn (
 NULL,
 'is3d_test',
 'geo',
 4326,
 'geometryzm',
 'xyzm',
 'null'
);

INSERT INTO is3d_test VALUES (
 1902,
 st_geometry ('polygon ((40 120, 90 120, 90 150, 40 150, 40 120))', 4326)
);

INSERT INTO is3d_test VALUES (
 1903,
 st_geometry ('multipoint m((10 10 5), (50 10 6), (10 30 8))' , 4326)
);

INSERT INTO is3d_test VALUES (
 1904,
 st_geometry ('linestring z(10 10 166, 20 10 168)', 4326)
);

INSERT INTO is3d_test VALUES (
 1905,
 st_geometry ('point zm(10 10 16 30)', 4326)
);
```

```
SELECT id, st_is3d (geo)
 AS "Is_3D"
 FROM is3d_test;

      id      Is_3D
```

```
1902          0
1903          0
1904          1
1905          1
```

ST_IsClosed

定義

ST_IsClosed は、ST_LineString または ST_MultiLineString を入力として、閉じている場合は 1 (Oracle および SQLite) または t (PostgreSQL)、それ以外の場合は 0 (Oracle および SQLite) または f (PostgreSQL) を返します。

構文

Oracle および PostgreSQL

```
sde.st_isclosed (line1 sde.st_geometry)
sde.st_isclosed (multiline1 sde.st_geometry)
```

SQLite

```
st_isclosed (geometry1 geometryblob)
```

戻り値のタイプ

Boolean

例

ラインストリングのテスト

ラインストリング列を 1 つ持つ closed_linestring テーブルを作成します。

INSERT ステートメントは、closed_linestring テーブルに 2 つのレコードを挿入します。最初のレコードは閉じたラインストリングではなく、2 番目のレコードは閉じたラインストリングです。

クエリは、ST_IsClosed 関数の結果を返します。最初の行はラインストリングが閉じていないため 0 または f を、2 番目の行はラインストリングが閉じているため 1 または t を返します。

Oracle

```
CREATE TABLE closed_linestring (ln1 sde.st_geometry);
```

```
INSERT INTO CLOSED_LINESTRING VALUES (
 sde.st_linefromtext ('linestring (10.02 20.01, 10.32 23.98, 11.92 25.64)', 4326)
);

INSERT INTO CLOSED_LINESTRING VALUES (
 sde.st_linefromtext ('linestring (10.02 20.01, 11.92 35.64, 25.02 34.15, 19.15 33.94,
10.02 20.01)', 4326)
);
```

```
SELECT sde.st_isclosed (ln1) Is_it_closed
 FROM CLOSED_LINESTRING;
```

```
Is_it_closed

0
1
```

PostgreSQL

```
CREATE TABLE closed_linestring (ln1 sde.st_geometry);
```

```
INSERT INTO closed_linestring VALUES (
 sde.st_linestring ('linestring (10.02 20.01, 10.32 23.98, 11.92 25.64)', 4326)
);

INSERT INTO closed_linestring VALUES (
sde.st_linestring ('linestring (10.02 20.01, 11.92 35.64, 25.02 34.15, 19.15 33.94,
10.02 20.01)', 4326)
);
```

```
SELECT sde.st_isclosed (ln1) AS Is_it_closed
FROM closed_linestring;

is_it_closed

f
t
```

SQLite

```
CREATE TABLE closed_linestring (id integer);

SELECT AddGeometryColumn (
 NULL,
 'closed_linestring',
 'ln1',
 4326,
 'linestring',
 'xy',
 'null'
);
```

```
INSERT INTO closed_linestring VALUES (
 1,
 st_linefromtext ('linestring (10.02 20.01, 10.32 23.98, 11.92 25.64)', 4326)
);

INSERT INTO closed_linestring VALUES (
 2,
 st_linefromtext ('linestring (10.02 20.01, 11.92 35.64, 25.02 34.15, 19.15 33.94,
10.02 20.01)', 4326)
);
```

```
SELECT st_isclosed (ln1)
 AS "Is_it_closed"
 FROM closed_linestring;

Is_it_closed

0
1
```

マルチラインストリングのテスト

ST_MultiLineString 列を 1 つ持つ closed_linestring テーブルを作成します。

INSERT ステートメントは、閉じていない ST_MultiLineString レコードと閉じている ST_MultiLineString レコードを挿入します。

クエリは、ST_IsClosed 関数の結果を返します。最初の行は、マルチラインストリングが閉じていないため 0 または f を返します。2 番目の行は、ln1 列に格納されたマルチラインストリングが閉じているため 1 または t を返します。マルチラインストリングは、そのラインストリング エレメントがすべて閉じている場合に閉じます。

Oracle

```
CREATE TABLE closed_mlinestring (mln1 sde.st_geometry);
```

```
INSERT INTO closed_mlinestring VALUES (
 sde.st_mlinefromtext ('multilinestring ((10.02 20.01, 10.32 23.98, 11.92 25.64), (9.55
23.75, 15.36 30.11))', 4326)
);
```

```
INSERT INTO closed_mlinestring VALUES (
 sde.st_mlinefromtext ('multilinestring ((10.02 20.01, 11.92 35.64, 25.02
34.15, 19.15 33.94, 10.02 20.01), (51.71 21.73, 73.36 27.04, 71.52 32.87, 52.43 31.90,
51.71 21.73))', 4326)
);
```

```
SELECT sde.st_isclosed (mln1) Is_it_closed
 FROM CLOSED_MLINESTRING;

Is_it_closed

0
1
```

PostgreSQL

```
CREATE TABLE closed_mlinestring (mln1 sde.st_geometry);
```

```
INSERT INTO closed_mlinestring VALUES (
 sde.st_mlinefromtext ('multilinestring ((10.02 20.01, 10.32 23.98, 11.92 25.64), (9.55
23.75, 15.36 30.11))', 4326)
);

INSERT INTO closed_mlinestring VALUES (
 sde.st_mlinefromtext ('multilinestring ((10.02 20.01, 11.92 35.64, 25.02
34.15, 19.15 33.94, 10.02 20.01), (51.71 21.73, 73.36 27.04, 71.52 32.87, 52.43 31.90,
51.71 21.73))', 4326)
);
```

```
SELECT st_isclosed (mln1)
 AS Is_it_closed
 FROM closed_mlinestring;

is_it_closed

f
t
```

SQLite

```
CREATE TABLE closed_mlinestring (mln1 geometryblob);

SELECT AddGeometryColumn (
 NULL,
 'closed_mlinestring',
 'mln1',
 4326,
 'multilinestring',
 'xy',
 'null'
);
```

```
INSERT INTO closed_mlinestring VALUES (
 st_mlinefromtext ('multilinestring ((10.02 20.01, 10.32 23.98, 11.92 25.64), (9.55
23.75, 15.36 30.11))', 4326)
);

INSERT INTO closed_mlinestring VALUES (
 st_mlinefromtext ('multilinestring ((10.02 20.01, 11.92 35.64, 25.02
34.15, 19.15 33.94, 10.02 20.01), (51.71 21.73, 73.36 27.04, 71.52 32.87, 52.43 31.90,
51.71 21.73))', 4326)
);
```

```
SELECT sde.st_isclosed (mln1)
 AS "Is_it_closed"
 FROM CLOSED_MLINESTRING;

Is_it_closed

0
1
```

ST_IsEmpty

定義

ST_IsEmpty は、ST_Geometry オブジェクトが空の場合は 1 (Oracle および SQLite) または t (PostgreSQL)、それ以外の場合は 0 (Oracle および SQLite) または f (PostgreSQL) を返します。

構文

Oracle および PostgreSQL

```
sde.st_isempty (geometry1 sde.st_geometry)
```

SQLite

```
st_isempty (geometry1 geometryblob)
```

戻り値のタイプ

Boolean

例

次の CREATE TABLE ステートメントは、geotype を持つ empty_test テーブルを作成します。これは、g1 列に格納されているサブクラスのデータ タイプを格納します。

INSERT ステートメントは、ポイント、ラインストリング、ポリゴンのジオメトリ サブクラスに対して、空のレコードと空でないレコードを挿入します。

SELECT クエリは、geotype 列のジオメトリ タイプと、ST_IsEmpty 関数の結果を返します。

Oracle

```
CREATE TABLE empty_test (
 geotype varchar(20),
 g1 sde.st_geometry
);

INSERT INTO EMPTY_TEST VALUES (
 'Point',
 sde.st_pointfromtext ('point (10.02 20.01)', 4326)
);

INSERT INTO EMPTY_TEST VALUES (
 'Point',
 sde.st_pointfromtext ('point empty', 4326)
);

INSERT INTO EMPTY_TEST VALUES (
 'Linestring',
 sde.st_linefromtext ('linestring (10.02 20.01, 10.32 23.98, 11.92 25.64)', 4326)
);

INSERT INTO EMPTY_TEST VALUES (
 'Linestring',
```

```
 sde.st_linefromtext ('linestring empty', 4326)
);

INSERT INTO EMPTY_TEST VALUES (
 'Polygon',
 sde.st_polyfromtext ('polygon ((10.02 20.01, 11.92 35.64, 25.02 34.15,
19.15 33.94, 10.02 20.01))', 4326)
);

INSERT INTO EMPTY_TEST VALUES (
 'Polygon',
 sde.st_polyfromtext('polygon empty', 4326)
);
```

```
SELECT geotype, sde.st_isempty (g1) Is_it_empty
FROM EMPTY_TEST;

GEOTYPE    Is_it_empty

Point         0
Point         1
Linestring    0
Linestring    1
Polygon       0
Polygon       1
```

PostgreSQL

```
CREATE TABLE empty_test (
 geotype varchar(20),
 g1 sde.st_geometry
);

INSERT INTO empty_test VALUES (
 'Point',
 sde.st_point ('point (10.02 20.01)', 4326)
);

INSERT INTO empty_test VALUES (
 'Point',
 sde.st_point ('point empty', 4326)
);

INSERT INTO empty_test VALUES (
 'Linestring',
 sde.st_linestring ('linestring (10.02 20.01, 10.32 23.98, 11.92 25.64)', 4326)
);

INSERT INTO empty_test VALUES (
 'Linestring',
 sde.st_linestring ('linestring empty', 4326)
);

INSERT INTO empty_test VALUES (
 'Polygon',
 sde.st_polygon ('polygon ((10.02 20.01, 11.92 35.64, 25.02 34.15,
19.15 33.94, 10.02 20.01))', 4326)
);
```

```
INSERT INTO empty_test VALUES (
 'Polygon',
 sde.st_polygon ('polygon empty', 4326)
);
```

```
SELECT geotype, sde.st_isempty (g1)
 AS Is_it_empty
 FROM empty_test;

geotype   is_it_empty

Point         f
Point         t
Linestring    f
Linestring    t
Polygon       f
Polygon       f
```

SQLite

```
CREATE TABLE empty_test (
 geotype text(20)
);

SELECT AddGeometryColumn (
 NULL,
 'empty_test',
 'g1',
 4326,
 'geometry',
 'xy',
 'null'
);

INSERT INTO empty_test VALUES (
 'Point',
 st_point ('point (10.02 20.01)', 4326)
);

INSERT INTO empty_test VALUES (
 'Point',
 st_point ('point empty', 4326)
);

INSERT INTO empty_test VALUES (
 'Linestring',
 st_linestring ('linestring (10.02 20.01, 10.32 23.98, 11.92 25.64)', 4326)
);

INSERT INTO empty_test VALUES (
 'Linestring',
 st_linestring ('linestring empty', 4326)
);

INSERT INTO empty_test VALUES (
 'Polygon',
 st_polygon ('polygon ((10.02 20.01, 11.92 35.64, 25.02 34.15,
19.15 33.94, 10.02 20.01))', 4326)
);
```

```
INSERT INTO empty_test VALUES (
 'Polygon',
 st_polygon ('polygon empty', 4326)
);
```

```
SELECT geotype, st_isempty (g1)
 AS "Is_it_empty"
 FROM empty_test;

GEOTYPE    Is_it_empty

Point         0
Point         1
Linestring    0
Linestring    1
Polygon       0
Polygon       1
```

ST_IsMeasured

定義

ST_IsMeasured はジオメトリ オブジェクトを入力パラメーターとして、指定されたジオメトリにメジャーがある場合は 1 (Oracle および SQLite) または t (PostgreSQL) を返します。ない場合は 0 (Oracle および SQLite) または f (PostgreSQL) を返します。

構文

Oracle および PostgreSQL

```
sde.st_ismeasured (geometry1 sde.st_geometry)
```

SQLite

```
st_ismeasured (geometry1 geometryblob)
```

戻り値のタイプ

Boolean

例

ism_test テーブルを作成して値を挿入し、ism_test テーブル内でメジャーを含む行を特定します。

Oracle

```
CREATE TABLE ism_test (
 id integer,
 geom sde.st_geometry
);

INSERT INTO ISM_TEST VALUES (
 19,
 sde.st_geometry ('polygon ((40 120, 90 120, 90 150, 40 150, 40 120))', 4326)
);

INSERT INTO ISM_TEST VALUES (
 20,
 sde.st_geometry ('multipoint m((10 10 5), (50 10 6), (10 30 8))' , 4326)
);

INSERT INTO ISM_TEST VALUES (
 21,
 sde.st_geometry ('linestring z(10 10 166, 20 10 168)', 4326)
);

INSERT INTO ISM_TEST VALUES (
 22,
 sde.st_geometry ('point zm(10 10 16 30)', 4326)
);
```

```
SELECT id, sde.st_ismeasured (geom) M_values
 FROM ISM_TEST;

      ID    M_values

      19          0
      20          1
      21          0
      22          1
```

PostgreSQL

```
CREATE TABLE ism_test (
 id integer,
 geom sde.st_geometry
);

INSERT INTO ISM_TEST VALUES (
 19,
 sde.st_geometry ('polygon ((40 120, 90 120, 90 150, 40 150, 40 120))', 4326)
);

INSERT INTO ISM_TEST VALUES (
 20,
 sde.st_geometry ('multipoint m(10 10 5, 50 10 6, 10 30 8)' , 4326)
);

INSERT INTO ISM_TEST VALUES (
 21,
 sde.st_geometry ('linestring z(10 10 166, 20 10 168)', 4326)
);
```

```
INSERT INTO ISM_TEST VALUES (
 22,
 sde.st_geometry ('point zm(10 10 16 30)', 4326)
);
```

```
SELECT id, sde.st_ismeasured (geom)
  AS has_measures
  FROM ism_test;

      id   has_measures

      19          f
      20          t
      21          f
      22          t
```

SQLite

```
CREATE TABLE ism_test (
 id integer
);

SELECT AddGeometryColumn (
 NULL,
 'ism_test',
 'geom',
 4326,
 'geometryzm',
 'xyzm',
 'null'
);

INSERT INTO ISM_TEST VALUES (
 19,
 st_geometry ('polygon ((40 120, 90 120, 90 150, 40 150, 40 120))', 4326)
);

INSERT INTO ISM_TEST VALUES (
 20,
 st_geometry ('multipoint m((10 10 5), (50 10 6), (10 30 8))' , 4326)
);

INSERT INTO ISM_TEST VALUES (
 21,
 st_geometry ('linestring z(10 10 166, 20 10 168)', 4326)
);

INSERT INTO ISM_TEST VALUES (
 22,
 st_geometry ('point zm(10 10 16 30)', 4326)
);
```

```
SELECT id, st_ismeasured (geom)
 AS "M_values"
 FROM ism_test;

        ID    M_values
```

```
19          0
20          1
21          0
22          1
```

ST_IsRing

定義

ST_IsRing は、ST_LineString を入力として、それがリングの場合 (たとえば、ST_LineString が閉じていてシンプルの場合) は 1 (Oracle および SQLite) または t (PostgreSQL)、それ以外の場合は 0 (Oracle および SQLite) または f (PostgreSQL) を返します。

構文

Oracle および PostgreSQL

```
sde.st_isring (line1 sde.st_geometry)
```

SQLite

```
st_isring (line1 geometryblob)
```

戻り値のタイプ

Boolean

例

ST_LineString 列の ln1 を 1 つ持つ ring_linestring テーブルを作成します。

INSERT ステートメントは、3 つのラインストリングを ln1 列に挿入します。最初の行には、閉じていないためリングではないラインストリングが含まれています。2 番目の行には、リングである、閉じてシンプルなラインストリングが含まれています。3 番目の行には、閉じてはいるが、内部で交わっているためにシンプルでないラインストリングが含まれています。これは、リングではありません。

SELECT クエリは、ST_IsRing 関数の結果を返します。最初の行はラインストリングがリングでないため 0 または f を、2 番目と 3 番目の行はラインストリングがリングであるため 1 または t を返します。

Oracle

```
CREATE TABLE ring_linestring (ln1 sde.st_geometry);
```

```
INSERT INTO RING_LINESTRING VALUES (
 sde.st_linefromtext ('linestring (10.02 20.01, 10.32 23.98, 11.92 25.64)', 4326)
);

INSERT INTO RING_LINESTRING VALUES (
 sde.st_linefromtext ('linestring (10.02 20.01, 11.92 35.64, 25.02 34.15, 19.15 33.94,
10.02 20.01)', 4326)
);

INSERT INTO ring_linestring (ln1) VALUES (
 sde.st_linestring ('linestring (11 31, 11.25 31.12, 21.83 44.13, 16.45 44.24, 11 31)',
4326)
);
```

```
SELECT sde.st_isring (ln1) Is_it_a_ring
 FROM RING_LINESTRING;

Is_it_a_ring

0
1
1
```

PostgreSQL

```
CREATE TABLE ring_linestring (ln1 sde.st_geometry);
```

```
INSERT INTO ring_linestring VALUES (
 sde.st_linestring ('linestring (10.02 20.01, 10.32 23.98, 11.92 25.64)', 4326)
);

INSERT INTO ring_linestring VALUES (
 sde.st_linestring ('linestring (10.02 20.01, 11.92 35.64, 25.02 34.15, 19.15 33.94,
10.02 20.01)', 4326)
);

INSERT INTO ring_linestring (ln1) VALUES (
 sde.st_linestring ('linestring (11 31, 11.25 31.12, 21.83 44.13, 16.45 44.24, 11 31)',
4326)
);
```

```
SELECT sde.st_isring (ln1)
 AS Is_it_a_ring
 FROM ring_linestring;

Is_it_a_ring

f
```

```
t
t
```

SQLite

```
CREATE TABLE ring_linestring (id integer primary key autoincrement not null);

SELECT AddGeometryColumn (
 NULL,
 'ring_linestring',
 'ln1',
 4326,
 'linestring',
 'xy',
 'null'
);
```

```
INSERT INTO ring_linestring (ln1) VALUES (
 st_linestring ('linestring (10.02 20.01, 10.32 23.98, 11.92 25.64)', 4326)
);

INSERT INTO ring_linestring (ln1) VALUES (
 st_linestring ('linestring (10.02 20.01, 11.92 35.64, 25.02 34.15, 19.15 33.94, 10.02
20.01)', 4326)
);

INSERT INTO ring_linestring (ln1) VALUES (
 st_linestring ('linestring (11 31, 11.25 31.12, 21.83 44.13, 16.45 44.24, 11 31)',
4326)
);
```

```
SELECT st_isring (ln1)
 AS "Is it a ring?"
 FROM ring_linestring;

Is it a ring?

0
1
1
```

ST_IsSimple

定義

ST_IsSimple は、OGC (Open Geospatial Consortium) に定義されているように ジオメトリ オブジェクトがシンプルの場合は 1 (Oracle および SQLite) または t (PostgreSQL)、それ以外の場合は 0 (Oracle および SQLite) または f (PostgreSQL) を返します。

構文

Oracle および PostgreSQL

```
sde.st_issimple (geometry1 sde.st_geometry)
```

SQLite

```
st_issimple (geometry1 geometryblob)
```

戻り値のタイプ

Boolean

例

issimple_test テーブルは、2 つの列で作成されます。pid は smallint データ タイプで、各行の一意の識別子が含まれています。g1 列は、シンプルまたは非シンプルのジオメトリ サンプルを格納します。

INSERT ステートメントは、2 つのレコードを issimple_test テーブルに挿入します。最初のレコードは、内部で交わらないシンプルなラインストリングです。2 番目のレコードは、内部で交わる OGC による定義では非シンプルになります。

クエリは、ST_IsSimple 関数の結果を返します。最初のレコードは、ラインストリングがシンプルであるため 1 または t を返します。2 番目のレコードは、ラインストリングがシンプルでないため 0 または f を返します。

Oracle

```
CREATE TABLE issimple_test (
 pid smallint,
 g1 sde.st_geometry
);
```

```
INSERT INTO ISSIMPLE_TEST VALUES (
 1,
 sde.st_linefromtext ('linestring (10 10, 20 20, 30 30)', 4326)
);

INSERT INTO ISSIMPLE_TEST VALUES (
 2,
 sde.st_linefromtext ('linestring (10 10, 20 20, 20 30, 10 30, 10 20,
20 10)', 4326)
);
```

```
SELECT pid, sde.st_issimple (g1) Is_it_simple
 FROM ISSIMPLE_TEST;

PID                 Is_it_simple

1                    1
2                    0
```

PostgreSQL

```
CREATE TABLE issimple_test (
 pid smallint,
 g1 sde.st_geometry
);
```

```
INSERT INTO issimple_test VALUES (
 1,
 sde.st_linestring ('linestring (10 10, 20 20, 30 30)', 4326)
);

INSERT INTO issimple_test VALUES (
 2,
 sde.st_linestring  ('linestring (10 10, 20 20, 20 30, 10 30, 10 20, 20 10)', 4326)
);
```

```
SELECT pid, sde.st_issimple (g1)
 AS Is_it_simple
 FROM issimple_test;

pid                is_it_simple

1                    t
2                    f
```

SQLite

```
CREATE TABLE issimple_test (
 pid integer
);

SELECT AddGeometryColumn (
 NULL,
 'issimple_test',
 'g1',
 4326,
 'linestring',
 'xy',
 'null'
);
```

```
INSERT INTO issimple_test VALUES (
 1,
 st_linestring ('linestring (10 10, 20 20, 30 30)', 4326)
);

INSERT INTO issimple_test VALUES (
 2,
 st_linestring  ('linestring (10 10, 20 20, 20 30, 10 30, 10 20, 20 10)', 4326)
);
```

```
SELECT pid, st_issimple (g1)
 AS Is_it_simple
 FROM issimple_test;

PID                 Is_it_simple

1                    1
2                    0
```

ST_Length

定義

ST_Length は、ライン ストリングまたはマルチライン ストリングの長さを返します。

構文

Oracle および PostgreSQL

```
sde.st_length (line1 sde.st_geometry)
sde.st_length (multiline1 sde.st_geometry)
```

SQLite

```
st_length (line1 geometryblob)
st_length (multiline1 geometryblob)
st_length (line1  geometryblob, unit_name text)
st_length (multiline1  geometryblob, unit_name text)
```

サポートされる単位名の一覧については、「ST_Distance」をご参照ください。

戻り値のタイプ

Double precision

例

国内の水路に生息する鮭の移動パターンを調査している生態学者は、国内のすべての水路と河川の長さを知りたいと考えています。

テーブルに格納された各水路と河川を識別する ID と name 列を持つ waterways テーブルを作成します。水路と河川は複数のラインストリングの集合であることが多いため、water 列はマルチラインストリングです。

SELECT クエリは、水路の名前と、ST_Length 関数によって出力されたその水路の長さを返します。Oracle および PostgreSQL の場合、単位は座標系で使用されているものになります。SQLite の場合、単位としてキロメートルが指定されます。

Oracle

```
CREATE TABLE waterways (
 oid integer,
 name varchar(128),
 water sde.st_geometry
);
```

```
INSERT INTO waterways (oid, name, water) VALUES (
  1111,
  'Genesee',
  sde.st_multilinestring ('multilinestring ((33 2, 34 3, 35 6),
(28 4, 29 5, 31 8, 43 12), (39 3, 37 4, 36 7))', 4326)
);
```

```
SELECT name, sde.st_length (water) "Length"
 FROM WATERWAYS;

NAME        Length

Genesee        27.6437123
```

PostgreSQL

```
CREATE TABLE waterways (
 oid serial,
 name varchar(128),
 water sde.st_geometry
);
```

```
INSERT INTO waterways (name, water) VALUES (
  'Genesee',
  sde.st_multilinestring ('multilinestring ((33 2, 34 3, 35 6),
(28 4, 29 5, 31 8, 43 12), (39 3, 37 4, 36 7))', 4326)
);
```

```
SELECT name AS "Watershed Name",
 sde.st_length (water) AS "Length"
 FROM waterways;

Watershed Name         |  Length

Genesee               | 27.6437123387202
```

SQLite

```
CREATE TABLE waterways (
 oid integer primary key autoincrement not null,
 name text(128)
);
```

```
SELECT AddGeometryColumn (
 NULL,
 'waterways',
 'water',
 4326,
 'multilinestring',
 'xy',
 'null'
);
```

```
INSERT INTO waterways (name, water) VALUES (
  'Genesee',
  st_multilinestring ('multilinestring ((33 2, 34 3, 35 6),
(28 4, 29 5, 31 8, 43 12), (39 3, 37 4, 36 7))', 4326)
);
```

```
SELECT name AS "Watershed Name",
 st_length (water, 'kilometer') AS "Length"
 FROM waterways1;

Watershed Name            Length

Genesee                   3047.75515002795
```

ST_LineFromText

注意:
Oracle および SQLite でのみサポートされます。PostgreSQL では、ST_LineString を使用してください。

定義

ST_LineFromText は、ST_LineString タイプの WKT 表現と空間参照 ID を受け取って、ST_LineString を返します。

構文

Oracle

```
sde.st_linefromtext (wkt clob, srid integer)
```

```
sde.st_linefromtext (wkt clob)
```

SRID を指定しない場合、空間参照がデフォルトで 4326 に設定されます。

SQLite

```
st_linefromtext (wkt text, srid int32)
```

```
st_linefromtext (wkt text)
```

SRID を指定しない場合、空間参照がデフォルトで 4326 に設定されます。

戻り値のタイプ

ST_LineString

例

ln1 ST_LineString 列を 1 つ持つ linestring_test テーブルを作成します。

INSERT ステートメントは、ST_LineFromText 関数を使用して、ST_LineString を ln1 列に挿入します。

Oracle

```
CREATE TABLE linestring_test (ln1 sde.st_geometry);
```

```
INSERT INTO LINESTRING_TEST VALUES (
 sde.st_linefromtext ('linestring (10.01 20.03, 35.93 19.04)', 4326)
);
```

SQLite

```
CREATE TABLE linestring_test (id integer);
SELECT AddGeometryColumn (
 NULL,
 'linestring_test',
 'ln1',
 4326,
 'linestring',
 'xy',
 'null'
);
```

```
INSERT INTO LINESTRING_TEST (id, ln1) VALUES (
 1,
 st_linefromtext ('linestring (10.01 20.03, 35.93 19.04)', 4326)
);
```

ST_LineFromWKB

定義

ST_LineFromWKB は、ST_LineString タイプの WKB 表現と空間参照 ID を受け取って、ST_LineString を返します。

構文

Oracle

```
sde.st_linefromwkb (wkb blob, srid integer)
```

```
sde.st_linefromwkb (wkb blob)
```

SRID を指定しない場合、空間参照がデフォルトで 4326 に設定されます。

PostgreSQL

```
sde.st_linefromwkb (wkb bytea, srid integer)
```

SQLite

```
st_linefromwkb (wkb blob, srid int32)
```

```
st_linefromwkb (wkb blob)
```

SRID を指定しない場合、空間参照がデフォルトで 4326 に設定されます。

戻り値のタイプ

ST_LineString

例

次のコマンドはテーブル (sample_lines) を作成し、ST_LineFromWKB 関数を使用して、WKB 表現からラインを挿入します。ID と WKB 表現の空間参照系 4326 のラインを含む sample_lines テーブルに行を挿入します。

Oracle

```
CREATE TABLE sample_lines (
 id smallint,
 geometry sde.st_linestring,
 wkb blob
);
INSERT INTO SAMPLE_LINES (id, geometry) VALUES (
 1901,
 sde.st_linestring ('linestring (850 250, 850 850)', 4326)
);
```

```
INSERT INTO SAMPLE_LINES (id, geometry) VALUES (
 1902,
 sde.st_linestring ('linestring (33 2, 34 3, 35 6)', 4326)
);
UPDATE SAMPLE_LINES
 SET wkb = sde.st_asbinary (geometry)
 WHERE id = 1901;
UPDATE SAMPLE_LINES
 SET wkb = sde.st_asbinary (geometry)
 WHERE id = 1902;
SELECT id, sde.st_astext (sde.st_linefromwkb (wkb,4326)) LINE
 FROM SAMPLE_LINES;
ID   LINE
1901 LINESTRING (850.00000000 250.00000000, 850.00000000 850.00000000)
1902 LINESTRING (33.00000000 2.00000000, 34.00000000 3.00000000, 35.00000000 6.00000000)
```

PostgreSQL

```
CREATE TABLE sample_lines (
 id serial,
 geometry sde.st_linestring,
 wkb bytea
);
INSERT INTO sample_lines (geometry) VALUES (
 sde.st_linestring ('linestring (850 250, 850 850)', 4326)
);
INSERT INTO sample_lines (geometry) VALUES (
 sde.st_linestring ('linestring (33 2, 34 3, 35 6)', 4326)
);
--Replace ID values if necessary.
UPDATE sample_lines
 SET wkb = sde.st_asbinary (geometry)
 WHERE id = 1;
UPDATE sample_lines
 SET wkb = sde.st_asbinary (geometry)
 WHERE id = 2;
SELECT id, sde.st_astext (st_linefromwkb (wkb,4326))
 AS LINE
 FROM sample_lines;
id   line
1    LINESTRING (850 250, 850 850)
2    LINESTRING (33 2, 34 3, 35 6)
```

SQLite

```
CREATE TABLE sample_lines (
 id integer primary key autoincrement not null,
 wkb blob
);
SELECT AddGeometryColumn (
 NULL,
 'sample_lines',
 'geometry',
 4326,
 'linestring',
 'xy',
 'null'
);
INSERT INTO sample_lines (geometry) VALUES (
```

```
 st_linestring ('linestring (850 250, 850 850)', 4326)
);
INSERT INTO sample_lines (geometry) VALUES (
 st_linestring ('linestring (33 2, 34 3, 35 6)', 4326)
);
--Replace ID values if necessary.
UPDATE sample_lines
 SET wkb = st_asbinary (geometry)
 WHERE id = 1;
UPDATE sample_lines
 SET wkb = st_asbinary (geometry)
 WHERE id = 2;
SELECT id, st_astext (st_linefromwkb (wkb,4326))
 AS LINE
 FROM sample_lines;
id   LINE
1    LINESTRING (850.00000000 250.00000000, 850.00000000 850.00000000)
2    LINESTRING (33.00000000 2.00000000, 34.00000000 3.00000000, 35.00000000 6.00000000)
```

ST_LineString

定義

ST_LineString は、WKT 表現からラインストリングを構築するアクセサ関数です。

注意:
ArcGIS で使用する空間テーブルを作成する場合は、ST_Geometry サブタイプを指定するよりも、ジオメトリ スーパータイプ (たとえば、ST_Geometry) として列を作成することをお勧めします。

構文

Oracle

```
sde.st_linestring (wkt clob, srid integer)
```

PostgreSQL

```
sde.st_linestring (wkt text, srid integer)
sde.st_linestring (esri_shape bytea, srid integer)
```

SQLite

```
st_linestring (wkt text, srid int32)
```

戻り値のタイプ

ST_LineString

例

Oracle

```
CREATE TABLE lines_test (
 id smallint,
 geometry sde.st_geometry
);

INSERT INTO LINES_TEST (id, geometry) VALUES (
 1901,
 sde.st_linestring ('linestring (750 150, 750 750)', 4326)
);

SELECT id, sde.st_astext (geometry) Linestring
 FROM   LINES_TEST;

  ID  LINESTRING

1901  LINESTRING  (750.00000000 150.00000000,
750.00000000 750.00000000)
```

PostgreSQL

```
CREATE TABLE lines_test (
 id serial,
 geometry sde.st_geometry
);

INSERT INTO lines_test (geometry) VALUES (
 sde.st_linestring ('linestring (750 150, 750 750)', 4326)
);

SELECT id, sde.st_astext (geometry)
 AS Linestring
 FROM lines_test;

  id  linestring

  1  LINESTRING  (750 150, 750 750)
```

SQLite

```
CREATE TABLE lines_test (
 id integer primary key autoincrement not null
);

SELECT AddGeometryColumn (
 NULL,
 'lines_test',
 'geometry',
 4326,
 'linestring',
 'xy',
 'null'
);

INSERT INTO lines_test (geometry) VALUES (
 st_linestring ('linestring (750 150, 750 750)', 4326)
);

SELECT id, st_astext (geometry)
 AS "Linestring"
 FROM lines_test;

  id  linestring

  1  LINESTRING  (750.00000000 150.00000000, 750.00000000 750.00000000)
```

ST_M

定義

ST_M は、ST_Point を入力パラメーターとして、そのメジャー (M) 座標を返します。

SQLite では、ST_M はメジャー値の更新にも使用できます。

構文

Oracle および PostgreSQL

```
sde.st_m (point1 sde.st_point)
```

SQLite

```
st_m (point1 geometryblob)
st_m (point1 geometryblob, new_Mvalue double)
```

戻り値のタイプ

Oracle および PostgreSQL

Number

SQLite

メジャー値を検索している場合は Double precision。メジャー値を更新している場合は geometryblob。

例

Oracle

m_test テーブルを作成し、3 つのポイントを挿入します。3 つのポイントすべてに、メジャー値が含まれています。ST_M 関数を使用して SELECT ステートメントを実行すると、各ポイントのメジャー値が返されます。

```
CREATE TABLE m_test (
 id integer,
 geometry sde.st_point);
INSERT INTO M_TEST VALUES (
 1,
 sde.st_point (2, 3, 32, 5, 4322)
);
INSERT INTO M_TEST VALUES (
 2,
 sde.st_point (4, 5, 20, 4, 4326)
);
INSERT INTO M_TEST VALUES (
 3,
 sde.st_point (3, 8, 23, 7, 4326)
);
SELECT id, sde.st_m (geometry) M_COORD
 FROM M_TEST;
        ID    M_COORD
```

```
         1          5
         2          4
         3          7
```

PostgreSQL

m_test テーブルを作成し、3 つのポイントを挿入します。3 つのポイントすべてに、メジャー値が含まれています。ST_M 関数を使用して SELECT ステートメントを実行すると、各ポイントのメジャー値が返されます。

```
CREATE TABLE m_test (
 id serial,
 geometry sde.st_point
);
INSERT INTO m_test (geometry) VALUES (
 sde.st_point (2, 3, 32, 5, 4326)
);
INSERT INTO m_test (geometry) VALUES (
 sde.st_point (4, 5, 20, 4, 4326)
);
INSERT INTO m_test (geometry) VALUES (
 sde.st_point (3, 8, 23, 7, 4326)
);
SELECT id, sde.st_m (geometry)
 AS M_COORD
 FROM m_test;
        id    m_coord
         1          5
         2          4
         3          7
```

SQLite

最初の例では、m_test テーブルを作成し、3 つのポイントを挿入します。3 つのポイントすべてに、メジャー値が含まれています。ST_M 関数を使用して SELECT ステートメントを実行すると、各ポイントのメジャー値が返されます。

```
CREATE TABLE m_test (
 id integer primary key autoincrement not null
);
SELECT AddGeometryColumn (
 NULL,
 'm_test',
 'geometry',
 4326,
 'pointzm',
 'xyzm',
 'null'
);
INSERT INTO m_test (geometry) VALUES (
 st_point (2, 3, 32, 5, 4326)
);
INSERT INTO m_test (geometry) VALUES (
 st_point (4, 5, 20, 4, 4326)
);
INSERT INTO m_test (geometry) VALUES (
 st_point (3, 8, 23, 7, 4326)
);
SELECT id, st_m (geometry)
```

```
 AS M_COORD
 FROM m_test;
id    m_coord
1     5.0
2     4.0
3     7.0
```

2 つ目の例では、m_test テーブル内にあるレコード 3 のメジャー値を更新します。

```
SELECT st_m (geometry, 7.5)
 FROM m_test
 WHERE id = 3;
```

ST_MaxM

定義

ST_MaxM は、ジオメトリを入力パラメーターとして、その最大 M 座標を返します。

構文

Oracle および PostgreSQL

```
sde.st_maxm (geometry1 sde.st_geometry)
```

SQLite

```
st_maxm (geometry1 geometryblob)
```

戻り値のタイプ

Oracle および PostgreSQL

Number

M 値が存在しない場合、NULL が返されます。

SQLite

Double precision

M 値が存在しない場合、NULL が返されます。

例

maxm_test テーブルを作成し、2 つのポリゴンを挿入します。次に、ST_MaxM を実行して、各ポリゴンの最大 M 値を判定します。

Oracle

```
CREATE TABLE maxm_test (
 id integer,
 geometry sde.st_geometry
);
INSERT INTO MAXM_TEST VALUES (
 1901,
 sde.st_polygon ('polygon zm((110 120 20 3, 110 140 22 3, 120 130 26 4, 110 120 20
3))', 4326)
);
INSERT INTO MAXM_TEST VALUES (
 1902,
 sde.st_polygon ('polygon zm((0 0 40 7, 0 4 35 9, 5 4 32 12, 5 0 31 5, 0 0 40 7))',
4326)
);
SELECT id, sde.st_maxm (geometry) Max_M
 FROM MAXM_TEST;
        ID             MAX_M
```

```
      1901                4
      1902               12
```

PostgreSQL

```
CREATE TABLE maxm_test (
 id integer,
 geometry sde.st_geometry
);
INSERT INTO maxm_test VALUES (
 1901,
 sde.st_polygon ('polygon zm((110 120 20 3, 110 140 22 3, 120 130 26 4, 110 120 20
3))', 4326)
);
INSERT INTO maxm_test VALUES (
 1902,
 sde.st_polygon ('polygon zm((0 0 40 7, 0 4 35 9, 5 4 32 12, 5 0 31 5, 0 0 40 7))',
4326)
);
SELECT id, sde.st_maxm (geometry)
 AS Max_M
 FROM maxm_test;
        id             max_m
      1901                4
      1902               12
```

SQLite

```
CREATE TABLE maxm_test (
 id integer
);
SELECT AddGeometryColumn (
 NULL,
 'maxm_test',
 'geometry',
 4326,
 'polygonzm',
 'xyzm',
 'null'
);
INSERT INTO maxm_test VALUES (
 1901,
 st_polygon ('polygon zm((110 120 20 3, 110 140 22 3, 120 130 26 4, 110 120 20 3))',
4326)
);
INSERT INTO maxm_test VALUES (
 1902,
 st_polygon ('polygon zm((0 0 40 7, 0 4 35 9, 5 4 32 12, 5 0 31 5, 0 0 40 7))', 4326)
);
SELECT id, st_maxm (geometry)
 AS "Max M"
 FROM maxm_test;
id             Max M
1901           4.0
1902           12.0
```

ST_MaxX

定義

ST_MaxX は、ジオメトリを入力パラメーターとして、その最大 X 座標を返します。

構文

Oracle および PostgreSQL

```
sde.st_maxx (geometry1 sde.st_geometry)
```

SQLite

```
st_maxx (geometry1 geometryblob)
```

戻り値のタイプ

Oracle および PostgreSQL

Number

SQLite

Double precision

例

maxx_test テーブルを作成し、2 つのポリゴンを挿入します。次に、ST_MaxX 関数を使用して、各ポリゴンの最大 X 値を判定します。

Oracle

```
CREATE TABLE maxx_test (
 id integer,
 geometry sde.st_geometry
);
INSERT INTO MAXX_TEST VALUES (
 1901,
 sde.st_polygon ('polygon zm((110 120 20 3, 110 140 22 3, 120 130 26 4, 110 120 20
3))', 4326)
);
INSERT INTO MAXX_TEST VALUES (
 1902,
 sde.st_polygon ('polygon zm((0 0 40 7, 0 4 35 9, 5 4 32 12, 5 0 31 5, 0 0 40 7))',
4326)
);
SELECT id, sde.st_maxx (geometry) Max_X
 FROM MAXX_TEST;
        ID      MAX_X
      1901        120
      1902          5
```

PostgreSQL

```
CREATE TABLE maxx_test (
 id integer,
 geometry sde.st_geometry
);
INSERT INTO maxx_test VALUES (
 1901,
 sde.st_polygon ('polygon zm((110 120 20 3, 110 140 22 3, 120 130 26 4, 110 120 20
3))', 4326)
);
INSERT INTO maxx_test VALUES (
 1902,
 sde.st_polygon ('polygon zm((0 0 40 7, 0 4 35 9, 5 4 32 12, 5 0 31 5, 0 0 40 7))',
4326)
);
SELECT id, sde.st_maxx (geometry)
 AS Max_X
 FROM maxx_test;
        id      max_x
      1901        120
      1902          5
```

SQLite

```
CREATE TABLE maxx_test (
 id integer
);
SELECT AddGeometryColumn (
 NULL,
 'maxx_test',
 'geometry',
 4326,
 'polygonzm',
 'xyzm',
 'null'
);
INSERT INTO maxx_test VALUES (
 1901,
 st_polygon ('polygon zm((110 120 20 3, 110 140 22 3, 120 130 26 4, 110 120 20 3))',
4326)
);
INSERT INTO maxx_test VALUES (
 1902,
 st_polygon ('polygon zm((0 0 40 7, 0 4 35 9, 5 4 32 12, 5 0 31 5, 0 0 40 7))', 4326)
);
SELECT id, st_maxx (geometry)
 AS "max_x"
 FROM maxx_test;
id      max_x
1901    120.0
1902    5.00000000
```

ST_MaxY

定義

ST_MaxY は、ジオメトリを入力パラメーターとして、その最大 Y 座標を返します。

構文

Oracle および PostgreSQL

```
sde.st_maxy (geometry1 sde.st_geometry)
```

SQLite

```
st_maxy (geometry1 geometryblob)
```

戻り値のタイプ

Oracle および PostgreSQL

Number

SQLite

Double precision

例

maxy_test テーブルを作成し、2 つのポリゴンを挿入します。次に、ST_MaxY 関数を使用して、各ポリゴンの最大 Y 値を判定します。

Oracle

```
CREATE TABLE maxy_test (
 id integer,
 geometry sde.st_geometry
);
INSERT INTO MAXY_TEST VALUES (
 1901,
 sde.st_polygon ('polygon zm((110 120 20 3, 110 140 22 3, 120 130 26 4, 110 120 20
3))', 4326)
);
INSERT INTO MAXY_TEST VALUES (
 1902,
 sde.st_polygon ('polygon zm((0 0 40 7, 0 4 35 9, 5 4 32 12, 5 0 31 5, 0 0 40 7))',
4326)
);
SELECT id, sde.st_maxy (geometry) Max_Y
 FROM MAXY_TEST;
        ID      MAX_Y
      1901        140
      1902          4
```

PostgreSQL

```
CREATE TABLE maxy_test (
 id integer,
 geometry sde.st_geometry
);
INSERT INTO maxy_test VALUES (
 1901,
 sde.st_polygon ('polygon zm((110 120 20 3, 110 140 22 3, 120 130 26 4, 110 120 20
3))', 4326)
);
INSERT INTO maxy_test VALUES (
 1902,
 sde.st_polygon ('polygon zm((0 0 40 7, 0 4 35 9, 5 4 32 12, 5 0 31 5, 0 0 40 7))',
4326)
);
SELECT id, sde.st_maxy (geometry)
 AS Max_Y
 FROM maxy_test;
        id      max_y
      1901        140
      1902          4
```

SQLite

```
CREATE TABLE maxy_test (
 id integer
);
SELECT AddGeometryColumn (
 NULL,
 'maxy_test',
 'geometry',
 4326,
 'polygonzm',
 'xyzm',
 'null'
);
INSERT INTO maxy_test VALUES (
 1901,
 st_polygon ('polygon zm((110 120 20 3, 110 140 22 3, 120 130 26 4, 110 120 20 3))',
4326)
);
INSERT INTO maxy_test VALUES (
 1902,
 st_polygon ('polygon zm((0 0 40 7, 0 4 35 9, 5 4 32 12, 5 0 31 5, 0 0 40 7))', 4326)
);
SELECT id, st_maxy (geometry)
 AS "max_y"
 FROM maxy_test;
id      max_y
1901    140.0
1902    4.00000000
```

ST_MaxZ

定義

ST_MaxZ は、ジオメトリを入力パラメーターとして、その最大 Z 座標を返します。

構文

Oracle および PostgreSQL

```
sde.st_maxz (geometry1 sde.st_geometry)
```

SQLite

```
st_maxz (geometry1 geometryblob)
```

戻り値のタイプ

Oracle および PostgreSQL

Number

Z 値が存在しない場合、NULL が返されます。

SQLite

Double precision

Z 値が存在しない場合、NULL が返されます。

例

次の例では、maxz_test テーブルを作成し、2 つのポリゴンを挿入します。次に、ST_MaxZ を実行して、各ポリゴンの最大 Z 値が返されます。

Oracle

```
CREATE TABLE maxz_test (
 id integer,
 geometry sde.st_geometry
);
INSERT INTO MAXZ_TEST VALUES (
 1901,
 sde.st_polygon ('polygon zm((110 120 20 3, 110 140 22 3, 120 130 26 4, 110 120 20
3))', 4326)
);
INSERT INTO MAXZ_TEST VALUES (
 1902,
 sde.st_polygon ('polygon zm((0 0 40 7, 0 4 35 9, 5 4 32 12, 5 0 31 5, 0 0 40 7))',
4326)
);
SELECT id, sde.st_maxz (geometry) Max_Z
 FROM MAXZ_TEST;
        ID      MAX_Z
```

```
      1901         26
      1902         40
```

PostgreSQL

```
CREATE TABLE maxz_test (
 id integer,
 geometry sde.st_geometry
);
INSERT INTO maxz_test VALUES (
 1901,
 sde.st_polygon ('polygon zm((110 120 20 3, 110 140 22 3, 120 130 26 4, 110 120 20
3))', 4326)
);
INSERT INTO maxz_test VALUES (
 1902,
 sde.st_polygon ('polygon zm((0 0 40 7, 0 4 35 9, 5 4 32 12, 5 0 31 5, 0 0 40 7))',
4326)
);
SELECT id, sde.st_maxz (geometry)
 AS Max_Z
 FROM maxz_test;
        id      max_z
      1901         26
      1902         40
```

SQLite

```
CREATE TABLE maxz_test (
 id integer
);
SELECT AddGeometryColumn (
 NULL,
 'maxz_test',
 'geometry',
 4326,
 'polygonzm',
 'xyzm',
 'null'
);
INSERT INTO maxz_test VALUES (
 1901,
 st_polygon ('polygon zm((110 120 20 3, 110 140 22 3, 120 130 26 4, 110 120 20 3))',
4326)
);
INSERT INTO maxz_test VALUES (
 1902,
 st_polygon ('polygon zm((0 0 40 7, 0 4 35 9, 5 4 32 12, 5 0 31 5, 0 0 40 7))', 4326)
);
SELECT id AS "ID", st_maxz (geometry) AS "Max Z"
 FROM maxz_test;
ID      Max Z
1901    26.0
1902    40.0
```

ST_MinM

定義

ST_MinM は、ジオメトリを入力パラメーターとして、その最小 M 座標を返します。

構文

Oracle および PostgreSQL

```
sde.st_minm (geometry1 sde.st_geometry)
```

SQLite

```
st_minm (geometry1 geometryblob)
```

戻り値のタイプ

Oracle および PostgreSQL

Number

M 値が存在しない場合、NULL が返されます。

SQLite

Double precision

M 値が存在しない場合、NULL が返されます。

例

minm_test テーブルを作成し、2 つのポリゴンを挿入します。次に、ST_MinM を実行して、各ポリゴンの最小 M 値を判定します。

PostgreSQL

Oracle

```
CREATE TABLE minm_test (
 id integer,
 geometry sde.st_geometry
);
INSERT INTO MINM_TEST VALUES (
 1901,
 sde.st_polygon ('polygon zm((110 120 20 3, 110 140 22 3, 120 130 26 4, 110 120 20
3))', 4326)
);
INSERT INTO MINM_TEST VALUES (
 1902,
 sde.st_polygon ('polygon zm((0 0 40 7, 0 4 35 9, 5 4 32 12, 5 0 31 5, 0 0 40 7))',
4326)
);
SELECT id, sde.st_minm (geometry) MinM
```

```
 FROM MINM_TEST;
        ID       MINM
      1901          3
      1902          5
```

PostgreSQL

```
CREATE TABLE minm_test (
 id integer,
 geometry sde.st_geometry
);
INSERT INTO minm_test VALUES (
 1901,
 sde.st_polygon ('polygon zm((110 120 20 3, 110 140 22 3, 120 130 26 4, 110 120 20
3))', 4326)
);
INSERT INTO minm_test VALUES (
 1902,
 sde.st_polygon ('polygon zm((0 0 40 7, 0 4 35 9, 5 4 32 12, 5 0 31 5, 0 0 40 7))',
4326)
);
SELECT id, sde.st_minm (geometry)
 AS MinM
 FROM minm_test;
        id       minm
      1901          3
      1902          5
```

SQLite

```
CREATE TABLE minm_test (
 id integer
);
SELECT AddGeometryColumn (
 NULL,
 'minm_test',
 'geometry',
 4326,
 'polygonzm',
 'xyzm',
 'null'
);
INSERT INTO minm_test VALUES (
 1901,
 st_polygon ('polygon zm((110 120 20 3, 110 140 22 3, 120 130 26 4, 110 120 20 3))',
4326)
);
INSERT INTO minm_test VALUES (
 1902,
 st_polygon ('polygon zm((0 0 40 7, 0 4 35 9, 5 4 32 12, 5 0 31 5, 0 0 40 7))', 4326)
);
SELECT id, st_minm (geometry)
 AS "MinM"
 FROM minm_test;
id       MinM
1901     3.0
1902     5.0
```

ST_MinX

定義

ST_MinX は、ジオメトリを入力パラメーターとして、その最小 X 座標を返します。

構文

Oracle および PostgreSQL

```
sde.st_minx (geometry1 sde.st_geometry)
```

SQLite

```
st_minx (geometry1 geometryblob)
```

戻り値のタイプ

Oracle および PostgreSQL

Number

SQLite

Double precision

例

minx_test テーブルを作成し、2 つのポリゴンを挿入します。次に、ST_MinX が実行して、各ポリゴンの最小 X 座標値を判定します。

Oracle

```
CREATE TABLE minx_test (
 id integer,
 geometry sde.st_geometry
);
INSERT INTO MINX_TEST VALUES (
 1901,
 sde.st_polygon ('polygon zm((110 120 20 3, 110 140 22 3, 120 130 26 4, 110 120 20
3))', 4326)
);
INSERT INTO MINX_TEST VALUES (
 1902,
 sde.st_polygon ('polygon zm((0 0 40 7, 0 4 35 9, 5 4 32 12, 5 0 31 5, 0 0 40 7))',
4326)
);
SELECT id, sde.st_minx (geometry) MinX
 FROM MINX_TEST;
        ID       MINX
      1901        110
      1902          0
```

PostgreSQL

```
CREATE TABLE minx_test (
 id integer,
 geometry sde.st_geometry
);
INSERT INTO minx_test VALUES (
 1901,
 sde.st_polygon ('polygon zm((110 120 20 3, 110 140 22 3, 120 130 26 4, 110 120 20
3))', 4326)
);
INSERT INTO minx_test VALUES (
 1902,
 sde.st_polygon ('polygon zm((0 0 40 7, 0 4 35 9, 5 4 32 12, 5 0 31 5, 0 0 40 7))',
4326)
);
SELECT id, sde.st_minx (geometry)
 AS MinX
 FROM minx_test;
        id       minx
      1901        110
      1902          0
```

SQLite

```
CREATE TABLE minx_test (
 id integer
);
SELECT AddGeometryColumn (
 NULL,
 'minx_test',
 'geometry',
 4326,
 'polygonzm',
 'xyzm',
 'null'
);
INSERT INTO minx_test VALUES (
 1914,
 st_polygon ('polygon zm((110 120 20 3, 110 140 22 3, 120 130 26 4, 110 120 20 3))',
4326)
);
INSERT INTO minx_test VALUES (
 1915,
 st_polygon ('polygon zm((0 0 40 7, 0 4 35 9, 5 4 32 12, 5 0 31 5, 0 0 40 7))', 4326)
);
SELECT id AS "ID", st_minx (geometry) AS "MinX"
 FROM minx_test;
ID       MinX
1914     110.0
1915       0.0
```

ST_MinY

定義

ST_MinY は、ジオメトリを入力パラメーターとして、その最小 Y 座標を返します。

構文

Oracle および PostgreSQL

```
sde.st_miny (geometry1 sde.st_geometry)
```

SQLite

```
st_miny (geometry1 geometryblob)
```

戻り値のタイプ

Oracle および PostgreSQL

Number

SQLite

Double precision

例

miny_test テーブルを作成し、2 つのポリゴンを挿入します。次に、ST_MinY を実行して、各ポリゴンの最小 Y 座標値を判定します。

PostgreSQL

Oracle

```
CREATE TABLE miny_test (
 id integer,
 geometry sde.st_geometry
);
INSERT INTO MINY_TEST VALUES (
 1901,
 sde.st_polygon ('polygon zm((110 120 20 3, 110 140 22 3, 120 130 26 4, 110 120 20
3))', 4326)
);
INSERT INTO MINY_TEST VALUES (
 1902,
 sde.st_polygon ('polygon zm((0 0 40 7, 0 4 35 9, 5 4 32 12, 5 0 31 5, 0 0 40 7))',
4326)
);
SELECT id, sde.st_miny (geometry) MinY
 FROM MINY_TEST;
        ID       MINY
      1901        120
      1902          0
```

PostgreSQL

```
CREATE TABLE miny_test (
 id integer,
 geometry sde.st_geometry
);
INSERT INTO miny_test VALUES (
 1901,
 sde.st_polygon ('polygon zm((110 120 20 3, 110 140 22 3, 120 130 26 4, 110 120 20
3))', 4326)
);
INSERT INTO miny_test VALUES (
 1902,
 sde.st_polygon ('polygon zm((0 0 40 7, 0 4 35 9, 5 4 32 12, 5 0 31 5, 0 0 40 7))',
4326)
);
SELECT id, sde.st_miny (geometry)
 AS MinY
 FROM miny_test;
        id       miny
      1901        120
      1902          0
```

SQLite

```
CREATE TABLE miny_test (
 id integer
);
SELECT AddGeometryColumn (
 NULL,
 'miny_test',
 'geometry',
 4326,
 'polygonzm',
 'xyzm',
 'null'
);
INSERT INTO miny_test VALUES (
 101,
 st_polygon ('polygon zm((110 120 20 3, 110 140 22 3, 120 130 26 4, 110 120 20 3))',
4326)
);
INSERT INTO miny_test VALUES (
 102,
 st_polygon ('polygon zm((0 0 40 7, 0 4 35 9, 5 4 32 12, 5 0 31 5, 0 0 40 7))', 4326)
);
SELECT id, st_miny (geometry)
 AS "MinY"
 FROM miny_test;
id       MinY
101      120.0
102        0.0
```

ST_MinZ

定義

ST_MinZ は、ジオメトリを入力パラメーターとして、その最小 Z 座標を返します。

構文

Oracle および PostgreSQL

```
sde.st_minz (geometry1 sde.st_geometry)
```

SQLite

```
st_minz (geometry1 geometryblob)
```

戻り値のタイプ

Oracle および PostgreSQL

Number

Z 値が存在しない場合、NULL が返されます。

SQLite

Double precision

Z 値が存在しない場合、NULL が返されます。

例

minz_test テーブルを作成し、2 つのポリゴンを挿入します。次に、ST_MinZ を実行して、各ポリゴンの最小 Z 座標値を判定します。

Oracle

```
CREATE TABLE minz_test (
 id integer,
 geometry sde.st_geometry
);
INSERT INTO MINZ_TEST VALUES (
 1901,
 sde.st_polygon ('polygon zm((110 120 20 3, 110 140 22 3, 120 130 26 4, 110 120 20
3))', 4326)
);
INSERT INTO MINZ_TEST VALUES (
 1902,
 sde.st_polygon ('polygon zm((0 0 40 7, 0 4 35 9, 5 4 32 12, 5 0 31 5, 0 0 40 7))',
4326)
);
SELECT id, sde.st_minz (geometry) MinZ
 FROM MINZ_TEST;
        ID       MINZ
```

```
      1901         20
      1902         31
```

PostgreSQL

```
CREATE TABLE minz_test (
 id integer,
 geometry st_geometry
);
INSERT INTO minz_test VALUES (
 1901,
 st_polygon ('polygon zm((110 120 20 3, 110 140 22 3, 120 130 26 4, 110 120 20 3))',
4326)
);
INSERT INTO minz_test VALUES (
 1902,
 st_polygon ('polygon zm((0 0 40 7, 0 4 35 9, 5 4 32 12, 5 0 31 5, 0 0 40 7))', 4326)
);
SELECT id, st_minz (geometry)
 AS MinZ
 FROM minz_test;
        id       minz
      1901         20
      1902         31
```

SQLite

```
CREATE TABLE minz_test (
 id integer
);
SELECT AddGeometryColumn (
 NULL,
 'minz_test',
 'geometry',
 4326,
 'polygonzm',
 'xyzm',
 'null'
);
INSERT INTO minz_test VALUES (
 1901,
 st_polygon ('polygon zm((110 120 20 3, 110 140 22 3, 120 130 26 4, 110 120 20 3))',
4326)
);
INSERT INTO minz_test VALUES (
 1902,
 st_polygon ('polygon zm((0 0 40 7, 0 4 35 9, 5 4 32 12, 5 0 31 5, 0 0 40 7))', 4326)
);
SELECT id, st_minz (geometry)
 AS "MinZ"
 FROM minz_test;
id        MinZ
1901      20.0
1902      31.0
```

ST_MLineFromText

注意:
Oracle および SQLite でのみ使用されます。PostgreSQL では、ST_MultiLineString を使用してください。

定義

ST_MLineFromText は、ST_MultiLineString タイプの WKT 表現と空間参照 ID を受け取って、ST_MultiLineString を返します。

構文

Oracle

```
sde.st_mlinefromtext (wkt clob, srid integer)
```

```
sde.st_mlinefromtext (wkt clob)
```

SRID を指定しない場合、空間参照がデフォルトで 4326 に設定されます。

SQLite

```
st_mlinefromtext (wkt text, srid int32)
```

```
st_mlinefromtext (wkt text)
```

SRID を指定しない場合、空間参照がデフォルトで 4326 に設定されます。

戻り値のタイプ

ST_MultiLineString

例

行を一意に識別する gid smallint 列と ml1 ST_MultiLineString 列を持つ mlinestring_test テーブルを作成します。

INSERT ステートメントは、ST_MLineFromText 関数を使用して ST_MultiLineString を挿入します。

Oracle

```
CREATE TABLE mlinestring_test (
 gid smallint,
 ml1 sde.st_geometry
);
```

```
INSERT INTO MLINESTRING_TEST VALUES (
```

```
 1,
 sde.st_mlinefromtext ('multilinestring ((10.01 20.03, 10.52 40.11, 30.29 41.56,
31.78 10.74), (20.93 20.81, 21.52 40.10))', 4326)
);
```

SQLite

```
CREATE TABLE mlinestring_test (
 gid integer
);
SELECT AddGeometryColumn (
 NULL,
 'mlinestring_test',
 'ml1',
 4326,
 'multilinestring',
 'xy',
 'null'
);
```

```
INSERT INTO MLINESTRING_TEST VALUES (
 1,
 st_mlinefromtext ('multilinestring ((10.01 20.03, 10.52 40.11, 30.29 41.56,
31.78 10.74), (20.93 20.81, 21.52 40.10))', 4326)
);
```

ST_MLineFromWKB

定義

ST_MLineFromWKB は、ST_MultiLineString タイプの WKB 表現と空間参照 ID を使用して、ST_MultiLineString を作成します。

構文

Oracle

```
sde.st_mlinefromwkb (wkb blob, srid integer)
```

```
sde.st_mlinefromwkb (wkb blob)
```

SRID を指定しない場合、空間参照がデフォルトで 4326 に設定されます。

PostgreSQL

```
sde.st_mlinefromwkb (wkb bytea, srid integer)
```

SQLite

```
st_mlinefromwkb (wkb blob, srid int32)
```

```
st_mlinefromwkb (wkb blob)
```

SRID を指定しない場合、空間参照がデフォルトで 4326 に設定されます。

戻り値のタイプ

ST_MultiLineString

例

この例は、ST_MLineFromWKB を使用して、WKB 表現からマルチラインストリングを作成する方法を示したものです。ジオメトリは、空間参照系 4326 のマルチラインストリングです。この例では、マルチラインストリングを sample_mlines テーブルの geometry 列に ID = 10 で格納した後、wkb 列を (ST_AsBinary 関数を使用して) WKB表現で更新しています。最後に、ST_MLineFromWKB 関数を使用して、wkb 列からマルチラインストリングを返します。sample_mlines テーブルには、マルチラインストリングを格納する geometry 列と、マルチラインストリングの WKB 表現を格納する wkb 列があります。

SELECT ステートメントには ST_MLineFromWKB 関数が含まれています。この関数を使用して、wkb 列からマルチラインストリングを取得します。

Oracle

```
CREATE TABLE sample_mlines (
 id integer,
 geometry sde.st_geometry,
 wkb blob
);
INSERT INTO SAMPLE_MLINES (id, geometry) VALUES (
 10,
 sde.st_multilinestring ('multilinestring ((61 2, 64 3, 65 6), (58 4, 59 5, 61 8), (69
3, 67 4, 66 7, 68 9))', 4326)
);
UPDATE SAMPLE_MLINES
 SET wkb = sde.st_asbinary (geometry)
 WHERE id = 10;
```

```
SELECT id, sde.st_astext (sde.st_mlinefromwkb (wkb,0)) MULTI_LINE_STRING
 FROM SAMPLE_MLINES
 WHERE id = 10;
ID    MULTI_LINE_STRING
10    MULTILINESTRING ((61.00000000 2.00000000, 64.00000000 3.00000000, 65.00000000
6.00000000), (58.00000000 4.00000000, 59.00000000 5.00000000, 61.00000000 8.0000000),
(69.00000000 3.00000000, 67.00000000 4.00000000, 66.00000000 7.00000000, 68.00000000
9.00000000 ))
```

PostgreSQL

```
CREATE TABLE sample_mlines (
 id integer,
 geometry sde.st_geometry,
 wkb bytea);
INSERT INTO sample_mlines (id, geometry) VALUES (
 10,
 sde.st_multilinestring ('multilinestring ((61 2, 64 3, 65 6), (58 4, 59 5, 61 8), (69
3, 67 4, 66 7, 68 9))', 4326)
);
UPDATE sample_mlines
 SET wkb = sde.st_asbinary (geometry)
 WHERE id = 10;
```

```
SELECT id, sde.st_astext (sde.st_mlinefromwkb (wkb,4326))
 AS MULTI_LINE_STRING
 FROM sample_mlines
 WHERE id = 10;
id   multi_line_string
10    MULTI_LINE_STRING ((61 2, 64 3, 65 6), (58 4, 59 5,61 8), (69 3, 67 4, 66 7, 68 9
))
```

SQLite

```
CREATE TABLE sample_mlines (
 id integer,
 wkb blob);
SELECT AddGeometryColumn (
 NULL,
```

```
 'sample_mlines',
 'geometry',
 4326,
 'multilinestring',
 'xy',
 'null'
);
INSERT INTO sample_mlines (id, geometry) VALUES (
 10,
 st_multilinestring ('multilinestring ((61 2, 64 3, 65 6), (58 4, 59 5, 61 8), (69 3,
67 4, 66 7, 68 9))', 4326)
);
UPDATE sample_mlines
 SET wkb = st_asbinary (geometry)
 WHERE id = 10;
```

```
SELECT id, st_astext (st_mlinefromwkb (wkb,4326))
 AS MULTI_LINE_STRING
 FROM sample_mlines
 WHERE id = 10;
id   multi_line_string
10    MULTI_LINE_STRING ((61.00000000 2.00000000, 64.00000000 3.00000000, 65.00000000
6.00000000),
 (58.00000000 4.00000000, 59.00000000 5.00000000, 61.00000000 8.00000000),
 (69.00000000 3.00000000, 67.00000000 4.00000000, 66.00000000 7.00000000, 68.00000000
9.00000000 ))
```

ST_MPointFromText

注意:
Oracle および SQLite のみ。PostgreSQL の場合、ST_MultiPoint を使用します。

定義

ST_MPointFromText は、ST_MultiPoint タイプの WKT 表現と空間参照 ID を受け取って、ST_Multipoint を作成します。

構文

Oracle

```
sde.st_mpointfromtext (wkt clob, srid integer)
```

```
sde.st_mpointfromtext (wkt clob)
```

SRID を指定しない場合、空間参照がデフォルトで 4326 に設定されます。

SQLite

```
st_mpointfromtext (wkt text, srid int32)
```

```
st_mpointfromtext (wkt text)
```

SRID を指定しない場合、空間参照がデフォルトで 4326 に設定されます。

戻り値のタイプ

ST_MultiPoint

例

ST_MultiPoint の mpt1 列を 1 つ持つ multipoint_test テーブルを作成します。

INSERT ステートメントは、ST_MpointFromText 関数を使用して、マルチポイントを mpt1 列に挿入します。

Oracle

```
CREATE TABLE multipoint_test (mpt1 sde.st_geometry);
```

```
INSERT INTO MULTIPOINT_TEST VALUES (
 sde.st_mpointfromtext ('multipoint ((10.01 20.03), (10.52 40.11), (30.29 41.56),
(31.78 10.74))', 4326));
```

SQLite

```
CREATE TABLE multipoint_test (id integer);

SELECT AddGeometryColumn (
 NULL,
 'multipoint_test',
 'pt1',
 4326,
 'multipoint',
 'xy',
 'null'
);
```

```
INSERT INTO MULTIPOINT_TEST VALUES (
 1,
 st_mpointfromtext ('multipoint ((10.01 20.03), (10.52 40.11), (30.29 41.56), (31.78
10.74))', 4326));
```

ST_MPointFromWKB

定義

ST_MPointFromText は、ST_MultiPoint タイプの WKB 表現と空間参照 ID を受け取って、ST_MultiPoint を作成します。

構文

Oracle

```
sde.st_mpointfromwkb (wkb blob, srid integer)
```

```
sde.st_mpointfromwkb (wkb blob)
```

SRID を指定しない場合、空間参照がデフォルトで 4326 に設定されます。

PostgreSQL

```
sde.st_mpointfromwkb (wkb bytea, srid integer)
```

SQLite

```
st_mpointfromwkb (wkb blob, srid int32)
```

```
st_mpointfromwkb (wkb blob)
```

SRID を指定しない場合、空間参照がデフォルトで 4326 に設定されます。

戻り値のタイプ

ST_MultiPoint

例

この例では、ST_MPointFromWKB を使用して、WKB (Well-Known Binary) 表現からマルチポイントを作成する方法を説明します。 ジオメトリは、空間参照系 4326 のマルチポイントです。 この例では、SAMPLE_MPOINTS テーブルの GEOMETRY 列に ID = 10 のマルチポイントを格納し、WKB 列が Well-Known Binary 表現で更新されます (ST_AsBinary 関数を使用)。 最後に、ST_MPointFromWKB 関数を使用して、WKB 列からマルチポイントを返します。 SAMPLE_MPOINTS テーブルには、マルチポイントを格納する GEOMETRY 列と、マルチポイントの WKB 表現を格納する WKB 列があります。

次の SELECT ステートメントで、ST_MPointFromWKB 関数を使用して、WKB 列からマルチポイントを取得します。

Oracle

```
CREATE TABLE sample_mpoints (
 id integer,
 geometry sde.st_geometry,
 wkb blob
);

INSERT INTO SAMPLE_MPOINTS (id, geometry) VALUES (
 10,
 sde.st_multipoint ('multipoint ((4 14), (35 16), (24 13))', 4326)
);

UPDATE SAMPLE_MPOINTS
 SET wkb = sde.st_asbinary (geometry)
 WHERE id = 10;
```

```
SELECT id, sde.st_astext (sde.st_mpointfromwkb (wkb,4326)) MULTI_POINT
 FROM SAMPLE_MPOINTS
 WHERE id = 10;


ID           MULTI_POINT

10    MULTIPOINT ((4.00000000 14.00000000), (35.00000000 16.00000000), (24.00000000
13.00000000))
```

PostgreSQL

```
CREATE TABLE sample_mpoints (
 id integer,
 geometry sde.st_geometry,
 wkb bytea
);

INSERT INTO sample_mpoints (id, geometry) VALUES (
 10,
 sde.st_multipoint ('multipoint (4 14, 35 16, 24 13)', 4326)
);

UPDATE sample_mpoints
 SET wkb = sde.st_asbinary (geometry)
 WHERE id = 10;
```

```
SELECT id, sde.st_astext (sde.st_mpointfromwkb (wkb,4326))
 AS "MULTI_POINT"
 FROM sample_mpoints
 WHERE id = 10;


id           MULTI_POINT

10     MULTIPOINT (4 14, 35 16, 24 13)
```

SQLite

```
CREATE TABLE sample_mpoints (
 id integer,
 wkb blob
);

SELECT AddGeometryColumn (
 NULL,
 'sample_mpoints',
 'geometry',
 4326,
 'multipointzm',
 'xyzm',
 'null'
);

INSERT INTO SAMPLE_MPOINTS (id, geometry) VALUES (
 10,
 st_multipoint ('multipoint ((4 14), (35 16), (24 13))', 4326)
);

UPDATE sample_mpoints
 SET wkb = st_asbinary (geometry)
 WHERE id = 10;
```

```
SELECT id AS "ID",
 st_astext (st_mpointfromwkb (wkb,4326))
 AS "MULTI_POINT"
 FROM sample_mpoints
 WHERE id = 10;


ID           MULTI_POINT

10    MULTIPOINT ((4.00000000 14.00000000), (35.00000000 16.00000000), (24.00000000
13.00000000))
```

ST_MPolyFromText

注意:
Oracle および SQLite のみ。PostgreSQL の場合、ST_MultiPolygon を使用します。

定義

ST_MPointFromText は、ST_MultiPolygon タイプの WKT 表現と空間参照 ID を受け取って、ST_MultiPolygon を返します。

構文

SRID を指定しない場合、空間参照がデフォルトで 4326 に設定されます。

Oracle

```
sde.st_mpolyfromtext (wkt clob, srid integer)
```

```
sde.st_mpolyfromtext (wkt clob)
```

SQLite

```
st_mpolyfromtext (wkt text, srid int32)
```

```
st_mpolyfromtext (wkt text)
```

戻り値のタイプ

ST_MultiPolygon

例

ST_MultiPolygon の mpl1 列を 1 つ持つ multipolygon_test テーブルを作成します。

INSERT ステートメントは、ST_MpolyFromText 関数を使用して、ST_MultiPolygon を mpl1 列に挿入します。

Oracle

```
CREATE TABLE mpolygon_test (mpl1 sde.st_geometry);
```

```
INSERT INTO MPOLYGON_TEST VALUES (
 sde.st_mpolyfromtext ('multipolygon (((10.01 20.03, 10.52 40.11, 30.29 41.56,
31.78 10.74, 10.01 20.03), (21.23 15.74, 21.34 35.21, 28.94 35.35,
29.02 16.83, 21.23 15.74)), ((40.91 10.92, 40.56 20.19, 50.01 21.12,
51.34 9.81, 40.91 10.92)))', 4326)
);
```

SQLite

```
CREATE TABLE mpolygon_test (id integer);

SELECT AddGeometryColumn(
 NULL,
 'mpolygon_test',
 'mpl1',
 4326,
 'multipolygon',
 'xy',
 'null'
);
```

```
INSERT INTO MPOLYGON_TEST VALUES (
 1,
 st_mpolyfromtext ('multipolygon (((10.01 20.03, 10.52 40.11, 30.29 41.56,
31.78 10.74, 10.01 20.03), (21.23 15.74, 21.34 35.21, 28.94 35.35,
29.02 16.83, 21.23 15.74)), ((40.91 10.92, 40.56 20.19, 50.01 21.12,
51.34 9.81, 40.91 10.92)))', 4326)
);
```

ST_MPolyFromWKB

定義

ST_MPointFromWKB は、ST_MultiPolygon タイプの WKB 表現と空間参照 ID を受け取って、ST_MultiPolygon を返します。

構文

Oracle

```
sde.st_mpolyfromwkb (wkb blob, srid integer)
```

```
sde.st_mpolyfromwkb (wkb blob)
```

SRID を指定しない場合、空間参照がデフォルトで 4326 に設定されます。

PostgreSQL

```
sde.st_mpolyfromwkb (wkb bytea, srid integer)
```

SQLite

```
st_mpolyfromwkb (wkb blob, srid int32)
```

```
st_mpolyfromwkb (wkb blob)
```

SRID を指定しない場合、空間参照がデフォルトで 4326 に設定されます。

戻り値のタイプ

ST_MultiPolygon

例

この例では、ST_MPolyFromWKB を使用して、その WKB 表現からマルチポリゴンを作成する方法を示しています。ジオメトリは、空間参照系 4326 のマルチポリゴンです。この例では、マルチポリゴンを sample_mpolys テーブルの geometry 列に ID = 10 で格納した後、wkb 列を (ST_AsBinary 関数を使用して) WKB 表現で更新しています。最後に、ST_MPolyFromWKB 関数を使用して、wkb 列からマルチポリゴンを返します。sample_mpolys テーブルには、マルチポリゴンを格納する geometry 列と、マルチポリゴンの WKB 表現を格納する wkb 列があります。

SELECT ステートメントには ST_MPolyFromWKB 関数が含まれています。この関数を使用して、WKB 列からマルチポリゴンを取得します。

Oracle

```
CREATE TABLE sample_mpolys (
 id integer,
 geometry sde.st_geometry,
 wkb blob
);
INSERT INTO SAMPLE_MPOLYS (id, geometry) VALUES (
 10,
 sde.st_multipolygon ('multipolygon (((1 72, 4 79, 5 76, 1 72), (10 20, 10 40, 30 41,
10 20), (9 43, 7 44, 6 47, 9 43)))', 4326)
);
UPDATE SAMPLE_MPOLYS
 SET wkb = sde.st_asbinary (geometry)
 WHERE id = 10;
```

```
SELECT id, sde.st_astext (sde.st_mpolyfromwkb (wkb,4326)) MULTIPOLYGON
 FROM SAMPLE_MPOLYS
 WHERE id = 10;
ID   MULTIPOLYGON
10   MULTIPOLYGON (((10.00000000 20.00000000, 30.00000000 41.00000000, 10.00000000
40.00000000, 10.00000000 20.00000000)), (1.00000000 72.00000000, 5.00000000
76.00000000, 4.00000000 79.0000000, 1.00000000 72,00000000)), (9.00000000 43.00000000,
6.00000000 47.00000000, 7.00000000 44.00000000, 9.00000000 43.00000000 )))
```

PostgreSQL

```
CREATE TABLE sample_mpolys (
 id integer,
 geometry sde.st_geometry,
 wkb bytea
);
INSERT INTO sample_mpolys (id, geometry) VALUES (
 10,
 sde.st_multipolygon ('multipolygon (((1 72, 4 79, 5 76, 1 72), (10 20, 10 40, 30 41,
10 20), (9 43, 7 44, 6 47, 9 43)))', 4326)
);
UPDATE sample_mpolys
 SET wkb = sde.st_asbinary (geometry)
 WHERE id = 10;
```

```
SELECT id, sde.st_astext (sde.st_mpolyfromwkb (wkb,4326))
 AS MULTIPOLYGON
 FROM sample_mpolys
 WHERE id = 10;
id   multipolygon
10   MULTIPOLYGON (((10 20, 30 41, 10 40, 10 20)),
 ((1 72, 5 76, 4 79, 1 72)), ((9 43, 6 47, 7 44, 9 43)))
```

SQLite

```
CREATE TABLE sample_mpolys (
 id integer,
 wkb blob
);
```

```
SELECT AddGeometryColumn(
 NULL,
 'sample_mpolys',
 'geometry',
 4326,
 'multipolygon',
 'xy',
 'null'
);
INSERT INTO SAMPLE_MPOLYS (id, geometry) VALUES (
 10,
 st_multipolygon ('multipolygon (((1 72, 4 79, 5 76, 1 72), (10 20, 10 40, 30 41, 10
20), (9 43, 7 44, 6 47, 9 43)))', 4326)
);
UPDATE SAMPLE_MPOLYS
 SET wkb = st_asbinary (geometry)
 WHERE id = 10;
```

```
SELECT id, st_astext (st_mpolyfromwkb (wkb,4326))
 AS "Multipolygon"
 FROM sample_mpolys
 WHERE id = 10;
id   Multipolygon
10   MULTIPOLYGON ((( 10.00000000 20.00000000, 30.00000000 41.00000000, 10.00000000
40.00000000, 10.00000000 20.00000000)),
 ((1.00000000 72.00000000, 5.00000000 76.00000000, 4.00000000 79.00000000, 1.00000000
72.00000000)),
 ((9.00000000 43.00000000, 6.00000000 47.00000000, 7.00000000 44.00000000, 9.00000000
43.00000000)))
```

ST_MultiCurve

注意:
Oracle のみ

定義

ST_MultiCurve は、WKT 表現からマルチカーブ フィーチャを構築します。

構文

```
sde.st_multicurve (wkt clob, srid integer)
```

戻り値のタイプ

ST_MultiLinestring

例

```
CREATE TABLE mcurve_test (id integer, geometry sde.st_geometry);

INSERT INTO MCURVE_TEST VALUES (
1910,
sde.st_multicurve ('multilinestring ((33 2, 34 3, 35 6),
(28 4, 29 5, 31 8, 43 12), (39 3, 37 4, 36 7))', 0)
);

SELECT sde.st_astext (geometry) MCURVE
FROM MCURVE_TEST;

ID        MCURVE

1110      MULTILINESTRING ((33.00000000 2.00000000, 34.00000000
           3.00000000, 35.00000000 6.00000000), (28.00000000 4.00000000,
           29.00000000 5.00000000, 31.00000000 8.0000000, 43.00000000
           12.00000000), (39.00000000 3.00000000, 37.00000000 4.00000000,
           36.00000000 7.00000000 ))
```

ST_MultiLineString

定義

ST_MultiLineString は、WKT 表現からマルチラインストリングを構築します。

注意:
ArcGIS で使用する空間テーブルを作成する場合は、ST_Geometry サブタイプを指定するよりも、ジオメトリ スーパータイプ (たとえば、ST_Geometry) として列を作成することをお勧めします。

構文

Oracle

```
sde.st_multilinestring (wkt clob, srid integer)
```

PostgreSQL

```
sde.st_multilinestring (wkt clob, srid integer)
sde.st_multilinestring (esri_shape bytea, srid integer)
```

SQLite

```
st_multilinestring (wkt text, srid int32)
```

戻り値のタイプ

ST_MultiLineString

例

mlines_test テーブルを作成し、ST_MultiLineString 関数を使用して 1 つのマルチラインをこのテーブルに挿入します。

Oracle

```
CREATE TABLE mlines_test (
 id integer,
 geometry sde.st_geometry
);

INSERT INTO MLINES_TEST VALUES (
 1910,
 sde.st_multilinestring ('multilinestring ((33 2, 34 3, 35 6), (28 4, 29 5, 31 8, 43
12), (39 3, 37 4, 36 7))', 4326)
);
```

PostgreSQL

```
CREATE TABLE mlines_test (
 id integer,
 geometry sde.st_geometry
);

INSERT INTO mlines_test VALUES (
 1910,
 sde.st_multilinestring ('multilinestring ((33 2, 34 3, 35 6), (28 4, 29 5, 31 8, 43
12), (39 3, 37 4, 36 7))', 4326)
);
```

SQLite

```
CREATE TABLE mlines_test (
 id integer
);

SELECT AddGeometryColumn(
 NULL,
 'mlines_test',
 'geometry',
 4326,
 'multilinestring',
 'xy',
 'null'
);

INSERT INTO mlines_test VALUES (
 1910,
 st_multilinestring ('multilinestring ((33 2, 34 3, 35 6), (28 4, 29 5, 31 8, 43 12),
(39 3, 37 4, 36 7))', 4326)
);
```

ST_MultiPoint

定義

ST_MultiPoint は、WKT 表現からマルチポイント フィーチャを構築します。

注意:
ArcGIS で使用する空間テーブルを作成する場合は、ST_Geometry サブタイプを指定するよりも、ジオメトリ スーパータイプ (たとえば、ST_Geometry) として列を作成することをお勧めします。

構文

Oracle

```
sde.st_multipoint (wkt clob, srid integer)
```

PostgreSQL

```
sde.st_multipoint (wkt clob, srid integer)
sde.st_multipoint (esri_shape bytea, srid integer)
```

SQLite

```
st_multipoint (wkt text, srid int32)
```

戻り値のタイプ

ST_MultiPoint

例

mpoint_test テーブルを作成し、ST_MultiPoint 関数を使用して 1 つのマルチポイントをこのテーブルに挿入します。

Oracle

```
CREATE TABLE mpoint_test (
 id integer,
 geometry sde.st_geometry
);

INSERT INTO MPOINT_TEST VALUES (
 1110,
 sde.st_multipoint ('multipoint ((1 2), (3 4), (5 6))', 4326)
);
```

PostgreSQL

```
CREATE TABLE mpoint_test (
 id integer,
 geometry sde.st_geometry
);

INSERT INTO mpoint_test VALUES (
 1110,
 sde.st_multipoint ('multipoint (1 2, 3 4, 5 6)', 4326)
);
```

SQLite

```
CREATE TABLE mpoint_test (
 id integer
);

SELECT AddGeometryColumn(
 NULL,
 'mpoint_test',
 'geometry',
 4326,
 'multipoint',
 'xy',
 'null'
);

INSERT INTO mpoint_test VALUES (
 1110,
 st_multipoint ('multipoint ((1 2), (3 4), (5 6))', 4326)
);
```

ST_MultiPolygon

定義

ST_MultiPolygon は、WKT 表現からマルチポリゴン フィーチャを構築します。

注意:
ArcGIS で使用する空間テーブルを作成する場合は、ST_Geometry サブタイプを指定するよりも、ジオメトリ スーパータイプ (たとえば、ST_Geometry) として列を作成することをお勧めします。

構文

Oracle

```
sde.st_multipolygon (wkt clob, srid integer)
```

PostgreSQL

```
sde.st_multipolygon (wkt clob, srid integer)
sde.st_multipolygon (esri_shape bytea, srid integer)
```

SQLite

```
st_multipolygon (wkt text, srid int32)
```

戻り値のタイプ

ST_MultiPolygon

例

mpoly_test テーブルを作成し、ST_MultiPolygon 関数を使用して 1 つのマルチポリゴンをこのテーブルに挿入します。

Oracle

```
CREATE TABLE mpoly_test (
 id integer,
 geometry sde.st_geometry
);

INSERT INTO MPOLY_TEST VALUES (
 1110,
 sde.st_multipolygon ('multipolygon (((3 3, 4 6, 5 3, 3 3),(8 24, 9 25, 1 28, 8 24),
(13 33, 7 36, 1 40, 10 43, 13 33)))', 4326)
);
```

PostgreSQL

```
CREATE TABLE mpoly_test (
 id integer,
 geometry sde.st_geometry
);

INSERT INTO mpoly_test VALUES (
 1110,
 sde.st_multipolygon ('multipolygon (((3 3, 4 6, 5 3, 3 3), (8 24, 9 25, 1 28, 8 24),
(13 33, 7 36, 1 40, 10 43, 13 33)))', 4326)
);
```

SQLite

```
CREATE TABLE mpoly_test (
 id integer
);

SELECT AddGeometryColumn(
 NULL,
 'mpoly_test',
 'geometry',
 4326,
 'multipolygon',
 'xy',
 'null'
);

INSERT INTO mpoly_test VALUES (
 1110,
 st_multipolygon ('multipolygon (((3 3, 4 6, 5 3, 3 3), (8 24, 9 25, 1 28, 8 24), (13
33, 7 36, 1 40, 10 43, 13 33)))', 4326)
);
```

ST_MultiSurface

注意:
Oracle のみ

定義

ST_MultiSurface は、WKT 表現からマルチサーフェス フィーチャを構築します。

構文

```
sde.st_multisurface (wkt clob, srid integer)
```

戻り値のタイプ

ST_MultiSurface

例

```
CREATE TABLE msurf_test (id integer, geometry sde.st_geometry);

INSERT INTO MSURF_TEST VALUES (
1110,
sde.st_multisurface ('multipolygon (((3 3, 4 6, 5 3, 3 3),(8 24, 9 25, 1 28, 8 24), (13
33, 7 36, 1 40, 10 43, 13 33)))', 0)
);

SELECT id, sde.st_astext (geometry) MULTI_SURFACE
FROM MSURF_TEST
WHERE id = 1110;

     ID    MULTI_SURFACE

   1110    MULTIPOLYGON (((13.00000000 33.00000000, 10.00000000
           43.00000000, 1.00000000 40.00000000, 7.00000000 36.00000000,
           13.00000000 33.00000000)), ((8.00000000 24.00000000, 9.00000000
           25.00000000, 1.00000000 28.0000000, 8.00000000 24.00000000)),
           ((3.00000000 3.00000000, 5.00000000 3.00000000,
           4.00000000 6.00000000, 3.00000000 3.00000000)))
```

ST_NumGeometries

定義

ST_NumGeometries はジオメトリのコレクションを入力として、コレクション内のジオメトリ数を返します。

構文

Oracle

```
sde.st_numgeometries (multipoint1 sde.st_geometry)
sde.st_numgeometries (multiline1 sde.st_geometry)
sde.st_numgeometries (multipolygon1 sde.st_geometry)
```

PostgreSQL

```
sde.st_numgeometries (geometry1 sde.st_geomcollection)
```

SQLite

```
st_numgeometries (geometry1 geometryblob)
```

戻り値のタイプ

Integer

例

次の例では、sample_numgeom という名前のテーブルが作成されます。 1 つのマルチポリゴンと 1 つのマルチポイントをテーブルに挿入します。 SELECT ステートメントでは、ST_NumGeometries 関数は各ジオメトリ内のジオメトリ (またはフィーチャ) の数を確認するために使用されます。

Oracle

```
CREATE TABLE sample_numgeom (
 id integer,
 geometry sde.st_geometry
);

INSERT INTO SAMPLE_NUMGEOM VALUES (
 1,
 sde.st_multipolygon ('multipolygon (((3 3, 4 6, 5 3, 3 3), (8 24, 9 25, 1 28, 8 24),
(13 33, 7 36, 1 40, 10 43, 13 33)))', 4326)
);

INSERT INTO SAMPLE_NUMGEOM VALUES (
 2,
 sde.st_multipoint ('multipoint ((1 2), (4 3), (5 6), (7 6), (8 8))', 4326)
);

SELECT id, sde.st_numgeometries (geometry) NUM_GEOMS_IN_COLL
 FROM SAMPLE_NUMGEOM;
```

```
ID          NUM_GEOMS_IN_COLL

  1                 3
  2                 5
```

PostgreSQL

```
CREATE TABLE sample_numgeom (
 id integer,
 geometry sde.st_geometry
);

INSERT INTO sample_numgeom VALUES (
 1,
 sde.st_multipolygon ('multipolygon (((3 3, 4 6, 5 3, 3 3), (8 24, 9 25, 1 28, 8 24),
(13 33, 7 36, 1 40, 10 43, 13 33)))', 4326)
);

INSERT INTO sample_numgeom VALUES (
 2,
 sde.st_multipoint ('multipoint (1 2, 4 3, 5 6, 7 6, 8 8)', 4326)
);

SELECT id, sde.st_numgeometries (geometry)
 AS "number of geometries"
 FROM sample_numgeom;

id          number of geometries

  1                 3
  2                 5
```

SQLite

```
CREATE TABLE sample_numgeom (
 id integer
);

SELECT AddGeometryColumn(
 NULL,
 'sample_numgeom',
 'geometry',
 4326,
 'geometry',
 'xy',
 'null'
);

INSERT INTO sample_numgeom VALUES (
 1,
 st_multipolygon ('multipolygon (((3 3, 4 6, 5 3, 3 3), (8 24, 9 25, 1 28, 8 24), (13
33, 7 36, 1 40, 10 43, 13 33)))', 4326)
);

INSERT INTO sample_numgeom VALUES (
 2,
 st_multipoint ('multipoint ((1 2), (4 3), (5 6), (7 6), (8 8))', 4326)
);
```

```
SELECT id, st_numgeometries (geometry)
 AS "number of geometries"
 FROM sample_numgeom;

id          number of geometries

  1                 3
  2                 5
```

ST_NumInteriorRing

定義

ST_NumInteriorRing は、ST_Polygon を入力として、その内部リングの数を返します。

構文

Oracle および PostgreSQL

```
sde.st_numinteriorring (polygon1 sde.st_geometry)
```

SQLite

```
st_numinteriorring (polygon1 geometryblob)
```

戻り値のタイプ

Integer

例

鳥類学者は、南海の島々に生息する鳥を調査したいと考えています。そして、目的の鳥類の餌場が淡水の湖に限られるため、1 つ以上の湖がある島を特定したいと考えています。

islands テーブルの ID 列と name 列は各島を識別します。land ST_Polygon 列は、島のジオメトリを格納します。

内部リングは湖を表すため、ST_NumInteriorRing 関数を含む SELECT ステートメントは、少なくとも 1 つの内部リングを持つ島だけをリストします。

Oracle

```
CREATE TABLE islands (
 id integer,
 name varchar(32),
 land sde.st_geometry
);
INSERT INTO islands VALUES (
 1,
 'Bear',
 sde.st_polygon ('polygon ((40 120, 90 120, 90 150, 40 150, 40 120),(50 130, 60 130, 60
140, 50 140, 50 130),(70 130, 80 130, 80 140, 70 140, 70 130))', 4326)
);
INSERT INTO islands VALUES (
 2,
 'Johnson',
 sde.st_polygon ('polygon ((10 10, 50 10, 10 30, 10 10))', 4326)
);
```

```
SELECT name
 FROM ISLANDS
 WHERE sde.st_numinteriorring (land)> 0;
NAME
```

```
Bear
```

PostgreSQL

```
CREATE TABLE islands (
 id integer,
 name varchar(32),
 land sde.st_geometry
);
INSERT INTO islands VALUES (
 1,
 'Bear',
 sde.st_polygon ('polygon ((40 120, 90 120, 90 150, 40 150, 40 120),(50 130, 60 130, 60
140, 50 140, 50 130),(70 130, 80 130, 80 140, 70 140, 70 130))', 4326)
);
INSERT INTO islands VALUES (
 2,
 'Johnson',
 sde.st_polygon ('polygon ((10 10, 50 10, 10 30, 10 10))', 4326)
);
```

```
SELECT name
 FROM islands
 WHERE sde.st_numinteriorring (land)> 0;
name
Bear
```

SQLite

```
CREATE TABLE islands (
 id integer,
 name varchar(32)
);
SELECT AddGeometryColumn(
 NULL,
 'islands',
 'land',
 4326,
 'polygon',
 'xy',
 'null'
);
INSERT INTO islands VALUES (
 1,
 'Bear',
 st_polygon ('polygon ((40 120, 90 120, 90 150, 40 150, 40 120),(50 130, 60 130, 60
140, 50 140, 50 130),(70 130, 80 130, 80 140, 70 140, 70 130))', 4326)
);
INSERT INTO islands VALUES (
 2,
 'Johnson',
 st_polygon ('polygon ((10 10, 50 10, 10 30, 10 10))', 4326)
);
```

```
SELECT name
```

```
 FROM islands
 WHERE st_numinteriorring (land)> 0;
name
Bear
```

ST_NumPoints

定義

ST_NumPoints は、ジオメトリ内にあるポイント (頂点) の数を返します。

ポリゴンの場合、始点の頂点と終点の頂点が同じ位置にあっても、別々にカウントされます。

この数値は、ST_Geometry タイプの NUMPTS 属性とは違いますので注意してください。NUMPTS 属性では、パート間のセパレーターを含む、ジオメトリのすべての部分における頂点がカウントされます。各パートの間には 1 つのセパレーターがあります。たとえば、3 つの部分があるマルチパート ラインストリングには 2 つのセパレーターがあります。NUMPTS 属性では、各セパレーターは 1 つの頂点としてカウントされます。それに対して ST_NumPoints 関数では、セパレーターは頂点としてカウントされません。

構文

Oracle および PostgreSQL

```
sde.st_numpoints (geometry1 sde.st_geometry)
```

SQLite

```
st_numpoints (geometry1 geometryblob)
```

戻り値のタイプ

Integer

例

geotype 列と、geometry タイプを格納する g1 列を持つ numpoints_test テーブルを作成します。

INSERT ステートメントは、ポイント、ラインストリング、ポリゴンを挿入します。

SELECT クエリは ST_NumPoints 関数を使用して、各フィーチャ タイプについて、各フィーチャ内にあるポイントの数を取得します。

Oracle

```
CREATE TABLE numpoints_test (
 geotype varchar(12),
 g1 sde.st_geometry
);
INSERT INTO NUMPOINTS_TEST VALUES (
 'point',
 sde.st_pointfromtext ('point (10.02 20.01)', 4326)
);
INSERT INTO NUMPOINTS_TEST VALUES (
 'linestring',
 sde.st_linefromtext ('linestring (10.02 20.01, 23.73 21.92)', 4326)
);
INSERT INTO NUMPOINTS_TEST VALUES (
 'polygon',
 sde.st_polyfromtext ('polygon ((10.02 20.01, 23.73 21.92, 24.51 12.98, 11.64 13.42,
```

```
10.02 20.01))', 4326)
);
```

```
SELECT geotype, sde.st_numpoints (g1) Number_of_points
 FROM NUMPOINTS_TEST;
GEOTYPE     Number_of_points
point                       1
linestring                  2
polygon                     5
```

PostgreSQL

```
CREATE TABLE numpoints_test (
 geotype varchar(12),
 g1 sde.st_geometry
);
INSERT INTO numpoints_test VALUES (
 'point',
 sde.st_point ('point (10.02 20.01)', 4326)
);
INSERT INTO numpoints_test VALUES (
 'linestring',
 sde.st_linestring ('linestring (10.02 20.01, 23.73 21.92)', 4326)
);
INSERT INTO numpoints_test VALUES (
 'polygon',
 sde.st_polygon ('polygon ((10.02 20.01, 23.73 21.92, 24.51 12.98, 11.64 13.42, 10.02
20.01))', 4326)
);
```

```
SELECT geotype, sde.st_numpoints (g1)
 AS Number_of_points
 FROM numpoints_test;
geotype     number_of_points
point                       1
linestring                  2
polygon                     5
```

SQLite

```
CREATE TABLE numpoints_test (
 geotype text(12)
);
SELECT AddGeometryColumn(
 NULL,
 'numpoints_test',
 'g1',
 4326,
 'geometry',
 'xy',
 'null'
);
INSERT INTO numpoints_test VALUES (
 'point',
 st_point ('point (10.02 20.01)', 4326)
```

```
);
INSERT INTO numpoints_test VALUES (
 'linestring',
 st_linestring ('linestring (10.02 20.01, 23.73 21.92)', 4326)
);
INSERT INTO numpoints_test VALUES (
 'polygon',
 st_polygon ('polygon ((10.02 20.01, 23.73 21.92, 24.51 12.98, 11.64 13.42, 10.02
20.01))', 4326)
);
```

```
SELECT geotype AS "Type of geometry", st_numpoints (g1) AS "Number of points"
 FROM numpoints_test;
Type of geometry     Number of points
point                       1
linestring                  2
polygon                     5
```

ST_OrderingEquals

注意:
Oracle と PostgreSQL のみ

定義

ST_OrderingEquals は、2 つの ST_Geometry を比較し、ジオメトリが等しく、座標が同じ順序の場合は 1 (Oracle) または t (PostgreSQL)、それ以外の場合は 0 (Oracle) または f (PostgreSQL) を返します。

構文

```
sde.st_orderingequals (g1 sde.st_geometry, g2 sde.st_geometry)
```

戻り値のタイプ

Boolean

例

Oracle

次の CREATE TABLE ステートメントは、ln1 と ln2 という 2 つのラインストリング列を持つ LINESTRING_TEST テーブルを作成します。

```
CREATE TABLE linestring_test (
lid integer,
ln1 sde.st_geometry,
ln2 sde.st_geometry);
```

次の INSERT ステートメントは、2 つの ST_LineString 値を ln1 と ln2 に挿入します。これらは同じで、座標の順序も同じです。

```
INSERT INTO LINESTRING_TEST VALUES (
1,
sde.st_geometry ('linestring (10.01 20.02, 21.50 12.10)', 0),
sde.st_geometry ('linestring (21.50 12.10, 10.01 20.02)', 0)
);
```

次の SELECT ステートメントとそれに対応する結果セットは、ST_Equals 関数が座標の順序に関係なく 1 (真) を返す様子を示しています。 ST_OrderingEquals 関数は、ジオメトリが等しく座標の順序が同じであるという条件を満たしていない場合は 0 (偽) を返します。

```
SELECT lid, sde.st_equals (ln1, ln2) Equals, sde.st_orderingequals (ln1, ln2)
OrderingEquals
FROM LINESTRING_TEST;

lid Equals     OrderingEquals

1   1          0
```

PostgreSQL

次の CREATE TABLE ステートメントは、ln1 と ln2 という 2 つのラインストリング列を持つ LINESTRING_TEST テーブルを作成します。

```
CREATE TABLE linestring_test (
lid integer,
ln1 sde.st_geometry,
ln2 sde.st_geometry);
```

次の INSERT ステートメントは、2 つの ST_LineString 値を ln1 と ln2 に挿入します。これらは同じで、座標の順序も同じです。

```
INSERT INTO linestring_test VALUES (
1,
sde.st_linestring ('linestring (10.01 20.02, 21.50 12.10)', 0),
sde.st_linestring ('linestring (21.50 12.10, 10.01 20.02)', 0)
);
```

次の SELECT ステートメントとそれに対応する結果セットは、ST_Equals 関数が座標の順序に関係なく t (真) を返す様子を示しています。 ST_OrderingEquals 関数は、ジオメトリが等しく座標の順序が同じであるという条件を満たしていない場合は f (偽) を返します。

```
SELECT lid, sde.st_equals (ln1, ln2) AS Equals, sde.st_orderingequals (ln1, ln2)
AS OrderingEquals
FROM linestring_test;

lid equals     orderingequals

1   t          f
```

ST_Overlaps

定義

ST_Overlaps は、2 つのジオメトリ オブジェクトを入力として、オブジェクトのインターセクトが、同じディメンションのジオメトリ オブジェクトで、かつソース オブジェクトとは等しくない場合に 1 (Oracle および SQLite) または t (PostgreSQL) を返します。それ以外の場合は、0 (Oracle および SQLite) または f (PostgreSQL) を返します。

構文

Oracle および PostgreSQL

```
sde.st_overlaps (geometry1 sde.st_geometry, geometry2 sde.st_geometry)
```

SQLite

```
st_overlaps (geometry1 geometryblob, geometry2 geometryblob)
```

戻り値のタイプ

Boolean

例

国の行政官は、有害廃棄物区域のバッファー範囲に重なる保護区域のリストを必要としています。sensitive_areas テーブルには、脅威を受ける施設を記述する複数の列と、施設の ST_Polygon ジオメトリを格納する shape 列があります。

hazardous_sites テーブルは、id 列にサイトの ID、site ポイント列に各サイトの実際の地理的位置を格納します。

sensitive_areas および hazardous_sites テーブルは、ST_Overlaps 関数によって結合され、hazardous_sites のポイントのバッファー範囲に重なるポリゴンを持つすべての sensitive_areas の行の ID を返します。

Oracle

```
CREATE TABLE sensitive_areas (
 id integer,
 shape sde.st_geometry
);
CREATE TABLE hazardous_sites (
 id integer,
 site sde.st_geometry
);
INSERT INTO sensitive_areas VALUES (
 1,
 sde.st_geometry ('polygon ((.20 .30, .30 .30, .30 .40, .20 .40, .20 .30))', 4326)
);
INSERT INTO sensitive_areas VALUES (
 2,
 sde.st_geometry ('polygon ((.30 .30, .30 .50, .50 .50, .50 .30, .30 .30))', 4326)
);
INSERT INTO sensitive_areas VALUES (
 3,
 sde.st_geometry ('polygon ((.40 .40, .40 .60, .60 .60, .60 .40, .40 .40))', 4326)
```

```
);
INSERT INTO hazardous_sites VALUES (
 4,
 sde.st_geometry ('point (.60 .60)', 4326)
);
INSERT INTO hazardous_sites VALUES (
 5,
 sde.st_geometry ('point (.30 .30)', 4326)
);
```

```
SELECT UNIQUE (hs.id)
 FROM HAZARDOUS_SITES hs, SENSITIVE_AREAS sa
 WHERE sde.st_overlaps (sde.st_buffer (hs.site, .001), sa.shape) = 1;
ID
4
5
```

PostgreSQL

```
CREATE TABLE sensitive_areas (
 id serial,
 shape sde.st_geometry
);
CREATE TABLE hazardous_sites (
 id serial,
 site sde.st_geometry
);
INSERT INTO sensitive_areas (shape) VALUES (
 sde.st_geometry ('polygon ((.20 .30, .30 .30, .30 .40, .20 .40, .20 .30))', 4326)
);
INSERT INTO sensitive_areas (shape) VALUES (
 sde.st_geometry ('polygon ((.30 .30, .30 .50, .50 .50, .50 .30, .30 .30))', 4326)
);
INSERT INTO sensitive_areas (shape) VALUES (
 sde.st_geometry ('polygon ((.40 .40, .40 .60, .60 .60, .60 .40, .40 .40))', 4326)
);
INSERT INTO hazardous_sites (site) VALUES (
 sde.st_geometry ('point (.60 .60)', 4326)
);
INSERT INTO hazardous_sites (site) VALUES (
 sde.st_geometry ('point (.30 .30)', 4326)
);
```

```
SELECT DISTINCT (hs.id) AS "Hazardous Site ID"
 FROM hazardous_sites hs, sensitive_areas sa
 WHERE sde.st_overlaps (sde.st_buffer (hs.site, .001), sa.shape) = 't';
id
1
2
```

SQLite

```
CREATE TABLE sensitive_areas (
 id integer primary key autoincrement not null
);
```

```
SELECT AddGeometryColumn(
 NULL,
 'sensitive_areas',
 'shape',
 4326,
 'polygon',
 'xy',
 'null'
);
CREATE TABLE hazardous_sites (
 id integer primary key autoincrement not null,
 site_name varchar(30)
);
SELECT AddGeometryColumn(
 NULL,
 'hazardous_sites',
 'site',
 4326,
 'point',
 'xy',
 'null'
);
INSERT INTO sensitive_areas (shape) VALUES (
 st_geometry ('polygon ((.20 .30, .30 .30, .30 .40, .20 .40, .20 .30))', 4326)
);
INSERT INTO sensitive_areas (shape) VALUES (
 st_geometry ('polygon ((.30 .30, .30 .50, .50 .50, .50 .30, .30 .30))', 4326)
);
INSERT INTO sensitive_areas (shape) VALUES (
 st_geometry ('polygon ((.40 .40, .40 .60, .60 .60, .60 .40, .40 .40))', 4326)
);
INSERT INTO hazardous_sites (site_name, site) VALUES (
 'Kemlabs',
 st_geometry ('point (.60 .60)', 4326)
);
INSERT INTO hazardous_sites (site_name, site) VALUES (
 'Medi-Waste',
 st_geometry ('point (.30 .30)', 4326)
);
```

```
SELECT DISTINCT (hs.site_name) AS "Hazardous Site"
 FROM hazardous_sites hs, sensitive_areas sa
 WHERE st_overlaps (st_buffer (hs.site, .001), sa.shape) = 1;
Hazardous Site
Kemlabs
Medi-Waste
```

ST_Perimeter

定義

ST_Perimeter は、閉じたポリゴンまたはマルチポリゴン フィーチャの境界線を形成する連続したラインの長さを返します。

これは 10.8.1 の新しい関数です。

構文

各セクションの最初の 2 つのオプションは、フィーチャに定義された座標系の単位で周長を返します。2 つ目のオプションでは、計測距離単位を指定できます。linear_unit_name でサポートされる値のリストについては、「ST_Distance」をご参照ください。

Oracle と PostgreSQL

```
sde.st_perimeter (polygon sde.st_geometry)

sde.st_perimeter (multipolygon sde.st_geometry)

sde.st_perimeter (polygon sde.st_geometry, linear_unit_name text)

sde.st_perimeter (multipolygon sde.st_geometry, linear_unit_name text)
```

SQLite

```
st_perimeter (polygon sde.st_geometry)

st_perimeter (multipolygon sde.st_geometry)

st_perimeter (polygon sde.st_geometry, linear_unit_name text)

st_perimeter (multipolygon sde.st_geometry, linear_unit_name text)
```

戻り値のタイプ

Double precision

例

Oracle

次の例では、海岸線の野鳥を研究する生態学者が、特定エリアの湖の周長を求める必要があります。湖は、waterbodies テーブルにポリゴンとして格納されています。ST_Perimeter 関数を使用した SELECT ステートメントは、waterbodies テーブル内の各湖 (フィーチャ) の周長を返すために使用されます。

```
--Create table named waterbodies
CREATE TABLE waterbodies (wbid INTEGER not null, waterbody sde.st_geometry);
--Insert a polygon feature to the waterbodies table
INSERT INTO waterbodies VALUES (
  1,
  sde.ST_Polygon ('polygon ((0 0, 0 4, 5 4, 5 0, 0 0))', 1)
);
--Find the perimeter of the polygon
SELECT sde.ST_Perimeter (waterbody)
```

```
  FROM waterbodies;
```

SELECT ステートメントは、以下を返します。

```
ID PERIMETER
1 +1.8000000
```

次の例では、bfp というテーブルを作成し、3 つのフィーチャを挿入して、各フィーチャの周長を距離単位で計算します。

```
--Create table named bfp
CREATE TABLE bfp (
 building_id integer not null,
 footprint sde.st_geometry);
--Insert polygon features to the bfp table
INSERT INTO BFP (building_id, footprint) VALUES (
 1,
 sde.st_polygon ('polygon ((0 0, 0 10, 10 10, 10 0, 0 0))', 4326)
);
INSERT INTO BFP (building_id, footprint) VALUES (
 2,
 sde.st_polygon ('polygon ((20 0, 30 20, 40 0, 20 0))', 4326)
);
INSERT INTO BFP (building_id, footprint) VALUES (
 3,
 sde.st_polygon ('polygon ((20 30, 25 35, 30 30, 20 30))', 4326)
);
--Find the perimeter of each polygon
SELECT sde.ST_Perimeter(footprint)
            ,sde.ST_Perimeter(footprint, 'meter') as Meter
            ,sde.ST_Perimeter(footprint, 'km') as KM
            ,sde.ST_Perimeter(footprint, 'yard') As Yard
FROM bfp;
```

SELECT ステートメントは、各フィーチャの周長を 3 つの単位で返します。

```
st_perimeter    |       meter       |         km         |        yard
------------------+-------------------+--------------------+--------------------
 40.00000000000001 | 4421256.128972424 |  4421.256128972425 |   4835144.49800134
  64.7213595499958 | 7159231.951087892 | 7159.2319510878915 |  7829431.267593933
 24.14213562373095 | 2417672.365575198 |  2417.672365575198 | 2643998.6500166208
```

PostgreSQL

次の例では、海岸線の野鳥を研究する生態学者が、特定エリアの湖の周長を求める必要があります。湖は、waterbodies テーブルにポリゴンとして格納されています。ST_Perimeter 関数を使用した SELECT ステートメントは、waterbodies テーブル内の各湖 (フィーチャ) の周長を返すために使用されます。

```
--Create table named waterbodies
CREATE TABLE waterbodies (wbid INTEGER not null, waterbody sde.st_geometry);
--Insert a polygon feature to the waterbodies table
INSERT INTO waterbodies VALUES (
  1,
```

```
  sde.ST_Polygon ('polygon ((0 0, 0 4, 5 4, 5 0, 0 0))', 1)
);
--Find the perimeter of the polygon
SELECT sde.ST_Perimeter (waterbody)
  FROM waterbodies;
```

SELECT ステートメントは、以下を返します。

```
ID PERIMETER
1 +1.8000000
```

次の例では、bfp というテーブルを作成し、3 つのフィーチャを挿入して、各フィーチャの周長を距離単位で計算します。

```
--Create table named bfp
CREATE TABLE bfp (
 building_id serial,
 footprint sde.st_geometry);
--Insert polygon features to the bfp table
INSERT INTO bfp (footprint) VALUES (
 sde.st_polygon ('polygon ((0 0, 0 10, 10 10, 10 0, 0 0))', 4326)
);
INSERT INTO bfp (footprint) VALUES (
 sde.st_polygon ('polygon ((20 0, 30 20, 40 0, 20 0))', 4326)
);
INSERT INTO bfp (footprint) VALUES (
 sde.st_polygon ('polygon ((20 30, 25 35, 30 30, 20 30))', 4326)
);
--Find the perimeter of each polygon
SELECT sde.st_perimeter(footprint)
            ,sde.st_perimeter(footprint, 'meter') as Meter
            ,sde.st_perimeter(footprint, 'km') as KM
            ,sde.st_perimeter(footprint, 'yard') As Yard
FROM bfp;
```

SELECT ステートメントは、各フィーチャの周長を 3 つの単位で返します。

```
 st_perimeter    |       meter       |         km         |        yard
-------------------+-------------------+--------------------+--------------------
 40.00000000000001 | 4421256.128972424 |  4421.256128972425 |   4835144.49800134
  64.7213595499958 | 7159231.951087892 | 7159.2319510878915 |  7829431.267593933
 24.14213562373095 | 2417672.365575198 |  2417.672365575198 | 2643998.6500166208
```

SQLite

次の例では、海岸線の野鳥を研究する生態学者が、特定エリアの湖の周長を求める必要があります。湖は、waterbodies テーブルにポリゴンとして格納されています。ST_Perimeter 関数を使用した SELECT ステートメントは、waterbodies テーブル内の各湖 (フィーチャ) の周長を返すために使用されます。

```
--Create table named waterbodies and add a spatial column (waterbody) to it
CREATE TABLE waterbodies (wbid integer primary key autoincrement not null
);
SELECT AddGeometryColumn(
```

```
 NULL,
 'waterbodies',
 'waterbody',
 4326,
 'polygon',
 'xy',
 'null'
);
--Insert a polygon feature to the waterbodies table
INSERT INTO waterbodies VALUES (
  1,
  ST_Polygon ('polygon ((0 0, 0 4, 5 4, 5 0, 0 0))', 1)
);
--Find the perimeter of the polygon
SELECT ST_Perimeter (waterbody)
  FROM waterbodies;
```

SELECT ステートメントは、以下を返します。

```
ID PERIMETER
1 +1.8000000
```

次の例では、bfp というテーブルを作成し、3 つのフィーチャを挿入して、各フィーチャの周長を距離単位で計算します。

```
--Create table named bfp and add a spatial column (footprints) to it
CREATE TABLE bfp (
 building_id integer primary key autoincrement not null
);
SELECT AddGeometryColumn(
 NULL,
 'bfp',
 'footprint',
 4326,
 'polygon',
 'xy',
 'null'
);
--Insert polygon features to the bfp table
INSERT INTO bfp (footprint) VALUES (
 st_polygon ('polygon ((0 0, 0 10, 10 10, 10 0, 0 0))', 4326)
);
INSERT INTO bfp (footprint) VALUES (
 st_polygon ('polygon ((20 0, 30 20, 40 0, 20 0))', 4326)
);
INSERT INTO bfp (footprint) VALUES (
 st_polygon ('polygon ((20 30, 25 35, 30 30, 20 30))', 4326)
);
--Find the perimeter of each polygon
SELECT ST_Perimeter(footprint)
            ,ST_Perimeter(footprint, 'meter') as Meter
            ,ST_Perimeter(footprint, 'km') as KM
            ,ST_Perimeter(footprint, 'yard') As Yard
FROM bfp;
```

SELECT ステートメントは、各フィーチャの周長を 3 つの単位で返します。

```
 st_perimeter    |       meter       |         km          |        yard
-------------------+-------------------+---------------------+--------------------
 40.00000000000001 | 4421256.128972424 |  4421.256128972425 |   4835144.49800134
  64.7213595499958 | 7159231.951087892 | 7159.2319510878915 |  7829431.267593933
 24.14213562373095 | 2417672.365575198 |  2417.672365575198 | 2643998.6500166208
```

ST_Point

定義

ST_Point は、WKT オブジェクトまたは座標と空間参照 ID を受け取って、ST_Point を返します。

注意:
ArcGIS で使用する空間テーブルを作成する場合は、ST_Geometry サブタイプを指定するよりも、ジオメトリ スーパータイプ (たとえば、ST_Geometry) として列を作成することをお勧めします。

構文

Oracle

```
sde.st_point (wkt clob, srid integer)
sde.st_point (x number, y number, srid integer)
sde.st_point (x number, y number, m number, srid integer)
sde.st_point (x number, y number, z number, srid integer)
sde.st_point (x number, y number, z number, m number, srid integer)
```

PostgreSQL

```
sde.st_point (wkt clob, srid integer)
sde.st_point (esri_shape bytea, srid integer)sde.
sde.st_point (x double precision, y double precision, srid integer)
sde.st_point (x double precision, y double precision, m double precision, srid integer)
sde.st_point (x double precision, y double precision, z double precision, srid integer)
sde.st_point (x double precision, y double precision, z double precision, m double
precision, srid integer)
```

SQLite

```
st_point (wkt text, srid int32)
st_point (x float64, y float64, srid int32)
st_point (x float64, y float64, z float64, m float64, srid int32)
```

戻り値のタイプ

ST_Point

例

次の CREATE TABLE ステートメントは、PT1 ポイント列を 1 つ持つ point_test テーブルを作成します。

ST_Point 関数によってポイント座標を ST_Point ジオメトリに変換してから、pt1 列に挿入します。

Oracle

```
CREATE TABLE point_test (pt1 sde.st_geometry);
```

```
INSERT INTO point_test VALUES (
 sde.st_point (10.01, 20.03, 4326)
);
```

PostgreSQL

```
CREATE TABLE point_test (pt1 sde.st_geometry);
```

```
INSERT INTO point_test VALUES (
 sde.st_point (10.01, 20.03, 4326)
);
```

SQLite

```
CREATE TABLE point_test (id integer);

SELECT AddGeometryColumn(
 NULL,
 'point_test',
 'pt1',
 4326,
 'point',
 'xy',
 'null'
);
```

```
INSERT INTO point_test VALUES (
 1,
 st_point (10.01, 20.03, 4326)
);
```

ST_PointFromText

注意:
Oracle および SQLite でのみ使用されます。PostgreSQL では、ST_Point を使用してください。

定義

ST_PointFromText は、ポイント タイプの WKT 表現と空間参照 ID を受け取って、ポイントを返します。

構文

Oracle

```
sde.st_pointfromtext (wkt varchar2, srid integer)
```

```
sde.st_pointfromtext (wkt varchar2)
```

SRID を指定しない場合、空間参照がデフォルトで 4326 に設定されます。

SQLite

```
st_pointfromtext (wkt text, srid int32)
```

```
st_pointfromtext (wkt text)
```

SRID を指定しない場合、空間参照がデフォルトで 4326 に設定されます。

戻り値のタイプ

ST_Point

例

ST_Point 列の pt1 を 1 つ持つ point_test テーブルを作成します。

ST_PointFromText 関数は、INSERT ステートメントがポイントを pt1 列に挿入する前に、ポイント テキスト座標をポイント形式に変換します。

Oracle

```
CREATE TABLE point_test (pt1 sde.st_geometry);
```

```
INSERT INTO POINT_TEST VALUES (
 sde.st_pointfromtext ('point (10.01 20.03)', 4326)
);
```

SQLite

```
CREATE TABLE pt_test (id integer);
SELECT AddGeometryColumn(
 NULL,
 'pt_test',
 'pt1',
 4326,
 'point',
 'xy',
 'null'
);
```

```
INSERT INTO pt_test VALUES (
 1,
 st_pointfromtext ('point (10.01 20.03)', 4326)
);
```

ST_PointFromWKB

定義

ST_PointFromWKB は、WKB 表現と空間参照 ID を受け取って、ST_Point を返します。

構文

Oracle

```
sde.st_pointfromwkb (wkb blob, srid integer)
```

```
sde.st_pointfromwkb (wkb blob)
```

SRID を指定しない場合、空間参照がデフォルトで 4326 に設定されます。

PostgreSQL

```
sde.st_pointfromwkb (wkb bytea, srid integer)
```

SQLite

```
st_pointfromwkb (wkb blob, srid int32)
```

```
st_pointfromwkb (wkb blob)
```

SRID を指定しない場合、空間参照がデフォルトで 4326 に設定されます。

戻り値のタイプ

ST_Point

例

この例では、ST_PointFromWKB を使用して、その WKB 表現からポイントを作成する方法を示しています。ジオメトリは、空間参照系 4326 のポイントです。この例では、ポイントを sample_points テーブルの geometry 列に格納した後、wkb 列を (ST_AsBinary 関数を使用して) WKB 表現で更新しています。最後に、ST_PointFromWKB 関数を使用して、WKB 列からポイントを返します。sample-points テーブルには、ポイントを格納する geometry 列と、ポイントの WKB 表現を格納する wkb 列があります。

SELECT ステートメントで、ST_PointFromWKB 関数を使用して WKB 列からポイントを取得します。

Oracle

```
CREATE TABLE sample_points (
 id integer,
 geometry sde.st_point,
```

```
 wkb blob
);
INSERT INTO SAMPLE_POINTS (id, geometry) VALUES (
 10,
 sde.st_point ('point (44 14)', 4326)
);
INSERT INTO SAMPLE_POINTS (id, geometry) VALUES (
 11,
 sde.st_point ('point (24 13)', 4326)
);
UPDATE SAMPLE_POINTS
 SET wkb = sde.st_asbinary (geometry)
 WHERE id = 10;
UPDATE SAMPLE_POINTS
 SET wkb = sde.st_asbinary (geometry)
 WHERE id = 11;
```

```
SELECT id, sde.st_astext (sde.st_pointfromwkb(wkb, 4326)) POINTS
 FROM SAMPLE_POINTS;
ID POINTS
10 POINT (44.00000000 14.00000000)
11 POINT (24.00000000 13.00000000)
```

PostgreSQL

```
CREATE TABLE sample_points (
 id integer,
 geometry sde.st_point,
 wkb bytea
);
INSERT INTO sample_points (id, geometry) VALUES (
 10,
 sde.st_point ('point (44 14)', 4326)
);
INSERT INTO sample_points (id, geometry) VALUES (
 11,
 sde.st_point ('point (24 13)', 4326)
);
UPDATE sample_points
 SET wkb = sde.st_asbinary (geometry)
 WHERE id = 10;
UPDATE sample_points
 SET wkb = sde.st_asbinary (geometry)
 WHERE id = 11;
```

```
SELECT id, sde.st_astext (sde.st_pointfromwkb(wkb, 4326))
 AS points
 FROM sample_points;
id points
10 POINT (44 14)
11 POINT (24 13)
```

SQLite

```
CREATE TABLE sample_pts (
```

```
 id integer,
 wkb blob
);
SELECT AddGeometryColumn(
 NULL,
 'sample_pts',
 'geometry',
 4326,
 'point',
 'xy',
 'null'
);
INSERT INTO sample_pts (id, geometry) VALUES (
 10,
 st_point ('point (44 14)', 4326)
);
INSERT INTO sample_pts (id, geometry) VALUES (
 11,
 st_point ('point (24 13)', 4326)
);
UPDATE sample_pts
 SET wkb = st_asbinary (geometry)
 WHERE id = 10;
UPDATE sample_pts
 SET wkb = st_asbinary (geometry)
 WHERE id = 11;
```

```
SELECT id, st_astext (st_pointfromwkb(wkb, 4326))
 AS "points"
 FROM sample_pts;
id points
10 POINT (44.00000000 14.00000000)
11 POINT (24.00000000 13.00000000)
```

ST_PointN

定義

ST_PointN は、ST_LineString と整数インデックスを入力として、ST_LineString のパスにある、n 番目の頂点のポイントを返します。

構文

Oracle および PostgreSQL

```
sde.st_pointn (line1 sde.st_linestring, index integer)
```

SQLite

```
st_pointn (line1 st_linestring, index int32)
```

戻り値のタイプ

ST_Point

例

各行を一意に識別する gid 列と、ln1 ST_LineString 列を持つ pointn_test テーブルを作成します。INSERT ステートメントは、2 つのラインストリング値を挿入します。最初のラインストリングには Z 座標またはメジャー値がありませんが、2 番目のラインストリングには両方あります。

SELECT クエリは ST_PointN および ST_AsText 関数を使用して、各ラインストリングの 2 番目の頂点の WKT (Well-Known Text) を返します。

Oracle

```
CREATE TABLE pointn_test (
 gid integer,
 ln1 sde.st_geometry
);
INSERT INTO POINTN_TEST VALUES (
 1,
 sde.st_linefromtext ('linestring (10.02 20.01, 23.73 21.92, 30.10 40.23)', 4326)
);
INSERT INTO POINTN_TEST VALUES (
 2,
 sde.st_linefromtext ('linestring zm(10.02 20.01 5.0 7.0, 23.73 21.92 6.5 7.1, 30.10
40.23 6.9 7.2)', 4326)
);
```

```
SELECT gid, sde.st_astext (sde.st_pointn (ln1, 2)) The_2ndvertex
 FROM POINTN_TEST;
GID The_2ndvertex
1   POINT (23.73 21.92)
2   POINT ZM (23.73 21.92 6.5 7.1)
```

PostgreSQL

```
CREATE TABLE pointn_test (
 gid serial,
 ln1 sde.st_geometry
);
INSERT INTO pointn_test (ln1) VALUES (
 sde.st_linestring ('linestring (10.02 20.01, 23.73 21.92, 30.10 40.23)', 4326)
);
INSERT INTO pointn_test (ln1) VALUES (
 sde.st_linestring ('linestring zm(10.02 20.01 5.0 7.0, 23.73 21.92 6.5 7.1, 30.10
40.23 6.9 7.2)', 4326)
);
```

```
SELECT gid, sde.st_astext (sde.st_pointn (ln1, 2))
 AS The_2ndvertex
 FROM pointn_test;
gid the_2ndvertex
1   POINT (23.73 21.92)
2   POINT ZM (23.73 21.92 6.5 7.1)
```

SQLite

```
CREATE TABLE pointn_test (
 gid integer primary key autoincrement not null
);
SELECT AddGeometryColumn(
 NULL,
 'pointn_test',
 'ln1',
 4326,
 'linestringzm',
 'xyzm',
 'null'
);
INSERT INTO pointn_test (ln1) VALUES (
 st_linestring ('linestring (10.02 20.01, 23.73 21.92, 30.10 40.23)', 4326)
);
INSERT INTO pointn_test (ln1) VALUES (
 st_linestring ('linestring zm(10.02 20.01 5.0 7.0, 23.73 21.92 6.5 7.1, 30.10 40.23
6.9 7.2)', 4326)
);
```

```
SELECT gid, st_astext (st_pointn (ln1, 2))
 AS "Second Vertex"
 FROM pointn_test;
gid  Second Vertex
1   POINT ( 23.73000000 21.92000000)
2   POINT ZM ( 23.73000000 21.92000000 6.50000000 7.10000000)
```

ST_PointOnSurface

説明

ST_PointOnSurface は、ST_Polygon または ST_MultiPolygon を入力として、サーフェス上にあることが保証される ST_Point を返します。

構文

Oracle および PostgreSQL

```
sde.st_pointonsurface (polygon1 sde.st_geometry)
sde.st_pointonsurface (multipolygon1 sde.st_geometry)
```

SQLite

```
st_pointonsurface (polygon1 geometryblob)
st_pointonsurface (multipolygon1 geometryblob)
```

戻り値のタイプ

ST_Point

例

都市エンジニアは、歴史的な建物のそれぞれにラベル ポイントを作成したいと考えています。歴史的な建物は、次の CREATE TABLE ステートメントで作成された hbuildings テーブルに格納されます。

ST_PointOnSurface 関数は、建物のサーフェス上にあることが保証されるポイントを生成します。
ST_PointOnSurface 関数は、ST_AsText 関数がアプリケーションで使用できるテキストに変換するポイントを返します。

Oracle

```
CREATE TABLE hbuildings (
 hbld_id integer,
 hbld_name varchar(40),
 footprint sde.st_geometry
);
```

```
INSERT INTO hbuildings (hbld_id, hbld_name, footprint) VALUES (
 1,
 'First National Bank',
 sde.st_polygon ('polygon ((0 0, 0 .010, .010 .010, .010 0, 0 0))', 4326)
);

INSERT INTO hbuildings (hbld_id, hbld_name, footprint) VALUES (
 2,
 'Courthouse',
 sde.st_polygon ('polygon ((.020 0, .020 .010, .030 .010, .030 0, .020 0))', 4326)
);
```

```
SELECT sde.st_astext (sde.st_pointonsurface (footprint)) Historic_Site
 FROM HBUILDINGS;

HISTORIC_SITE

POINT  (0.00500000 0.00500000)
POINT  (0.02500000 0.00500000)
```

PostgreSQL

```
CREATE TABLE hbuildings (
 hbld_id serial,
 hbld_name varchar(40),
 footprint sde.st_geometry
);
```

```
INSERT INTO hbuildings (hbld_name, footprint) VALUES (
 'First National Bank',
 sde.st_polygon ('polygon ((0 0, 0 .010, .010 .010, .010 0, 0 0))', 4326)
);

INSERT INTO hbuildings (hbld_name, footprint) VALUES (
 'Courthouse',
 sde.st_polygon ('polygon ((.020 0, .020 .010, .030 .010, .030 0, .020 0))', 4326)
);
```

```
SELECT sde.st_astext (sde.st_pointonsurface (footprint))
 AS "Historic Site"
 FROM hbuildings;

Historic Site
```

```
POINT  (0.00500000 0.00500000)
POINT  (0.02500000 0.00500000)
```

SQLite

```
CREATE TABLE hbuildings (
 hbld_id integer primary key autoincrement not null,
 hbld_name text(40)
);

SELECT AddGeometryColumn(
 NULL,
 'hbuildings',
 'footprint',
 4326,
 'polygon',
 'xy',
 'null'
);
```

```
INSERT INTO hbuildings (hbld_name, footprint) VALUES (
 'First National Bank',
 st_polygon ('polygon ((0 0, 0 .010, .010 .010, .010 0, 0 0))', 4326)
);

INSERT INTO hbuildings (hbld_name, footprint) VALUES (
 'Courthouse',
 st_polygon ('polygon ((.020 0, .020 .010, .030 .010, .030 0, .020 0))', 4326)
);
```

```
SELECT st_astext (st_pointonsurface (footprint))
 AS "Historic Site"
 FROM hbuildings;

Historic Site

POINT  (0.00500000 0.00500000)
POINT  (0.02500000 0.00500000)
```

ST_PolyFromText

注意:
Oracle と SQLite のみ

定義

ST_PolyFromText は、WKT 表現と空間参照 ID を受け取って、ST_Polygon を返します。

構文

Oracle

```
sde.st_polyfromtext (wkt clob, srid integer)
```

```
sde.st_polyfromtext (wkt clob)
```

SRID を指定しない場合、空間参照がデフォルトで 4326 に設定されます。

SQLite

```
st_polyfromtext (wkt text, srid int32)
```

```
st_polyfromtext (wkt text)
```

SRID を指定しない場合、空間参照がデフォルトで 4326 に設定されます。

戻り値のタイプ

ST_Polygon

例

ポリゴン列を 1 つ持つ polygon_test テーブルを作成します。

INSERT ステートメントは、ST_PolyFromText 関数を使用して、ポリゴンをポリゴン列に挿入します。

Oracle

```
CREATE TABLE polygon_test (pl1 sde.st_geometry);
```

```
INSERT INTO polygon_test VALUES (
 sde.st_polyfromtext ('polygon ((10.01 20.03, 10.52 40.11, 30.29 41.56, 31.78 10.74,
10.01 20.03))', 4326)
);
```

SQLite

```
CREATE TABLE polygon_test (id integer);
SELECT AddGeometryColumn(
 NULL,
 'polygon_test',
 'pl1',
 4326,
 'polygon',
 'xy',
 'null'
);
```

```
INSERT INTO polygon_test VALUES (
 1,
 st_polyfromtext ('polygon ((10.01 20.03, 10.52 40.11, 30.29 41.56, 31.78 10.74, 10.01
20.03))', 4326)
);
```

ST_PolyFromWKB

定義

ST_PolyFromWKB は、WKB 表現と空間参照 ID を受け取って、ST_Polygon を返します。

構文

Oracle

```
sde.st_polyfromwkb (wkb blob, srid integer)
```

```
sde.st_polyfromwkb (wkb blob)
```

SRID を指定しない場合、空間参照がデフォルトで 4326 に設定されます。

PostgreSQL

```
sde.st_polyfromwkb (wkb bytea, srid integer)
```

SQLite

```
st_polyfromwkb (wkb blob, srid int32)
```

```
st_polyfromwkb (wkb blob)
```

SRID を指定しない場合、空間参照がデフォルトで 4326 に設定されます。

戻り値のタイプ

ST_Polygon

例

この例では、ST_PolyFromWKB を使用して、その WKB 表現からポリゴンを作成する方法を示しています。ジオメトリは、空間参照系 4326 のポリゴンです。この例では、ポリゴンを sample_polys テーブルの geometry 列に ID = 1115 で格納した後、wkb 列を (ST_AsBinary 関数を使用して) WKB 表現で更新しています。最後に、ST_PolyFromWKB 関数を使用して、WKB 列からマルチポリゴンを返します。sample_polys テーブルには、ポリゴンを格納する geometry 列と、ポリゴンの WKB 表現を格納する wkb 列があります。

SELECT ステートメントで、ST_PointFromWKB 関数を使用して WKB 列からポイントを取得します。

Oracle

```
CREATE TABLE sample_polys (
 id integer,
 geometry sde.st_geometry,
```

```
 wkb blob
);
INSERT INTO SAMPLE_POLYS (id, geometry) VALUES (
 1115,
 sde.st_polyfromtext ('polygon ((10.01 20.03, 10.52 40.11, 30.29 41.56, 31.78 10.74,
10.01 20.03))', 4326)
);
UPDATE SAMPLE_POLYS
 SET wkb = sde.st_asbinary (geometry)
 WHERE id = 1115;
```

```
SELECT id, sde.st_astext (sde.st_polyfromwkb (wkb, 4326)) POLYS
 FROM SAMPLE_POLYS;
ID            POLYS
1115    POLYGON (10.01000000 20.03000000, 31.78000000 10.74000000, 30.29000000
41.56000000, 10.52000000 40.11000000, 10.01000000 20.03000000)
```

PostgreSQL

```
CREATE TABLE sample_polys (
 id integer,
 geometry sde.st_geometry,
 wkb bytea
);
INSERT INTO sample_polys (id, geometry) VALUES (
 1115,
 sde.st_polygon ('polygon  ((10.01 20.03, 10.52 40.11, 30.29 41.56, 31.78 10.74, 10.01
20.03))', 4326)
);
UPDATE sample_polys
 SET wkb = sde.st_asbinary (geometry)
 WHERE id = 1115;
```

```
SELECT id, sde.st_astext (sde.st_polyfromwkb (wkb, 4326))
 AS POLYS
 FROM sample_polys;
id    polys
1115    POLYGON (10.01000000 20.03000000, 31.78000000 10.74000000, 30.29000000
41.56000000, 10.52000000 40.11000000, 10.01000000 20.03000000)
```

SQLite

```
CREATE TABLE sample_polys(
 id integer,
 wkb blob
);
SELECT AddGeometryColumn(
 NULL,
 'sample_polys',
 'geometry',
 4326,
 'polygon',
 'xy',
 'null'
);
```

```
INSERT INTO sample_polys (id, geometry) VALUES (
 1115,
 st_polyfromtext ('polygon ((10.01 20.03, 10.52 40.11, 30.29 41.56, 31.78 10.74, 10.01
20.03))', 4326)
);
UPDATE sample_polys
 SET wkb = st_asbinary (geometry)
 WHERE id = 1115;
```

```
SELECT id, st_astext (st_polyfromwkb (wkb, 4326))
 AS "polygons"
 FROM sample_polys;
id    polygons
1115    POLYGON (10.01000000 20.03000000, 31.78000000 10.74000000, 30.29000000
41.56000000, 10.52000000 40.11000000, 10.01000000 20.03000000)
```

ST_Polygon

定義

ST_Polygon アクセサ関数は、WKT 表現と空間参照 ID (SRID) を受け取って、ST_Polygon を生成します。

注意:
ArcGIS で使用する空間テーブルを作成する場合は、ST_Geometry サブタイプを指定するよりも、ジオメトリ スーパータイプ (たとえば、ST_Geometry) として列を作成することをお勧めします。

構文

Oracle

```
sde.st_polygon (wkt clob, srid integer)
```

PostgreSQL

```
sde.st_polygon (wkt clob, srid integer)
sde.st_polygon (esri_shape bytea, srid integer)
```

SQLite

```
st_polygon (wkt text, srid int32)
```

戻り値のタイプ

ST_Polygon

例

次の CREATE TABLE ステートメントは、p1 列を 1 つ持つ polygon_test テーブルを作成します。 次の INSERT ステートメントは、ST_Polygon 関数を使用して、リング (閉じていてシンプルなポリゴン) を ST_Polygon に変換して p1 列に挿入します。

Oracle

```
CREATE TABLE polygon_test (p1 sde.st_geometry);

INSERT INTO polygon_test VALUES (
 sde.st_polygon ('polygon ((10.01 20.03, 20.94 21.34, 35.93 10.04, 10.01 20.03))', 4326)
);
```

PostgreSQL

```
CREATE TABLE polygon_test (p1 sde.st_geometry);

INSERT INTO polygon_test VALUES (
```

```
 sde.st_polygon ('polygon ((10.01 20.03, 20.94 21.34, 35.93 10.04, 10.01 20.03))', 4326)
);
```

SQLite

```
CREATE TABLE poly_test (id integerp1 geometryblob);

SELECT AddGeometryColumn(
 NULL,
 'poly_test',
 'p1',
 4326,
 'polygon',
 'xy',
 'null'
);

INSERT INTO poly_test VALUES (
 1,
 st_polygon ('polygon ((10.01 20.03, 20.94 21.34, 35.93 10.04, 10.01 20.03))', 4326)
);
```

ST_Relate

定義

ST_Relate は、2 つのジオメトリを比較し、ジオメトリが DE-9IM のパターン マトリックス文字列で指定された条件を満たす場合は 1 (Oracle および SQLite) または t (PostgreSQL) を返します。それ以外の場合は 0 (Oracle および SQLite) または f (PostgreSQL) を返します。

SQLite および Oracle で ST_Relate を使用する場合、2 つ目のオプションがあります。そのオプションでは、2 つのジオメトリを比較し、ジオメトリ間のリレーションシップを定義する DE-9IM のパターン マトリックスを表す文字列を返すことができます。

構文

Oracle

オプション 1

```
sde.st_relate (geometry1 sde.st_geometry, geometry2 sde.st_geometry, patternMatrix
string)
```

オプション 2

```
sde.st_relate (geometry1 sde.st_geometry, geometry2 sde.st_geometry)
```

PostgreSQL

```
sde.st_relate (geometry1 sde.st_geometry, geometry2 sde.st_geometry, patternMatrix
string)
```

SQLite

オプション 1

```
st_relate (geometry1 geometryblob, geometry2 geometryblob, patternMatrix string)
```

オプション 2

```
st_relate (geometry1 geometryblob, geometry2 geometryblob)
```

戻り値のタイプ

PostgreSQL ではブール値が返されます。

SQLite と Oracle 用のオプション 1 は整数を返します。

SQLite と Oracle 用のオプション 2 は文字列を返します。

例

DE-9IM のパターン マトリックスは、ジオメトリを比較するための手段です。 このようなマトリックスにはさまざまなタイプがあります。 たとえば、ST_Relate 関数とイコール パターン マトリックス (T*F**FFF*) を使用して、2 つのジオメトリが等しいかどうかを判定できますが、DE-9IM のパターン (1*F**FFF*) を指定することもできます。 後者のパターンでは、ST_Relate は 1 つ目の位置で 2 つのジオメトリが等しいかどうか、つまり両方のジオメトリの内部の交差がライン (1 次元) であるかどうかを判定します。

以下の例では、3 つの空間列を持つ relate_test テーブルを作成し、各列にポイント フィーチャを挿入します。 SELECT ステートメントの中で、ポイントが等しいかどうかをテストするために ST_Relate 関数が使用されます。

ジオメトリが等しいかどうかを判定し、リレーションシップの次元性を調べる必要がない場合は、代わりに ST_Equals 関数を使用してください。

Oracle

最初の例では、ST_Relate の 1 つ目のオプションを示します。この例では、DE-9IM のパターン マトリックスに基づいてジオメトリを比較し、ジオメトリがマトリックスで定義された要件を満たしている場合に 1 を返し、満たしていない場合に 0 を返します。

```
CREATE TABLE relate_test (
 id NUMBER GENERATED ALWAYS AS IDENTITY minvalue 0,
 g1 sde.st_geometry
);

CREATE TABLE relate_test2 (
 id NUMBER GENERATED ALWAYS AS IDENTITY minvalue 0,
 g2 sde.st_geometry
);

CREATE TABLE relate_test3 (
 id NUMBER GENERATED ALWAYS AS IDENTITY minvalue 0,
 g3 sde.st_geometry
);
```

```
INSERT INTO relate_test (g1) VALUES (sde.st_geometry ('point (10.02 20.01)', 4326));

INSERT INTO relate_test2 (g2) VALUES (sde.st_geometry ('point (10.02 20.01)', 4326));

INSERT INTO relate_test3 (g3) VALUES (sde.st_geometry ('point (30.01 20.01)', 4326));
```

```
SELECT sde.st_relate (relate_test.g1, relate_test2.g2, 'T*F**FFF*') AS "g1=g2",
       sde.st_relate (relate_test.g1, relate_test3.g3, 'T*F**FFF*') AS "g1=g3",
       sde.st_relate (relate_test2.g2, relate_test3.g3, 'T*F**FFF*') AS "g2=g3"
FROM relate_test, relate_test2, relate_test3;
```

次の値を返します。

```
g1=g2     g1=g3      g2=g3

1                0                0
```

この例では、2 つ目のオプションを示します。 この例では、2 つのジオメトリを比較し、DE-9IM のパターン マトリックスを返します。

```
SELECT sde.st_relate (relate_test.g1,relate_test2.g2) AS "g1 rel g2"
 FROM relate_test, relate_test2;
```

次の値を返します。

```
g1 rel g2
0FFFFFFF2
```

PostgreSQL

この例では、DE-9IM のパターン マトリックスに基づいてジオメトリを比較し、ジオメトリがマトリックスで定義された要件を満たしている場合に t を返し、満たしていない場合に f を返します。

```
CREATE TABLE relate_test (
 id SERIAL,
 g1 sde.st_geometry
);

CREATE TABLE relate_test2 (
 id SERIAL,
 g2 sde.st_geometry
);

CREATE TABLE relate_test3 (
 id SERIAL,
 g3 sde.st_geometry
);
```

```
INSERT INTO relate_test(g1) VALUES (sde.st_geometry ('point (10.02 20.01)', 4326));

INSERT INTO relate_test2 (g2) VALUES (sde.st_geometry ('point (10.02 20.01)', 4326));

INSERT INTO relate_test3 (g3) VALUES (sde.st_geometry ('point (30.01 20.01)', 4326));
```

```
SELECT sde.st_relate (relate_test.g1, relate_test2.g2, 'T*F**FFF*') AS "g1=g2",
       sde.st_relate (relate_test.g1, relate_test3.g3, 'T*F**FFF*') AS "g1=g3",
       sde.st_relate (relate_test2.g2, relate_test3.g3, 'T*F**FFF*') AS "g2=g3"
FROM relate_test, relate_test2, relate_test3;
```

次の値を返します。

```
g1=g2     g1=g3     g2=g3

t          f         f
```

SQLite

最初の例では、ST_Relate の 1 つ目のオプションを示します。この例では、DE-9IM のパターン マトリックスに基

づいてジオメトリを比較し、ジオメトリがマトリックスで定義された要件を満たしている場合に 1 を返し、満たしていない場合に 0 を返します。

```
CREATE TABLE relate_test (id integer primary key autoincrement not null);

SELECT AddGeometryColumn(
 NULL,
 'relate_test',
 'g1',
 4326,
 'point',
 'xy',
 'null'
);

CREATE TABLE relate_test2 (id integer primary key autoincrement not null);

SELECT AddGeometryColumn(
 NULL,
 'relate_test2',
 'g2',
 4326,
 'point',
 'xy',
 'null'
);

CREATE TABLE relate_test3 (id integer primary key autoincrement not null);

SELECT AddGeometryColumn(
 NULL,
 'relate_test3',
 'g3',
 4326,
 'point',
 'xy',
 'null'
);
```

```
INSERT INTO relate_test (g1) VALUES (
 st_geometry ('point (10.02 20.01)', 4326)
);

INSERT INTO relate_test2 (g2) VALUES (
 st_geometry ('point (10.02 20.01)', 4326)
);

INSERT INTO relate_test3 (g3) VALUES (
 st_geometry ('point (30.01 20.01)', 4326)
);
```

```
SELECT st_relate (relate_test.g1, relate_test2.g2, 'T*F**FFF*') AS "g1=g2",
       st_relate (relate_test.g1, relate_test3.g3, 'T*F**FFF*') AS "g1=g3",
       st_relate (relate_test2.g2, relate_test3.g3, 'T*F**FFF*') AS "g2=g3"
 FROM relate_test, relate_test2, relate_test3;
```

次の値を返します。

```
g1=g2     g1=g3     g2=g3

1          0         0
```

この例では、2 つ目のオプションを示します。 この例では、2 つのジオメトリを比較し、DE-9IM のパターン マトリックスを返します。

```
SELECT st_relate (relate_test.g1,relate_test2.g2) AS "g1 rel g2"
 FROM relate_test, relate_test2;
```

次の値を返します。

```
g1 rel g2
0FFFFFFF2
```

ST_SRID

定義

ST_SRID は、ジオメトリ オブジェクトを入力として、その空間参照 ID を返します。

構文

Oracle および PostgreSQL

```
sde.st_srid (geometry1 sde.st_geometry)
```

SQLite

```
st_srid (geometry1 geometryblob)
```

戻り値のタイプ

Integer

例

次のテーブルを作成します。

次のステートメントでは、座標 (10.01, 50.76) にあるポイント ジオメトリがジオメトリ列の g1 に挿入されます。ポイント ジオメトリが作成されるときに、SRID の値に 4326 が割り当てられます。

ST_SRID 関数は、入力したジオメトリの空間参照 ID を返します。

Oracle

```
CREATE TABLE srid_test (g1 sde.st_geometry);
```

```
INSERT INTO SRID_TEST VALUES (
 sde.st_geometry ('point (10.01 50.76)', 4326)
);
```

```
SELECT sde.st_srid (g1) SRID_G1
 FROM SRID_TEST;
SRID_G1
4326
```

PostgreSQL

```
CREATE TABLE srid_test (g1 sde.st_geometry);
```

```
INSERT INTO srid_test VALUES (
```

```
 sde.st_point ('point (10.01 50.76)', 4326)
);
```

```
SELECT sde.st_srid (g1)
 AS SRID_G1
 FROM srid_test;
srid_g1
4326
```

SQLite

```
CREATE TABLE srid_test (id integer);
SELECT AddGeometryColumn(
 NULL,
 'srid_test',
 'g1',
 4326,
 'point',
 'xy',
 'null'
);
```

```
INSERT INTO srid_test VALUES (
 1,
 st_point ('point (10.01 50.76)', 4326)
);
```

```
SELECT st_srid (g1)
 AS "SRID"
 FROM srid_test;
SRID
4326
```

ST_StartPoint

定義

ST_StartPoint は、ラインストリングの最初のポイントを返します。

構文

Oracle および PostgreSQL

```
sde.st_startpoint (ln1 sde.st_geometry)
```

SQLite

```
st_startpoint (ln1 geometryblob)
```

戻り値のタイプ

ST_Point

例

テーブルの行を一意に識別する gid 列と、ln1 ST_LineString 列を持つ startpoint_test テーブルを作成します。

INSERT ステートメントは、ST_LineString を ln1 列に挿入します。最初の ST_LineString には Z 座標またはメジャー値がありませんが、2 番目の ST_LineString には両方あります。

ST_StartPoint 関数は、各 ST_LineString の最初のポイントを抽出します。ソースのラインストリングと同様に、リスト内の最初のポイントには Z 座標またはメジャー値がなく、2 番目のポイントには両方あります。

Oracle

```
CREATE TABLE startpoint_test (
 gid integer,
 ln1 sde.st_geometry
);
```

```
INSERT INTO STARTPOINT_TEST VALUES (
 1,
 sde.st_linefromtext ('linestring (10.02 20.01, 23.73 21.92, 30.10 40.23)', 4326)
);
INSERT INTO STARTPOINT_TEST VALUES (
 2,
 sde.st_linefromtext ('linestring zm(10.02 20.01 5 7, 23.73 21.92 6.5 7.1, 30.10 40.23
6.9 7.2)', 4326)
);
```

```
SELECT gid, sde.st_astext (sde.st_startpoint (ln1)) Startpoint
 FROM STARTPOINT_TEST;
GID  Startpoint
1    POINT (10.02000000 20.01000000)
```

```
2    POINT ZM (10.02000000 20.01000000 5.00000000 7.00000000)
```

PostgreSQL

```
CREATE TABLE startpoint_test (
 gid serial,
 ln1 sde.st_geometry
);
```

```
INSERT INTO startpoint_test (ln1) VALUES (
 sde.st_linestring ('linestring (10.02 20.01, 23.73 21.92, 30.10 40.23)', 4326)
);
INSERT INTO startpoint_test (ln1) VALUES (
 sde.st_linestring ('linestring zm(10.02 20.01 5 7, 23.73 21.92 6.5 7.1, 30.10 40.23
6.9 7.2)', 4326)
);
```

```
SELECT gid, sde.st_astext (sde.st_startpoint (ln1))
 AS Startpoint
 FROM startpoint_test;
gid  startpoint
1    POINT (10.02000000 20.01000000)
2    POINT ZM (10.02000000 20.01000000 5.00000000 7.00000000)
```

SQLite

```
CREATE TABLE startpoint_test (
 gid integer primary key autoincrement not null
);
SELECT AddGeometryColumn(
 NULL,
 'startpoint_test',
 'ln1',
 4326,
 'linestringzm',
 'xyzm',
 'null'
);
```

```
INSERT INTO startpoint_test (ln1) VALUES (
 st_linestring ('linestring (10.02 20.01, 23.73 21.92, 30.10 40.23)', 4326)
);
INSERT INTO startpoint_test(ln1) VALUES (
 st_linestring ('linestring zm(10.02 20.01 5 7, 23.73 21.92 6.5 7.1, 30.10 40.23 6.9
7.2)', 4326)
);
```

```
SELECT gid, st_astext (st_startpoint (ln1))
 AS "Startpoint"
 FROM startpoint_test;
```

```
gid  Startpoint
1    POINT (10.02000000 20.01000000)
2    POINT ZM (10.02000000 20.01000000 5.00000000 7.00000000)
```

ST_Surface

注意:
Oracle と SQLite のみ

定義

ST_Surface は、WKT 表現からサーフェス フィーチャを構築します。 サーフェスは、ポリゴンと類似していますが、その範囲内の各ポイントに値を持ちます。

構文

Oracle

```
sde.st_surface (wkt clob, srid integer)
```

SQLite

```
st_surface (wkt text, srid int32)
```

戻り値のタイプ

ST_Polygon

例

surf_test テーブルを作成し、サーフェス ジオメトリを挿入します。

Oracle

```
CREATE TABLE surf_test (
 id integer,
 geometry sde.st_geometry
);

INSERT INTO SURF_TEST VALUES (
 1110,
 sde.st_surface ('polygon ((110 120, 110 140, 120 130, 110 120))', 4326)
);
```

SQLite

```
CREATE TABLE surf_test (
 id integer
);

SELECT AddGeometryColumn(
 NULL,
 'surf_test',
 'geometry',
 4326,
```

```
 'polygon',
 'xy',
 'null'
);

INSERT INTO SURF_TEST VALUES (
 1110,
 st_surface ('polygon ((110 120, 110 140, 120 130, 110 120))', 4326)
);
```

ST_SymmetricDiff

説明

ST_SymmetricDiff は、2 つのジオメトリ オブジェクトを入力として、両方のソース オブジェクトに共通でない部分から構成されるジオメトリ オブジェクトを返します。

構文

Oracle および PostgreSQL

```
sde.st_symmetricdiff (geometry1 sde.st_geometry, geometry2 sde.st_geometry)
```

SQLite

```
st_symmetricdiff (geometry1 geometryblob, geometry2 geometryblob)
```

戻り値のタイプ

Oracle および PostgreSQL

ST_Geometry

SQLite

Geometryblob

例

特別な報告書を提出するため、集水域と有害物汚染範囲域で、交差していないエリアを判定しなくてはならない。

集水域テーブルには、id 列、集水域の名前を格納する列 (wname)、集水域エリアのジオメトリを格納する shape 列が含まれています。

汚染範囲テーブルは、id 列にサイトの ID、site ポイント列に各サイトの実際の地理的位置を格納します。

ST_Buffer 関数は、有害廃棄物区域のポイントの周囲にバッファーを作成します。ST_SymmetricDiff 関数は、バッファーされた有害廃棄物区域と集水域で、交差していないエリアのポリゴンを返します。

有害廃棄物区域と集水域の対称差は、2 つの領域から双方が交差する領域を除外して得られます。

Oracle

```
CREATE TABLE watershed (
 id integer,
 wname varchar(40),
 shape sde.st_geometry
);

CREATE TABLE plumes (
 id integer,
 site sde.st_geometry
);
```

```
INSERT INTO WATERSHED (ID, WNAME, SHAPE) VALUES (
 1,
 'Big River',
 sde.st_geometry ('polygon ((20 30, 30 30, 30 40, 20 40, 20 30))', 4326)
);

INSERT INTO WATERSHED (ID, WNAME, SHAPE) VALUES (
 2,
 'Lost Creek',
 sde.st_geometry ('polygon ((30 30, 30 50, 50 50, 50 30, 30 30))', 4326)
);

INSERT INTO WATERSHED (ID, WNAME, SHAPE) VALUES (
 3,
 'Szymborska Stream',
 sde.st_geometry ('polygon ((40 40, 40 60, 60 60, 60 40, 40 40))', 4326)
);

INSERT INTO PLUMES (ID, SITE) VALUES (
 20,
 sde.st_geometry ('point (60 60)', 4326)
);

INSERT INTO PLUMES (ID, SITE) VALUES (
 21,
 sde.st_geometry ('point (30 30)', 4326)
);
```

```
SELECT ws.id WS_ID,
 sde.st_area (sde.st_symmetricdiff (sde.st_buffer (p.site, .1), ws.shape)) AREA_NO_INT
 FROM PLUMES p, WATERSHED ws
 WHERE p.id = 20;

     SA_ID     AREA_NO_INT

         1     100.031393
         2     400.031393
         3     400.015697
```

PostgreSQL

```
CREATE TABLE watershed (
 id serial,
 wname varchar(40),
```

```
 shape sde.st_geometry
);

CREATE TABLE plumes (
 id serial,
 site sde.st_geometry
);
```

```
INSERT INTO watershed (wname, shape) VALUES (
 'Big River',
 sde.st_geometry ('polygon ((20 30, 30 30, 30 40, 20 40, 20 30))', 4326)
);

INSERT INTO watershed (wname, shape) VALUES (
 'Lost Creek',
 sde.st_geometry ('polygon ((30 30, 30 50, 50 50, 50 30, 30 30))', 4326)
);

INSERT INTO watershed (wname, shape) VALUES (
 'Szymborska Stream',
 sde.st_geometry ('polygon ((40 40, 40 60, 60 60, 60 40, 40 40))', 4326)
);

INSERT INTO plumes (site) VALUES (
 sde.st_geometry ('point (60 60)', 4326)
);

INSERT INTO plumes (site) VALUES (
 sde.st_geometry ('point (30 30)', 4326)
);
```

```
SELECT ws.id AS WS_ID,
 sde.st_area (sde.st_symmetricdiff (sde.st_buffer (p.site, .1), ws.shape)) AS "no
intersection"
 FROM plumes p, watershed ws
 WHERE p.id = 1;

    ws_id     no intersection

         1    100.031393502001
         2    400.031393502001
         3    400.01569751
```

SQLite

```
CREATE TABLE watershed (
 id integer primary key autoincrement not null,
 wname text(40)
);

SELECT AddGeometryColumn(
 NULL,
 'watershed',
 'shape',
 4326,
 'polygon',
 'xy',
```

```
 'null'
);

CREATE TABLE plumes (
 id integer primary key autoincrement not null
);

SELECT AddGeometryColumn(
 NULL,
 'plumes',
 'site',
 4326,
 'point',
 'xy',
 'null'
);
```

```
INSERT INTO watershed (wname, shape) VALUES (
 'Big River',
 st_geometry ('polygon ((20 30, 30 30, 30 40, 20 40, 20 30))', 4326)
);

INSERT INTO watershed (wname, shape) VALUES (
 'Lost Creek',
 st_geometry ('polygon ((30 30, 30 50, 50 50, 50 30, 30 30))', 4326)
);

INSERT INTO watershed (wname, shape) VALUES (
 'Szymborska Stream',
 st_geometry ('polygon ((40 40, 40 60, 60 60, 60 40, 40 40))', 4326)
);

INSERT INTO plumes (site) VALUES (
 st_geometry ('point (60 60)', 4326)
);

INSERT INTO plumes (site) VALUES (
 st_geometry ('point (30 30)', 4326)
);
```

```
SELECT ws.id AS WS_ID,
 st_area (st_symmetricdiff (st_buffer (p.site, .1), ws.shape)) AS "no intersection"
 FROM plumes p, watershed ws
 WHERE p.id = 1;

    WS_ID     no intersection

         1    400.031393502001
         2    100.031393502001
         3    400.01569751
```

ST_Touches

定義

ST_Touches は、2 つのジオメトリに共通のポイントがどちらのジオメトリの内部ともインターセクト (交差) していない場合に 1 (Oracle および SQLite) または t (PostgreSQL) を返します。それ以外の場合は、0 (Oracle および SQLite) または f (PostgreSQL) を返します。少なくとも 1 つのジオメトリが ST_LineString、ST_Polygon、ST_MultiLineString、または ST_MultiPolygon でなければなりません。

構文

Oracle および PostgreSQL

```
sde.st_touches (geometry1 sde.st_geometry, geometry2 sde.st_geometry)
```

SQLite

```
st_touches (geometry1 geometryblob, geometry2 geometryblob)
```

戻り値のタイプ

Boolean

例

GIS 技術者は、エンドポイントが別の下水管と交差する下水管のリストを上司から求められています。

3 つの列を持つ sewerlines テーブルを作成します。最初の sewer_id 列は、各下水管を一意に識別します。整数の class 列は、一般的に下水管の能力と関連する下水管のタイプを識別します。sewer 列は、下水管のジオメトリを格納します。

SELECT クエリは、ST_Touches 関数を使用して互いに交差する下水管のリストを返します。

Oracle

```
CREATE TABLE sewerlines (
 sewer_id integer,
 sewer sde.st_geometry
);
INSERT INTO SEWERLINES VALUES (
 1,
 sde.st_mlinefromtext ('multilinestring ((20 30, 30 30, 30 40, 20 40, 20 30))', 4326)
);
INSERT INTO SEWERLINES VALUES (
 2,
 sde.st_mlinefromtext ('multilinestring ((30 30, 30 50, 50 50, 50 30, 30 30))', 4326)
);
INSERT INTO SEWERLINES VALUES (
 3,
 sde.st_mlinefromtext ('multilinestring ((40 40, 40 60, 60 60, 60 40, 40 40))', 4326)
);
INSERT INTO SEWERLINES VALUES (
 4,
 sde.st_linestring ('linestring (60 60, 70 70)', 4326)
```

```
);
INSERT INTO SEWERLINES VALUES (
 5,
 sde.st_linestring ('linestring (30 30, 60 60)', 4326)
);
```

```
SELECT s1.sewer_id, s2.sewer_id
 FROM SEWERLINES s1, SEWERLINES s2
 WHERE sde.st_touches (s1.sewer, s2.sewer) = 1;
  SEWER_ID   SEWER_ID
         1          5
         3          4
         4          3
         4          5
         5          1
         5          3
         5          4
```

PostgreSQL

```
CREATE TABLE sewerlines (
 sewer_id serial,
 sewer sde.st_geometry);
INSERT INTO sewerlines (sewer) VALUES (
 sde.st_multilinestring ('multilinestring ((20 30, 30 30, 30 40, 20 40, 20 30))', 4326)
);
INSERT INTO sewerlines (sewer) VALUES (
 sde.st_multilinestring ('multilinestring ((30 30, 30 50, 50 50, 50 30, 30 30))', 4326)
);
INSERT INTO sewerlines (sewer) VALUES (
 sde.st_multilinestring ('multilinestring ((40 40, 40 60, 60 60, 60 40, 40 40))', 4326)
);
INSERT INTO sewerlines (sewer) VALUES (
 sde.st_linestring ('linestring (60 60, 70 70)', 4326)
);
INSERT INTO sewerlines (sewer) VALUES (
 sde.st_linestring ('linestring (30 30, 60 60)', 4326)
);
```

```
SELECT s1.sewer_id, s2.sewer_id
 FROM sewerlines s1, sewerlines s2
 WHERE sde.st_touches (s1.sewer, s2.sewer) = 't';
  SEWER_ID   SEWER_ID
         1          5
         3          4
         4          3
         4          5
         5          1
         5          3
         5          4
```

SQLite

```
CREATE TABLE sewerlines (
 sewer_id integer primary key autoincrement not null
```

```
);
SELECT AddGeometryColumn(
 NULL,
 'sewerlines',
 'sewer',
 4326,
 'geometry',
 'xy',
 'null'
);
INSERT INTO sewerlines (sewer) VALUES (
 st_multilinestring ('multilinestring ((20 30, 30 30, 30 40, 20 40, 20 30))', 4326)
);
INSERT INTO sewerlines (sewer) VALUES (
 st_multilinestring ('multilinestring ((30 30, 30 50, 50 50, 50 30, 30 30))', 4326)
);
INSERT INTO sewerlines (sewer) VALUES (
 st_multilinestring ('multilinestring ((40 40, 40 60, 60 60, 60 40, 40 40))', 4326)
);
INSERT INTO sewerlines (sewer) VALUES (
 st_linestring ('linestring (60 60, 70 70)', 4326)
);
INSERT INTO sewerlines (sewer) VALUES (
 st_linestring ('linestring (30 30, 60 60)', 4326)
);
```

```
SELECT s1.sewer_id, s2.sewer_id
 FROM SEWERLINES s1, SEWERLINES s2
 WHERE st_touches (s1.sewer, s2.sewer) = 1;
sewer_id   sewer_id
1          5
3          4
3          5
4          3
4          5
5          1
5          3
5          4
```

ST_Transform

定義

ST_Transform タスクは、2 次元 ST_Geometry データを入力として、指定した空間参照 ID (SRID) で指定された空間参照に変換した値を返します。

⚠ 注意:

st_register_spatial_column 関数を使用して空間列を PostgreSQL データベースに登録した場合、登録時の SRID が sde_geometry_columns テーブルに書き込まれます。 Oracle データベースの空間列に空間インデックスを作成した場合、空間インデックスの作成時の SRID が st_geometry_columns テーブルに書き込まれます。 ST_Transform を使用して ST_Geometry データの SRID を変更しても、sde_geometry_columns または st_geometry_columns テーブルの SRID は更新されません。

地理座標系が異なる場合、ST_Transform は地理座標系変換を実行します。 地理座標系変換は、2 つの地理座標系間で変換します。 地理座標系変換は、NAD 1927 から NAD 1983 というように、特定の方向に定義されますが、ST_Transform 関数は変換元と変換先の座標系に関係なく、変換を正しく適用します。

地理座標系変換には、数式をベースとした方式と、ファイルベースの方式の 2 つのタイプがあります。 数式をベースとした方式は、自己包含型で外部からの情報を必要としません。 ファイルベースの方式は、オフセット値の計算にディスク上のファイルを使用します。 通常、ファイルベースの方式の方が、数式をベースとした方式より正確です。 ファイルベースの方式は、オーストラリア、カナダ、ドイツ、ニュージーランド、スペイン、米国で一般的に使用されています。 ファイルは (カナダのものを除く)、ArcGIS Pro をインストールした場所にあります。また、さまざまな政府機関の地図作成部門からも直接入手できます。

ファイルベースの変換をサポートする場合、ファイルはデータベースがインストールされているサーバー上に、ArcGIS Pro のインストール ディレクトリの pedata フォルダーと同じ相対フォルダー構造で配置する必要があります。

たとえば、`pedata` というフォルダーは、ArcGIS Pro のインストール ディレクトリの `Resources` フォルダーにあります。 `pedata` フォルダーにはいくつかのサブフォルダーがありますが、サポートされているファイルベースの方式を含む 3 つのフォルダーは、`harn`、`nadcon`、`ntv2` です。 `pedata` フォルダーとその内容を ArcGIS のインストール ディレクトリからデータベース サーバーへコピーするか、データベース サーバーに、サポートされているファイルベースの変換方式のサブディレクトリとファイルを含むディレクトリを作成します。 ファイルがデータベース サーバーに配置されたら、同じサーバー上で PEDATAHOME という名前のオペレーティング システム環境変数を設定します。 PEDATAHOME 変数を、サブディレクトリとファイルが含まれているディレクトリの場所に設定します。たとえば、pedata フォルダーを Microsoft Windows サーバーの `C:\pedata` にコピーした場合、PEDATAHOME 環境変数は `C:\pedata` に設定します。

環境変数の設定方法については、お使いのオペレーティング システムのマニュアルをご参照ください。

PEDATAHOME を設定したら、ST_Transform 関数を使用する前に、新しい SQL セッションを開始する必要があります。ただし、サーバーを再起動する必要はありません。

PostgreSQL での ST_Transform の使用

PostgreSQL では、同じ地理座標系または異なる地理座標系の空間参照間で変換できます。

データが (ジオデータベースではなく) データベースに格納されている場合、地理座標系が同じときは次の手順で

ST_Geometry データの空間参照を変更します。

1. テーブルのバックアップ コピーを作成します。
2. テーブル上に 2 つ目の (変換先の) ST_Geometry 列を作成します。
3. 新しい SRID を指定して、変換先の ST_Geometry 列を登録します。
   これによって、sde_geometry_columns システム テーブルにレコードを配置し、列の空間参照を指定します。
4. ST_Transform 関数を実行し、変換後のデータが変換先の ST_Geometry 列に出力されるように指定します。
5. 最初の (変換元の) ST_Geometry 列の登録を解除します。

データがジオデータベースに格納されている場合は、ArcGIS ツールを使用してデータを新しいフィーチャクラスに再投影します。 ジオデータベース フィーチャクラスで ST_Transform を実行すると、新しい SRID でジオデータベース システム テーブルを更新する機能が無視されます。

Oracle での ST_Transform の使用

Oracle では、同じ地理座標系または異なる地理座標系の空間参照間で変換できます。

データが (ジオデータベースではなく) データベースに格納されていて、空間列に空間インデックスが定義されていない場合は、2 つ目の ST_Geometry 列を追加し、変換後のデータをそれに出力することができます。 元の (ソース) ST_Geometry 列と変換先の ST_Geometry 列を両方ともテーブル内に保持することはできますが、ArcGIS で、ビューを使用するか、テーブルのクエリ レイヤー定義を変更して一度に表示できるのは 1 つの列のみです。

データが (ジオデータベースではなく) データベースに格納されていて、空間列に空間インデックスが定義されている場合は、元の ST_Geometry 列を保持することはできません。 空間インデックスが ST_Geometry 列に定義されたら、SRID が st_geometry_columns メタデータ テーブルに書き込まれます。 ST_Transform はそのテーブルを更新しません。

1. テーブルのバックアップ コピーを作成します。
2. テーブル上に 2 つ目の (変換先の) ST_Geometry 列を作成します。
3. ST_Transform 関数を実行し、変換後のデータが変換先の ST_Geometry 列に出力されるように指定します。
4. 変換元の ST_Geometry 列から空間インデックスを削除します。
5. 変換元の ST_Geometry 列を削除します。
6. 変換先の ST_Geometry 列に空間インデックスを作成します。

データがジオデータベースに格納されている場合は、ArcGIS ツールを使用してデータを新しいフィーチャクラスに再投影します。ジオデータベース フィーチャクラスで ST_Transform を実行すると、新しい SRID でジオデータベース システム テーブルを更新する機能が無視されます。

SQLite での ST_Transform の使用

SQLite では、同じ地理座標系または異なる地理座標系の空間参照間で変換できます。

構文

変換元と変換先の空間参照が同じ地理座標系である場合

Oracle および PostgreSQL

```
sde.st_transform (geometry1 sde.st_geometry, srid integer)
```

SQLite

```
st_transform (geometry1 geometryblob, srid in32)
```

変換元と変換先の空間参照が同じ地理座標系ではない場合

Oracle

```
sde.st_transform (g1 sde.st_geometry, srid integer, geogtrans_id integer)
```

PostgreSQL

オプション 1: `sde.st_transform (g1 sde.st_geometry, srid int)`

オプション 2: `sde.st_transform (g1 sde.st_geometry, srid int, [geogtrans_id int])`

オプション 3: `sde.st_transform (g1 sde.st_geometry, srid int, [extent double] [prime meridian double] [unit conversion factor double])`

オプション 3 では、必要に応じて、カンマ区切りリストとして次の順序で座標の範囲を指定できます。左下の X 座標、左下の Y 座標、右上の X 座標、右上の Y 座標。 範囲を指定しない場合、ST_Transform は、より大きく一般的な範囲を使用します。

範囲を指定する場合、本初子午線と単位変換係数パラメーターはオプションです。 この情報は、指定する範囲値がグリニッジを通る本初子午線または度 (10 進) を使用しない場合のみ指定する必要があります。

SQLite

```
st_transform (geometry1 geometryblob, srid int32, geogtrans_id int32)
```

戻り値のタイプ

Oracle および PostgreSQL

ST_Geometry

SQLite

Geometryblob

例

変換元と変換先の空間参照が同じ地理座標系である場合のデータの変換

次の例では、ln1 と ln2 という 2 つのラインストリング列を持つ transform_test テーブルを作成します。 SRID 4326 のラインを ln1 を挿入します。 次に UPDATE ステートメントで ST_Transform 関数を使用して、ln1 のラインストリングを入力として、SRID 4326 に割り当てられた座標参照系から SRID 3857 に割り当てられた座標参照系に変換し、結果を ln2 列に配置します。

注意:
SRID 4326 と SRID 3857 は両方とも同じ測地基準系です。

Oracle

```
CREATE TABLE transform_test (
 ln1 sde.st_geometry,
 ln2 sde.st_geometry);

INSERT INTO transform_test (ln1) VALUES (
 sde.st_geometry ('linestring (10.01 40.03, 92.32 29.39)', 4326)
);
```

```
UPDATE transform_test
 SET ln2 = sde.st_transform (ln1, 3857);
```

PostgreSQL

```
CREATE TABLE transform_test (
 ln1 sde.st_geometry,
 ln2 sde.st_geometry);

INSERT INTO transform_test (ln1) VALUES (
 sde.st_geometry ('linestring (10.01 40.03, 92.32 29.39)', 4326)
);
```

```
UPDATE transform_test
 SET ln2 = sde.st_transform (ln1, 3857);
```

SQLite

```
CREATE TABLE transform_test (id integer);

SELECT AddGeometryColumn(
 NULL,
 'transform_test',
 'ln1',
 4326,
 'linestring',
 'xy',
 'null'
);

INSERT INTO transform_test (ln1) VALUES (
 st_geometry ('linestring (10.01 40.03, 92.32 29.39)', 4326)
);
```

```
UPDATE transform_test
 SET ln1 = st_transform (ln1, 3857);
```

変換元と変換先の空間参照の地理座標系が同じでない場合のデータの変換

次の例では、id 列と geometry 列を持つ n27 テーブルを作成します。 SRID が 4267 のポイントを n27 テーブルに挿入します。 4267 SRID は NAD 1927 地理座標系を使用します。

次に、n83 テーブルを作成し、ST_Transform 関数を使用して n27 テーブルから n83 テーブルにジオメトリを挿入しますが、SRID は 4269、地理座標変換 ID は 1241 です。 SRID 4269 は NAD 1983 地理座標系を使用します。 1241 は、NAD_1927_To_NAD_1983_NADCON 変換の有名な ID です。 この変換はファイルベースで、米国 (アラスカ/ハワイを除く 48 州) で使用できます。

ヒント:

サポートされている地理座標変換のリストについては、「Esri 技術資料 000004829」および記事の **[関連情報]** セクションに記載されているリンクをご参照ください。

Oracle

```
--Create table.
CREATE TABLE n27 (
 id integer,
 geometry sde.st_geometry
);

--Insert point with SRID 4267.
INSERT INTO N27 (id, geometry) VALUES (
 1,
 sde.st_geometry ('point (-123.0 49.0)', 4267)
);

--Create the n83 table as the destination table of the transformation.
CREATE TABLE n83 (
 id integer,
 geometry sde.st_geometry
);

--Run the transformation.
INSERT INTO N83 (id, geometry)(
 select c.id, sde.st_transform (c.geometry, 4269, 1241)
 from N27 c
);
```

PEDATAHOME が正しく定義されていれば、SELECT ステートメントが n83 テーブルに対して実行され、以下が返されます。

```
SELECT id, sde.st_astext (geometry) description
 FROM N83;

ID        DESCRIPTION
1 | POINT((-123.00130569 48.999828199))
```

PostgreSQL

```
--Option 1
--Gets geographic transformation from ST_Geometry libraries.
--Does not require you to provide a GTid.
```

```
--Performs an equation-based transformation between two geographic coordinate systems
--with different datums. (SRID 4267/DATUM NAD27 to SRID 4269/DATUM NAD 83)

--Provide point to transform.
SELECT sde.ST_AsText(sde.ST_Transform(
  sde.ST_Geometry('point (-155.7029 63.6096)',4267), 4269));

--Returns output in SRID 4269.
"POINT ( -155.70290000 63.60960000)"
```

```
--Option 2
--Example uses input point in SRID 3857(DATUM: WGS 1984)
--and geographic transformation ID (GTid) 1251.
--Transforms point to SRID 102008 (DATUM: NAD 83)

--Provide point to transform.
SELECT sde.ST_AsText(sde.ST_Transform(
  sde.ST_Geometry('point (-13244252.9404 4224702.5198)', 3857), 102008, 1251));

--Returns output in SRID 102008.
"POINT (-1957193.14740000 -297059.19680000)"
```

SQLite

```
--Create source table.
CREATE TABLE n27 (id integer);

SELECT AddGeometryColumn(
 NULL,
 'n27',
 'geometry',
 4267,
 'point',
 'xy',
 'null'
);

--Insert point with SRID 4267.
INSERT INTO n27 (id, geometry) VALUES (
 1,
 st_geometry ('point (-123.0 49.0)', 4267)
);

--Create the n83 table as the destination table of the transformation.
CREATE TABLE n83 (id integer);

SELECT AddGeometryColumn(
 NULL,
 'n83',
 'geometry',
 4269,
 'point',
 'xy',
 'null'
);

--Run the transformation.
INSERT INTO n83 (id, geometry) VALUES (
 1,
```

```
 st_transform ((select geometry from n27 where id=1), 4269, 1241)
);
```

ST_Union

定義

ST_Union は、2 つのソース オブジェクトを結合 (ユニオン) したジオメトリ オブジェクトを返します。

構文

Oracle および PostgreSQL

```
sde.st_union (geometry1 sde.st_geometry, geometry2 sde.st_geometry)
```

SQLite

```
st_union (geometry1 geometryblob, geometry2 geometryblob)
```

戻り値のタイプ

Oracle および PostgreSQL

ST_Geometry

SQLite

Geometryblob

例

sensitive_areas テーブルには、脅威を受ける施設の ID と、施設のポリゴン ジオメトリを格納する shape 列があります。

hazardous_sites テーブルは、id 列にサイトの ID、site ポイント列に各サイトの実際の地理的位置を格納します。

ST_Buffer 関数は、有害廃棄物区域の周囲にバッファーを作成します。ST_Union 関数は、バッファーが作成された有害廃棄物区域と保護区域のユニオンからポリゴンを作成します。ST_Area 関数は、これらのポリゴンの面積を返します。

Oracle

```
CREATE TABLE sensitive_areas (
 id integer,
 shape sde.st_geometry
);
CREATE TABLE hazardous_sites (
 id integer,
 site sde.st_geometry
);
INSERT INTO SENSITIVE_AREAS VALUES (
 1,
 sde.st_geometry ('polygon ((20 30, 30 30, 30 40, 20 40, 20 30))', 4326)
);
INSERT INTO SENSITIVE_AREAS VALUES (
 2,
```

```
 sde.st_geometry ('polygon ((30 30, 30 50, 50 50, 50 30, 30 30))', 4326)
);
INSERT INTO SENSITIVE_AREAS VALUES (
 3,
 sde.st_geometry ('polygon ((40 40, 40 60, 60 60, 60 40, 40 40))', 4326)
);
INSERT INTO HAZARDOUS_SITES VALUES (
 4,
 sde.st_geometry ('point (60 60)', 4326)
);
INSERT INTO HAZARDOUS_SITES VALUES (
 5,
 sde.st_geometry ('point (30 30)', 4326)
);
```

```
SELECT sa.id SA_ID, hs.id HS_ID,
 sde.st_area (sde.st_union (sde.st_buffer (hs.site, .01), sa.shape)) UNION_AREA
 FROM HAZARDOUS_SITES hs, SENSITIVE_AREAS sa;
     SA_ID  HS_ID     UNION_AREA
         1    4       100.000313935011
         2    4       400.000313935011
         3    4       400.000235451258
         1    5       100.000235451258
         2    5       400.000235451258
         3    5       400.000313935011
```

PostgreSQL

```
CREATE TABLE sensitive_areas (
 id integer,
 shape sde.st_geometry
);
CREATE TABLE hazardous_sites (
 id integer,
 site sde.st_geometry
);
INSERT INTO SENSITIVE_AREAS VALUES (
 1,
 sde.st_geometry ('polygon ((20 30, 30 30, 30 40, 20 40, 20 30))', 4326)
);
INSERT INTO SENSITIVE_AREAS VALUES (
 2,
 sde.st_geometry ('polygon ((30 30, 30 50, 50 50, 50 30, 30 30))', 4326)
);
INSERT INTO SENSITIVE_AREAS VALUES (
 3,
 sde.st_geometry ('polygon ((40 40, 40 60, 60 60, 60 40, 40 40))', 4326)
);
INSERT INTO HAZARDOUS_SITES VALUES (
 4,
 sde.st_geometry ('point (60 60)', 4326)
);
INSERT INTO HAZARDOUS_SITES VALUES (
 5,
 sde.st_geometry ('point (30 30)', 4326)
);
```

```
SELECT sa.id AS SA_ID, hs.id AS HS_ID,
 sde.st_area (sde.st_union (sde.st_buffer (hs.site, .01), sa.shape)) AS UNION_AREA
 FROM hazardous_sites hs, sensitive_areas sa;
     sa_id  hs_id   union_area
         1    4       100.000313935011
         2    4       400.000313935011
         3    4       400.000235451258
         1    5       100.000235451258
         2    5       400.000235451258
         3    5       400.000313935011
```

SQLite

```
CREATE TABLE sensitive_areas (
 id integer
);
SELECT AddGeometryColumn(
 NULL,
 'sensitive_areas',
 'shape',
 4326,
 'polygon',
 'xy',
 'null'
);
CREATE TABLE hazardous_sites (
 id integer
);
```

```
SELECT AddGeometryColumn(
 NULL,
 'hazardous_sites',
 'site',
 4326,
 'point',
 'xy',
 'null'
);
INSERT INTO sensitive_areas VALUES (
 10,
 st_geometry ('polygon ((20 30, 30 30, 30 40, 20 40, 20 30))', 4326)
);
INSERT INTO sensitive_areas VALUES (
 11,
 st_geometry ('polygon ((30 30, 30 50, 50 50, 50 30, 30 30))', 4326)
);
INSERT INTO sensitive_areas VALUES (
 12,
 st_geometry ('polygon ((40 40, 40 60, 60 60, 60 40, 40 40))', 4326)
);
INSERT INTO hazardous_sites VALUES (
 40,
 st_geometry ('point (60 60)', 4326)
);
INSERT INTO hazardous_sites VALUES (
 41,
 st_geometry ('point (30 30)', 4326)
);
```

```
SELECT sa.id AS "sa_id", hs.id AS "hs_id",
 st_area (st_union (st_buffer (hs.site, .01), sa.shape)) AS "union"
 FROM hazardous_sites hs, sensitive_areas sa;
     sa_id  hs_id   union
         1    4       100.000313935011
         2    4       400.000313935011
         3    4       400.000235451258
         1    5       100.000235451258
         2    5       400.000235451258
         3    5       400.000313935011
```

ST_Within

定義

ST_Within は、最初の ST_Geometry オブジェクトが 2 番目のオブジェクトの中に完全に入っている場合は 1 (Oracle および SQLite) または t (PostgreSQL)、それ以外の場合は 0 (Oracle および SQLite) または f (PostgreSQL) を返します。

構文

Oracle および PostgreSQL

```
sde.st_within (geometry1 sde.st_geometry, geometry2 sde.st_geometry)
```

SQLite

```
st_within (geometry1 geometryblob, geometry2 geometryblob)
```

戻り値のタイプ

Boolean

例

以下の例では、zones と squares という 2 つのテーブルが作成されます。SELECT ステートメントにより、交差しているが、1 つの区画内に完全には含まれないすべての四角形が検索されます。

Oracle

```
CREATE TABLE squares (
 id integer,
 shape sde.st_geometry);
CREATE TABLE zones (
 id integer,
 shape sde.st_geometry);
INSERT INTO squares (id, shape) VALUES (
 1,
 sde.st_polygon ('polygon ((0 0, 0 10, 10 10, 10 0, 0 0))', 4326)
);
INSERT INTO squares (id, shape) VALUES (
 2,
 sde.st_polygon ('polygon ((20 0, 20 10, 30 10, 30 0, 20 0))', 4326)
);
INSERT INTO squares (id, shape) VALUES (
 3,
 sde.st_polygon ('polygon ((40 0, 40 10, 50 10, 50 0, 40 0))', 4326)
);
INSERT INTO zones (id, shape) VALUES (
 1,
 sde.st_polygon ('polygon ((-1 -1, -1 11, 11 11, 11 -1, -1 -1))', 4326)
);
INSERT INTO zones (id, shape) VALUES (
 2,
 sde.st_polygon ('polygon ((19 -1, 19 11, 29 9, 31 -1, 19 -1))', 4326)
```

```
);
INSERT INTO zones (id, shape) VALUES (
 3,
 sde.st_polygon ('polygon ((39 -1, 39 11, 51 11, 51 -1, 39 -1))', 4326)
);
```

```
SELECT s.id sq_id
 FROM SQUARES s, ZONES z
 WHERE sde.st_intersects (s.shape, z.shape) = 1
 AND sde.st_within (s.shape, z.shape) = 0;
SQ_ID
2
```

PostgreSQL

```
CREATE TABLE squares (
 id integer,
 shape sde.st_geometry);
CREATE TABLE zones (
 id integer,
 shape sde.st_geometry);
INSERT INTO squares (id, shape) VALUES (
 1,
 sde.st_polygon ('polygon ((0 0, 0 10, 10 10, 10 0, 0 0))', 4326)
);
INSERT INTO squares (id, shape) VALUES (
 2,
 sde.st_polygon ('polygon ((20 0, 20 10, 30 10, 30 0, 20 0))', 4326)
);
INSERT INTO squares (id, shape) VALUES (
 3,
 sde.st_polygon ('polygon ((40 0, 40 10, 50 10, 50 0, 40 0))', 4326)
);
INSERT INTO zones (id, shape) VALUES (
 1,
 sde.st_polygon ('polygon ((-1 -1, -1 11, 11 11, 11 -1, -1 -1))', 4326)
);
INSERT INTO zones (id, shape) VALUES (
 2,
 sde.st_polygon ('polygon ((19 -1, 19 11, 29 9, 31 -1, 19 -1))', 4326)
);
INSERT INTO zones (id, shape) VALUES (
 3,
 sde.st_polygon ('polygon ((39 -1, 39 11, 51 11, 51 -1, 39 -1))', 4326)
);
```

```
SELECT s.id
 AS sq_id
 FROM squares s, zones z
 WHERE st_intersects (s.shape, z.shape) = 't'
 AND st_within (s.shape, z.shape) = 'f';
sq_id
2
```

SQLite

```
CREATE TABLE squares (
 id integer
);
SELECT AddGeometryColumn(
 NULL,
 'squares',
 'shape',
 4326,
 'polygon',
 'xy',
 'null'
);
CREATE TABLE zones (
 id integer
);
SELECT AddGeometryColumn(
 NULL,
 'zones',
 'shape',
 4326,
 'polygon',
 'xy',
 'null'
);
INSERT INTO squares (id, shape) VALUES (
 1,
 st_polygon ('polygon ((0 0, 0 10, 10 10, 10 0, 0 0))', 4326)
);
INSERT INTO squares (id, shape) VALUES (
 2,
 st_polygon ('polygon ((20 0, 20 10, 30 10, 30 0, 20 0))', 4326)
);
INSERT INTO squares (id, shape) VALUES (
 3,
 st_polygon ('polygon ((40 0, 40 10, 50 10, 50 0, 40 0))', 4326)
);
INSERT INTO zones (id, shape) VALUES (
 1,
 st_polygon ('polygon ((-1 -1, -1 11, 11 11, 11 -1, -1 -1))', 4326)
);
INSERT INTO zones (id, shape) VALUES (
 2,
 st_polygon ('polygon ((19 -1, 19 11, 29 9, 31 -1, 19 -1))', 4326)
);
INSERT INTO zones (id, shape) VALUES (
 3,
 st_polygon ('polygon ((39 -1, 39 11, 51 11, 51 -1, 39 -1))', 4326)
);
```

```
SELECT s.id
 AS "sq_id"
 FROM squares s, zones1 z
 WHERE st_intersects (s.shape, z.shape) = 1
 AND st_within (s.shape, z.shape) = 0;
sq_id
2
```

ST_X

定義

ST_X は、ST_Point を入力パラメーターとして、その X 座標を返します。SQLite では、ST_X は ST_Point の X 座標を更新することもできます。

構文

Oracle および PostgreSQL

```
sde.st_x (point1 sde.st_point)
```

SQLite

```
st_x (point1 geometryblob)
st_x (input_point geometryblob, new_Xvalue double)
```

戻り値のタイプ

Double precision

SQLite では、ST_X 関数を使用してポイントの X 座標を更新できます。その場合は geometryblob が返されます。

例

行を一意に識別する gid 列と、pt1 ポイント列を持つ x_test テーブルを作成します。

INSERT ステートメントは、2 つの行を挿入します。1 つは、Z 座標またはメジャー値のないポイントです。もう 1 つは、Z 座標とメジャー値があるポイントです。

SELECT クエリは、ST_X 関数を使用して各ポイント フィーチャの X 座標を取得します。

Oracle

```
CREATE TABLE x_test (
 gid integer unique,
 pt1 sde.st_point
);
```

```
INSERT INTO X_TEST VALUES (
 1,
 sde.st_pointfromtext ('point (10.02 20.01)', 4326)
);
INSERT INTO X_TEST VALUES (
 2,
 sde.st_pointfromtext ('point zm(10.1 20.01 5 7)', 4326)
);
```

```
SELECT gid, sde.st_x (pt1) "The X coordinate"
 FROM X_TEST;
```

```
       GID       The X coordinate
         1            10.02
         2            10.10
```

PostgreSQL

```
CREATE TABLE x_test (
 gid integer unique,
 pt1 sde.st_point
);
```

```
INSERT INTO x_test VALUES (
 1,
 sde.st_point ('point (10.02 20.01)', 4326)
);
INSERT INTO x_test VALUES (
 2,
 sde.st_point ('point zm(10.1 20.01 5 7)', 4326)
);
```

```
SELECT gid, sde.st_x (pt1)
 AS "The X coordinate"
 FROM x_test;
       gid       The X coordinate
         1            10.02
         2            10.10
```

SQLite

```
CREATE TABLE x_test (gid integer);
SELECT AddGeometryColumn(
 NULL,
 'x_test',
 'pt1',
 4326,
 'pointzm',
 'xyzm',
 'null'
);
```

```
INSERT INTO x_test VALUES (
 1,
 st_point ('point (10.02 20.01)', 4326)
);
INSERT INTO x_test VALUES (
 2,
 st_point ('point zm(10.1 20.01 5 7)', 4326)
);
```

```
SELECT gid, st_x (pt1)
```

```
 AS "The X coordinate"
 FROM x_test;
       gid       The X coordinate
         1            10.02
         2            10.10
```

ST_X 関数を使用して、既存のポイントの座標値を更新することもできます。この例では、ST_X を使用して、x_test の最初のポイントの X 座標値を更新します。

```
UPDATE x_test
 SET pt1=st_x(
  (SELECT pt1 FROM x_test WHERE gid=1),
  10.04
  )
 WHERE gid=1;
```

ST_Y

定義

ST_Y は、ST_Point を入力パラメーターとして、その Y 座標を返します。SQLite では、ST_Y は ST_Point の Y 座標を更新することもできます。

構文

Oracle および PostgreSQL

```
sde.st_y (point1 sde.st_point)
```

SQLite

```
double   st_y (point1  geometryblob)
geometry st_y (input_shape geometryblob, new_Yvalue double)
```

戻り値のタイプ

Double precision

SQLite では、ST_Y 関数を使用してポイントの Y 座標を更新できます。その場合は geometryblob が返されます。

例

行を一意に識別する gid 列と、pt1 ポイント列を持つ y_test テーブルを作成します。

INSERT ステートメントは、2 つの行を挿入します。1 つは、Z 座標またはメジャー値のないポイントです。もう 1 つは、Z 座標とメジャー値があるポイントです。

SELECT クエリは、ST_Y 関数を使用して各ポイント フィーチャの Y 座標を取得します。

Oracle

```
CREATE TABLE y_test (
 gid integer unique,
 pt1 sde.st_point
);
```

```
INSERT INTO Y_TEST VALUES (
 1,
 sde.st_pointfromtext ('point (10.02 20.02)', 4326)
);
INSERT INTO Y_TEST VALUES (
 2,
 sde.st_pointfromtext ('point zm(10.1 20.01 5.0 7.0)', 4326)
);
```

```
SELECT gid, sde.st_y (pt1) "The Y coordinate"
 FROM Y_TEST;
```

```
       GID     The Y coordinate
         1          20.02
         2          20.01
```

PostgreSQL

```
CREATE TABLE y_test (
 gid integer unique,
 pt1 sde.st_point
);
```

```
INSERT INTO y_test VALUES (
 1,
 sde.st_point ('point (10.02 20.02)', 4326)
);
INSERT INTO y_test VALUES (
 2,
 sde.st_point ('point zm(10.1 20.01 5.0 7.0)', 4326)
);
```

```
SELECT gid, sde.st_y (pt1)
 AS "The Y coordinate"
 FROM y_test;
       gid    The Y coordinate
         1          20.02
         2          20.01
```

SQLite

```
CREATE TABLE y_test (gid integer);

SELECT AddGeometryColumn(
 NULL,
 'y_test',
 'pt1',
 4326,
 'pointzm',
 'xyzm',
 'null'
);
```

```
INSERT INTO y_test VALUES (
 1,
 st_point ('point (10.02 20.02)', 4326)
);
INSERT INTO y_test VALUES (
 2,
 st_point ('point zm(10.1 20.01 5.0 7.0)', 4326)
);
```

```
SELECT gid, st_y (pt1)
 AS "The Y coordinate"
 FROM y_test;
gid    The Y coordinate
1          20.02
2          20.01
```

ST_Y 関数を使用して、既存のポイントの座標値を更新することもできます。この例では、ST_Y を使用して、y_test の最初のポイントの Y 座標値を更新します。

```
UPDATE y_test
 SET pt1=st_y(
  (SELECT pt1 FROM y_test WHERE gid=2),
  20.1
  )
 WHERE gid=2;
```

ST_Z

定義

ST_Z は、ST_Point を入力パラメーターとして、その Z (標高) 座標を返します。SQLite では、ST_Z は ST_Point の Z 座標を更新することもできます。

構文

Oracle および PostgreSQL

```
sde.st_z (geometry1 sde.st_point)
```

SQLite

```
st_z (geometry  geometryblob)
st_z (input_shape geometryblob, new_Zvalue double)
```

戻り値のタイプ

Oracle

Number

PostgreSQL

Integer

SQLite

ST_Z を使用してポイントの Z 座標を返す場合、Double precision が返されます。ST_Z を使用してポイントの Z 座標を更新する場合、geometryblob が返されます。

例

行を一意に識別する gid 列と、geometry ポイント列を持つ z_test テーブルを作成します。INSERT ステートメントは、z_test テーブルに行を挿入します。

SELECT ステートメントは、前のステートメントで挿入されたポイントの id 列と倍精度の Z 座標をリストします。

Oracle

```
CREATE TABLE z_test (
 id integer unique,
 geometry sde.st_point
);
INSERT INTO z_test (id, geometry) VALUES (
 1,
 sde.st_point (2, 3, 32, 5, 4326)
);
```

```
SELECT id, sde.st_z (geometry) Z_COORD
 FROM Z_TEST;
        ID      Z_COORD
         1        32
```

PostgreSQL

```
CREATE TABLE z_test (
 id integer unique,
 geometry sde.st_point
);
INSERT INTO z_test (id, geometry) VALUES (
 1,
 sde.st_point (2, 3, 32, 5, 4326)
);
```

```
SELECT id, sde.st_z (geometry)
 AS Z_COORD
 FROM z_test;
        id      z_coord
         1        32
```

SQLite

```
CREATE TABLE z_test (id integer);
SELECT AddGeometryColumn(
 NULL,
 'z_test',
 'pt1',
 4326,
 'pointzm',
 'xyzm',
 'null'
);
INSERT INTO z_test (id, pt1) VALUES (
 1,
 st_point (2, 3, 32, 5, 4326)
);
```

```
SELECT id, st_z (pt1)
 AS "The z coordinate"
```

```
 FROM z_test;
id      The z coordinate
1        32.0
```

ST_Z 関数を使用して、既存のポイントの座標値を更新することもできます。この例では、ST_Z を使用して、z_test の最初のポイントの Z 座標値を更新します。

```
UPDATE z_test
 SET pt1=st_z(
  (SELECT pt1 FROM z_test where id=1), 32.04)
 WHERE id=1;
```