precisely

Location Intelligence SDK

User Guide

Version 1.14

Table of Contents

1 - Location Intelligence SDK

In this section

What is the Location Intelligence SDK?

The Precisely Location Intelligence SDK is a spatial toolkit that allows access to functions through an in-process API. This allows certain fundamental location intelligence functions to be integrated into any solution and architecture including big data platforms. This version of the Location Intelligence SDK contains geometry and coordinate operations, the ability to read TAB and SHP files, in-memory r-tree creation and searching, and vector tile creation in **MapBox Vector Tile** (MVT) format.

Provided in this product:

- The SDK, a core set of jar files.
- Samples to use as reference.
- JavaDocs providing descriptions of the SDK API.
- The User Guide describing core features and examples.

Note: The JavaDocs provide information and examples for the use of r-tree creation and searching.

System Requirements and Installation

The Location Intelligence SDK is a collection of jar files that can be deployed to any system, simply by extracting the files from the zip file to your desired location. There is no installation.

The Location Intelligence SDK works with JDK 7 and higher.

2 - Working with Geometries

In this section

Geometry API

The primary interface is `IGeometry` which defines the basic behavior of the most used geometry classes (point, line, polygon, rectangle, etc.). For example, it defines functionality for returning a geometry object's type, its coordinate system information, and an envelope that defines a minimum bounding rectangle.

Depending on the application, and the data being used for your geographic information, there are different geometry formats that are used to represent geometries in spatial databases. For example, we support WKT, WKB, GeoJSON, and KML. All have their own structure and method of defining the geometry.

Definitions and Concepts

Our geometry model is based on OGC's **Simple Feature Specification**.

`Point`: An object defined by a single X,Y coordinate pair.

`Curve`: An object made up of two or more points.

`Polygon`: An object forming a closed bounded region made up of multiple sets of points closed by the first and last point.

`GeometryFactory` is a factory class for creating geometries. See **Creating Geometries Programmatically** on page 7.

`Well-known text (WKT)` is an OCG standard for representing geometries and their coordinate systems in a text based markup language. The WKT format is described here: **opengeospatial.org/standards/sfa** (section 7).

`Well-known binary (WKB)` is the binary equivalent to WKT and is used to transfer and store the same information. The format is described here: **opengeospatial.org/standards/sfa** (section 8).

`Keyhole Markup Language (KML)` is an OGC implementation standard for visualizing geometries in a web environment. KML is described here: **opengeospatial.org/standards/kml**.

`GeoJSON` is an open standard designed as a data interchange format for geographical features and their non-spatial attributes. It is based on JavaScript Object Notation. GeoJSON is described here: **geojson.org/geojson-spec.html**.

Creating Geometries Programmatically

In order to programmatically create new geometry instances, the Location Intelligence SDK provides a single factory class. All geometry creation operations go through this geometry factory. Simple geometries can be constructed with a single method call. For more complex geometries, the geometry factory provides a fluent builder API. This enables the creation of complex geometries from simpler parts using a compact and easily readable syntax.

To create a new geometry factory, the API offers several static factory methods on the geometry factory class. In the code snippet below, we create a new factory instance for a given EPSG code.

```
GeometryFactory factory = GeometryFactory.forSrsName("EPSG:4326");
```

The newly created factory provides access to various geometry creation methods, ranging from methods for simple points to multi-polygons. In order to create a new point, perform the following:

```
IPoint point = factory.newPoint(1.23, 2.34);
```

A multi-point can be created by specifying multiple point coordinates at once.

```
IMultiPoint points = factory.newMultiPoint(
  pos(1.23, 2.34), pos(3.45, 4.56));
```

The use of the `pos()` method is a convenience method to create direct position instances. It is statically imported from the GeometryFactory class.

Points can also be added individually:

```
IMultiPoint points = factory.newMultiPoint()
  .withPoint(1.23, 2.34)
  .withPoint(3.45, 4.56)
 .build();
```

To create a polygon, we need to specify an outer ring and zero or more inner rings:

```
IPolygon polygon = factory.newPolygon()
  .withOuterRing(). withLineString(
     pos(1, 1), pos(2, 1),
     pos(2, 2), pos(1, 2), pos(1, 1))
   .build()
  .withInnerRing().withLineString(pos(1.1, 1.1),
     pos(1.4, 1.1), pos(1.4, 1.4), pos(1.1, 1.4),
     pos(1.1, 1.1))
   .build()
  .withInnerRing(). withLineString(pos(1.6, 1.6),
     pos(1.9, 1.6), pos(1.9, 1.9), pos(1.6, 1.9),
     pos(1.6, 1.6))
   .build()
 .build();
```

A new multipolygon can be constructed as follows:

```
   IMultiPolygon polygons = factory.newMultiPolygon()
    .withPolygon()
     .withOuterRing()
      .withLineString(pos(1, 1), pos(2, 1), pos(2, 2),
       pos(1, 2), pos(1, 1))
      .build()
     .build()
    .withPolygon()
     .withOuterRing()
      .withLineString(pos(3, 1), pos(4, 1), pos(4, 2),
       pos(3, 2), pos(3, 1))
      .build()
     .build()
    .build();
```

Geometry Operations

Geometry operations are used to perform a calculation or transformation on a geometry as well as see the relationship between geometries. The core set of geometry operations are in the `GeometryOperations` class. Since a coordinate system transformation is altering a geometry's coordinate system, it is also considered a geometry operation. The coordinate system transformation uses the `IGeometry.getCopy(CoordSys)` method to transform a geometry.

The following geometry operations are supported:

Table 1: Supported Geometry Operations

Operation	Description
Area	Returns the area of the geometry. The area of a polygon is computed as the area of its exterior ring minus the areas of its interior rings. Points, lines and curves have zero area.
Buffer	Returns a geometry consisting of all the points within the specified distance of the given geometry. A positive distance results in a larger geometry. A negative distance shrinks polygons, and causes points and curves to disappear.

Operation	Description
ClosestPoints	Returns the closest points between the two geometries. Distance is always non-negative (objects which intersect are distance zero from each other). In this case, a shared point will be returned.
ConvexHull	Forms the convex hull of the geometry (the smallest bounding geometry encompassing all of the specified geometries).
Disjoint	A boolean operation that tests if geometries have any points in common.
Distance	Returns the minimum distance between two geometries. Distance is always non-negative (objects which intersect are distance zero from each other).
Intersection	Returns the geometry consisting of all parts that are common between both specified geometries.
Intersects	A boolean operation that returns true if there is any direct position in common between the two geometries.
Length	Computes the length of a geometry (all curves).
Overlaps	A boolean operation that tests if two geometries overlap one another. Must be a perfect match. Otherwise it will return a false value in cases where either a within or contains operations would return true.
Perimeter	Returns the total perimeter of the geometry. The perimeter of a polygon is the sum of the lengths of its rings (both exterior and interior holes). Curves are treated as thin polygons.
Union	Computes the joining of the two geometries, and returns the geometry that represents the union of the specified geometries.
Within	A boolean operation that determines if the first geometry is within the second. For example, a line is within a polygon if the line does not extend outside the polygon, and the line is not only along the boundary of the polygon.
Transform	Returns the specified source position transformed to the destination coordinate system..

Note: For more information on the geometry operations and their methods, see the JavaDocs included with Location Intelligence SDK.

Samples

The following are a few samples of geometry operations. For a working example of these geometry operations, see the `GeometryOperationsExamples.java` file in the samples included with the Location Intelligence SDK.

Geometry Buffering: this example first creates a point, and then buffers that point by 500 meters. The result will be a circular polygon.

```
GeometryFactory factory =
  GeometryFactory.forCoordSys(CoordSysConstants.longLatWGS84);

IFeatureGeometry point = factory.newPoint(-74.0059, 40.7128);

Offset distance = new Offset(500, LinearUnit.METER);

IFeatureGeometry bufferGeometry =
  GeometryOperations.buffer(point, distance);
```

Calculating Distance: this example calculates the shortest distance between a point and a polygon. The result is a distance value in Miles.

```
GeometryFactory factory =
  GeometryFactory.forCoordSys(CoordSysConstants.longLatWGS84);

IFeatureGeometry geometry = factory.newPoint(-102.0059, 40.7128);

IFeatureGeometry multiPolygon = factory.newMultiPolygon()
  .withPolygon()
   .withOuterRing()
    .withLineString(
     pos(-80.707968, 43.750565), pos(-80.707968, 43.750565),
     pos(-80.67604, 43.822318), pos(-80.560607, 43.844464),
     pos(-80.450086, 43.830291), pos(-80.362897, 43.793971),
     pos(-80.707968, 43.750565))
    .build()
   .build()
  .build();

double distanceInMile = GeometryOperations.distance(
  geometry, multiPolygon).getValue(LinearUnit.MILE);
```

Geometry Intersects: this example will return a boolean value if the point intersects the polygon.

```
GeometryFactory factory =
  GeometryFactory.forCoordSys(CoordSysConstants.longLatWGS84);

IFeatureGeometry multiCurve = factory.newMultiCurve().withLineString(

  pos(-102.707968, 43.750565), pos(-80.54604, 43.822318))
  .build();
  IFeatureGeometry multiPolygon = factory.newMultiPolygon()
    .withPolygon()
     .withOuterRing()
      .withLineString(
       pos(-80.707968, 43.750565), pos(-80.67604, 43.822318),
       pos(-80.560607, 43.844464), pos(-80.450086, 43.830291),
       pos(-80.362897, 43.793971), pos(-80.707968, 43.750565))
      .build()
     .build()
    .build();

  boolean result = GeometryOperations.intersects(multiCurve,
multiPolygon);
```

Determining Within: this example will return a boolean value if first geometry (point) is within the second geometry (polygon).

```
  GeometryFactory factory =
    GeometryFactory.forCoordSys(CoordSysConstants.longLatWGS84);

  IFeatureGeometry point = factory.newPoint(-74.0059, 40.7128);

  IFeatureGeometry multiPolygon = factory.newMultiPolygon()
    .withPolygon()
     .withOuterRing()
      .withLineString(
        pos(-80.707968, 43.750565), pos(-80.67604, 43.822318),
        pos(-80.560607, 43.844464), pos(-80.450086, 43.830291),
        pos(-80.362897, 43.793971), pos(-80.707968, 43.750565))
      .build()
     .build()
    .build();

  boolean result = GeometryOperations.within(point, multiPolygon);
```

Transforming Coordinates: this example takes the linestring in WGS84 and transforms it into the Web Mercator EPSG:3857 coordinate system. The result is the transformed geometry.

```
  GeometryFactory factory =
    GeometryFactory.forCoordSys(CoordSysConstants.longLatWGS84);

  IFeatureGeometry geometryWGS84 = factory.newMultiCurve()
    .withLineString(
      pos(-102.707968, 43.750565),
      pos(-80.54604, 43.822318))
    .build();

  CoordSys coordsys3857 =
   CoordSysFactory.getDefaultCoordSysFactory().getCoordSys("epsg:3857");
```

```
IFeatureGeometry geometry3857 = geometryWGS84.getCopy(coordsys3857);
```

3 - Data Providers

In this section

Introduction to Data Providers

A typical early step in using the Location Intelligence SDK is connecting to your data. This release allows you to work with TAB (MapInfo tables) and SHP (ESRI Shape) files through the use of data providers. TAB and Shapefiles are popular geospatial vector data formats for geographic information systems software. Support for these files in the Location Intelligence API is limited to read-only, as you cannot modify these files. If you need to create or modify these files, use a suitable application like MapInfo Pro.

Understanding the Table Model

In order to work with TAB or SHP files, you must understand the table model. A table is the client-side representation of the data. The main abstraction for tables is an `IDataSourceTable`. A data source table is a feature collection with additional methods to search for features and to query the table metadata. Each feature represents a single record (row) in the table.

File-Based Data Tables (MapInfo TAB and ESRI Shapefiles)

This release of the Location Intelligence SDK supports two file-based data tables: MapInfo TAB (or Native) and ESRI Shapefiles.

MapInfo TAB

TAB files contain three unique and important columns:

- `Obj` - Geometry column holding all geometry objects (point, line, polygon, etc.)
- `MI_Style` - Pseudo column that contains style information on a per-feature basis
- `MapInfo_ID` - Pseudo column that acts as the primary key column identifying a unique row in the table, and can be used or asked for in a query.

The `NativeDataProvider` class is the entry point when working with TAB files. For more detail on working with the data provider, see the JavaDocs.

There are many different types of TAB tables. For this release, the following TAB file types are supported:

- MapInfo TAB files
- MapInfo Extended – Type NATIVEX

The following types of TAB files are not supported:

- Raster
- Seamless
- View

The native data provider supports the filter language (aggregation and MISQL languages are not supported). The following filters are supported in the native data provider:

- AttributeFilter
- EnvelopeFilter
- InListFilter
- KeyFilter
- NearestFilter

ESRI Shapefiles

Shapefiles contain two unique and important columns:

- `the_geom` - Geometry column holding all geometry objects (point, line, polygon, etc.)
- `MapInfo_ID` - Pseudo column that acts as the primary key column identifying a unique row in the table, and can be used or asked for in a query.

The `ShapeDataProvider` class is the entry point when working with Shapefiles. For more detail on working with the data provider, see the JavaDocs.

The shape data provider supports the filter language (aggregation and MISQL languages are not supported). For this release, the following filter is supported in the shape data provider:

- EnvelopeFilter

Using a Data Provider

Opening a Native Table

To start using the native data provider and TAB files, you first need to open the table. Once a table is opened, it represents the client side representation of the data, and you can start to describe the data (table metadata), filter the data, or perform operations on the data.

To open a table, use the following:

```
try (IDataSourceTable table =
  NativeDataProvider.getInstance().openTable(tableName)) {
    // ...
}
```

Opening a Shape Table

To start using the Shape data provider and Shapefiles, you first need to open the table. Once a table is opened, it represents the client side representation of the data, and you can start to describe the data (table metadata), filter the data, or perform operations on the data.

To open a table, use the following:

```
try (IDataSourceTable table =
  ShapeDataProvider.getInstance().openTable(tableName)) {
 // ...
}
```

Getting Table Metadata

Describing a TAB or SHP file will allow you to understand the structure of the table, including the columns and content description of the table.

To return the table's metadata (and get the column names), use the following:

```
for (IAttributeDefinition attributeDefinition :
  table.getMetadata().getAttributeDefinitions()) {
 System.out.println("Attribute name: " + attributeDefinition.getName());

}
```

Searching a Table

There are essentially two ways of accessing table data: search a table and return all of the data, or use a supported filter with some criteria to follow and return a portion of the data. Returning all the data is sometimes necessary.

To search and return all table data, you must iterate the entire table and provide a search all filter. To iterate through the entire table, use the following:

```
ICursor cursor = table.search(FilterSearch.searchALL());
```

To search the entire table and return a subset of data, choose which columns to return and use a filter to provide a search criteria to limit the results. To return only the wanted column information, you can create a select list of query attributes (columns). In this case a table consisting of States:

```
SelectList selectList = new SelectList("State_name", "State", "Obj");
```

Based on the above array of attributes to return, you can provide a filter to search for a criteria. In this case a State name:

```
InListFilter inListFilter = new InListFilter("State",
  new AllType("NY"), new AllType("NJ"),
  new AllType("CA"), new AllType("AZ"));

FilterSearch filterSearch
  = new FilterSearch(selectList, inListFilter, null);

ICursor cursor = table.search(filterSearch);
```

For more information on filters, see **Using Filters** on page 20.

Consuming Data

When working with data from tables using the native or Shape data provider, there are some best practices when consuming the data. If you need to reuse a feature, you must make a copy of it, as features returned from a table are transient. A transient feature is a feature whose data is connected to the memory of the feature iterator. When the next feature is accessed, the same memory is reused, thus making the previous feature invalid.

The following example shows how to iterate through each feature, and get different attributes by making copies:

```
List<String> stateNameColums = new LinkedList<>();
try {
 while (cursor.hasNext()) {
  // Get a feature
  IFeature feature = cursor.next();

  // if using the feature outside the loop, get a copy of that feature

  // IFeature featureCopy = feature.getCopy();

  // get information from the Feature geometry column
 IFeatureGeometry featureGeometry = feature.getFeatureGeometry("Obj");

  //State_name column
  stateNameColums.add(feature.getString("State_name"));
 }
}
finally {
 cursor.dispose();
}
```

Note: When using file-based data (such as a TAB file), the cursor may hold on to resources (e.g., file locks, file readers, etc) It is a good practice to always dispose of the cursor and table to indicate that it may release its resources.

4 - Filters

In this section

Using Filters

About Filters

The filter model provides a mechanism to reduce the data returned from a table based on some criteria. In addition to returning fewer results, filtering aids in performance by not having to search the entire table. Filtering is best done close to the data, so support for each filter depends on the data provider. For example, the Shape data provider supports the Envelope filter, while the Native data provider supports the Attribute, InList, Key and Nearest filters, in addition to the Envelope filter.

The following sections describe various types of filters and how they may be used. For more information on all supported filters, see the `com.mapinfo.midev.language.filter` package in the JavaDocs included in your Location Intelligence SDK download.

Attribute Filter

Compares the values in a table against specified values.

For example, to return all the records where the value of the "State_Name" column is "New York:

```
IFilter filter = AttributeFilter.equalsFilter("State_Name",
  new AllType("New York"));
```

Or, return all the records where the value of the "POP_1990" column is greater than 5 million:

```
IFilter filter = AttributeFilter.greaterThanFilter("POP_1990",
  new AllType(5000000));
```

Between Filter

Compares the values in a table to a bounding pair (returns everything between and including) of specified values.

For example, to return all the records where the value of the "POP_1990" column is greater than or equal to 5 million but less than or equal to 10 million:

```
IFilter filter = new BetweenFilter("POP_1990",
  new AllType(5000000), new AllType(10000000));
```

InList Filter

Compares the values in a table to a list of possible specified values.

For example, to return all the records where the value of the "State_Name" column is either "New York", "California", or "Florida":

```
IFilter filter = new InListFilter("State_Name",
  new AllType("New York"),
  new AllType("California"),
  new AllType("Florida"));
```

IsNull Filter

Inspects the values in a table to determine whether they are null. Note that not all data providers support the concept of a null value for all data types.

For example, to return all the records where the value of the "Nickname" column is null:

```
IFilter filter = new IsNullFilter("Nickname");
```

Key Filter

Compares the primary key values in a table to any of the specified values. Note that the values depend on how the primary key is defined by the table. It is very important to understand that there is no true primary key for a native TAB file. The row ID is used in this manner, so it is considered an implicit primary key. However, if you carry out an insert or update operation on a TAB file (outside of Location Intelligence SDK), the row ID values could change.

For example, to return the records where the primary key is 1, 5, or 7:

```
IFilter filter = new KeyFilter("1", "5", "7");
```

Like Filter

Compares the string values in a table with the specified matching pattern. The wildcard characters are '_' to match any single character, '%' to match zero or more characters, and '\' to represent an escape.

For example, to return the records where the value for the "State_Name" column starts with "New":

```
IFilter filter = new LikeFilter("State_Name",
  new AllType("New%"));
```

Distance Filter

Compares the distance between the geometry in a table with the specified geometry. For more information on creating geometries, see **Geometry API** on page 6.

For example, to return the records where the geometry in the "Obj" column that are within 500 miles of the specified point:

```
GeometryFactory factory =
  GeometryFactory.forCoordSys(CoordSysConstants.longLatWGS84);

IFeatureGeometry geom = factory.newPoint(-72, 42);
IFilter filter = new DistanceFilter(
  "Obj",
  geom,
  DistanceFilterType.INTERSECTS,
  new Length(500, LinearUnit.MILE));
```

Envelope Filter

The enveloper filter returns all geometries that intersect with an envelope. Envelope filters are mainly used as a pre-filter to narrow down the searching, as it maximizes performance due to the simplicity and speed of the envelope geometry searching.

For example, to return the records where the geometry in the "Obj" column intersects the specified envelope:

```
IFilter filter = new EnvelopeFilter(
  "Obj",
  new Envelope(-72, 42, -73, 41,
    SpatialInfo.create(CoordSysConstants.longLatWGS84)));
```

Geometry Filter

Compares the geometry in a table with its spatial relation to the specified geometry. See the JavaDocs for a full list of available methods in the `GeometryFilter` class.

For example, to return the records where the geometry in the "Obj" column contains the specified point:

```
GeometryFactory factory =
  GeometryFactory.forCoordSys(CoordSysConstants.longLatWGS84);

IFeatureGeometry geom = factory.newPoint(-72, 42);
IFilter filter = GeometryFilter.withinFilter("Obj", geom);
```

Or, return the records where the geometry in the "Obj" column intersects the specified linestring:

```
GeometryFactory factory =
  GeometryFactory.forCoordSys(CoordSysConstants.longLatWGS84);

IFeatureGeometry multiCurve = factory.newMultiCurve()
  .withLineString(pos(-72, 42), pos(-73, 41)).build();
IFilter filter = GeometryFilter.intersectsFilter("Obj", multiCurve);
```

Logical Filter

Combines multiple filters based on a logical operator.

For example, to return all the records where the value of the "Nickname" column is not null:

```
IFilter filter = LogicalFilter.notFilter(new IsNullFilter("Nickname"));
```

Or, return the records where the value of the "State_Name" column is "New York" and the value of the "POP_1990" column is greater than 5 million:

```
IFilter filter = LogicalFilter.andFilter(
  AttributeFilter.equalsFilter("State_Name", new AllType("New York")),

  AttributeFilter.greaterThanFilter("POP_1990", new AllType(5000000)));
```

Or, return the records where the value of the "State_Name" column is "New York" or the value of the "POP_1990" column is less than 1 million:

```
IFilter filter = LogicalFilter.orFilter(
  AttributeFilter.equalsFilter("State_Name", new AllType("New York")),

   AttributeFilter.lessThanFilter("POP_1990", new AllType(1000000)));
```

Nearest Filter

Compares the geometries in a table with another geometry object for the distance between them.

For example, to return all the records where the geometries in a table are closest from a point with longitude and latitude values (-72, 42) and limiting the search distance to 100 miles:

```
GeometryFactory factory =
  GeometryFactory.forCoordSys(CoordSysConstants.longLatWGS84);

IFeatureGeometry seedPoint = factory.newPoint(-72, 42);
IFilter filter = NearestFilter.newBuilder("Obj", seedPoint)
  .maxDistance(new Length(100, LinearUnit.MILE))
  .build();
```

DistanceToEdge Filter

Compares the geometries in a table with a point geometry to determine the distance between the point and the edge of the geometries,

For example, to return all the records with geometries whose edge falls within 100 meters from a point with longitude and latitude values (-72, 42):

```
GeometryFactory factory =
  GeometryFactory.forCoordSys(CoordSysConstants.longLatWGS84);
```

```
IPoint seedPoint = factory.newPoint(-72, 42);
IFilter filter = DistanceToEdgeFilter.newBuilder(
    "Obj", seedPoint, "DTE", LinearUnit.METER, "IS_WITHIN")
  .maxDistance(new Length(100, LinearUnit.METER))
  .build();
```

5 - Tile Grid API

In this section

Tile Grids

Map tiles are a widely used way of representing spatial data, typically for the purpose of efficiently transferring maps over a network connection and displaying them in a graphical user interface.

Tile grids are the bounds of a map divided into smaller and smaller rectangles called tiles. These tiles can either be raster-based and rendered as images, or vector-based where the tile contents are geometries and attribute data. A client application can request a range of tiles and render them at a certain resolution. Tiles can be rendered on the fly, but more commonly they are pre-generated and cached for performance reasons.

The Location Intelligence API supports this process by providing the `ITileGrid` abstraction. A tile grid is based on a particular coordinate system and covers a defined bounding box. Operations offered by the tile grid class include getting the geographic bounds of a map tile, retrieving the range of tiles that covers a given bounding box, as well as various operations to transform between geographic coordinates, screen coordinates and tile coordinates. Common applications that consume this API are tile web services and applications to pre-cached tiles.

Creating a Tile Grid

A tile grid is generated by using the `TileGridBuilder` class, for example:

```
ITileGrid grid = TileGridBuilder.webMercatorGrid().build();
```

The builder enables the customization of various tile grid aspects, but all of these aspects fall back to sensible default values, in order to keep the API simple. The following code snippet creates a grid with zero-based indexes, instead of the default 1-based indexes.

```
ITileGrid grid =
  TileGridBuilder.webMercatorGrid().zeroBased().build();
```

To get the bounds of a tile at a given row, column and zoom level, use:

```
int row = 12, col = 32, level = 10;
Envelope tileBounds =
  grid.newTileCoordinate(row, col, level).getBounds();
```

To get the range of tiles that cover a given bounding box, use the following. An `ITileCoordinateRange` implements `Iterable` and can therefore be used in common Java loop constructs.

```
int level = 10;
Envelope bounds = new Envelope(-10000, -20000, 30000, 40000,
   grid.getBounds().getSpatialInfo());
```

```
ITileCoordinateRange range = grid.tileRangeForBounds(bounds, level);
for (ITileCoordinate coordinate : range) {
 //...
}
```

Vector Tiles

Vector tiles are a similar to raster tiles but where as raster tiles are rendered as images, vector tiles contain the geometries and metadata for each tile. The geometries and feature data are converted into a structured and compact format which reduces duplicate values and reduces the amount of time the client needs to download each vector tile.

Vector tiles created with the Location Intelligence SDK follow the MapBox specification (**https://www.mapbox.com/vector-tiles/specification**). They can be rendered out of the box with OpenLayers, or in Leaflet provided a plug-in such as Leaflet.VectorGrid is used. For a look at how vector tiles are used in OpenLayers and Leaflet, see the samples provided in the SDK.

Using the Vector Tile Writer

`IVectorTileWriter` is used to construct map tiles containing vector features. Each tile can have one or more named layers and each layer can have one or more vector features. `IVectorTileWriter` has three states: initialization, tile generation and layer generation. The initialization state is when `IVectorTileWriter` is created and multiple operations can be set on how the writer handles each feature added to each tile.

Create a `FilterSearch` with the bounds for an `ITileGrid` using row, column and level.

```
  ITileGrid grid = TileGridBuilder.webMercatorGrid().zeroBased().
    withTileSize(width, height).build();
  IDataSourceTable table =
    NativeDataProvider.getInstance().openTable(tableName);
  Envelope mbr =
    grid.newTileCoordinate(row, column, level).getBounds();

  // In some cases, the styling used by the geometry could bleed into
this tile even though the actual
  // geometry is outside the MBR, to compensate for this, we are using
a buffer, defined by the QUERY_CLIP_BUFFER,
  // 2% in this example, when querying data.
  double envelopeBuffer = mbr.getExtentX() * 0.02;
  Envelope queryMBR = mbr.getCopy();
  queryMBR.expand(envelopeBuffer, envelopeBuffer);
```

```
FilterSearch search = FilterSearch.searchALL();
search.setFilter(new EnvelopeFilter(geometryColumnName, queryMBR));
```

Create an `IVectorTileWriter` using the `VectorTileWriterBuilder` with clipping, removal of points and transformation of the coordinate system.

```
IVectorTileWriter writer = VectorTileWriterBuilder.newMVTBuilder()
  .withClipping(0.02)
  .withRemovalOfDuplicatePoints()
  .withRemovalOfDegenerateGeometries()
  .withTransformation(CoordSysFactory.getDefaultCoordSysFactory()
    .getCoordSys("epsg", "3857"), extent)
  .build();
```

Before any features can be added to the writer, a new tile needs to be created with the tile bounds, followed by a new layer.

```
writer.startTile(mbr);
writer.startLayer("SampleLayerName");
```

Iterate over all the features from the filter search and copy the geometry and attributes into the builder as a new vector feature.

```
IFeatureIterator iterator = table.search(search);
try {
 while (iterator.hasNext()) {
  IFeature feature = iterator.next();
  IFeatureGeometry geometry =
    feature.getFeatureGeometry(geometryColumnName);
  Map<String, Object> attributes = new HashMap<>();
  attributes.put("State_name", feature.getString("State_name"));
  writer.addFeature(
    key++,
    geometry,
    attributes);
 }
}
finally {
 iterator.dispose();
}
```

When `endTile` is called, the final tile is created and returned as a byte array. This byte array has the structure according to the MVT specification.

```
writer.endLayer();
byte[] tile = writer.endTile();
```

6 - Grid Index API

In this section

Grids

A grid is a way of dividing the surface of the earth into contiguous cells with no gaps in between.

Hashing is a technique for encoding and decoding cells in the grid using the cell boundary and a unique identifier. This makes grids very useful for spatial indexing and aggregating. The hash value identifies the cell as a unique place on earth. The string can vary in length to represent the precision of the cell in the grid.

Precision is the length of the string key to be returned. The precision determines how large the grid cells are (longer strings means higher precision and smaller grid cells). It returns the string ID of the grid cell at the specified precision that contains the point.

The Location Intelligence SDK API supports hashes for three grid cell shapes: rectangular, square and hexagon. Square cells are often uses over rectangles when the grid is displayed in the Popular Mercator projection. Hexagons are often used in telecommunication solutions as they approximate circles while covering the surface of the earth without gaps.

For more information on the grid API, see the Javadocs.

Examples

Gets the hash value for a hexagon given a location and precision. The location is the center of the hexagon. Encodes the latitude/longitude with a hash value.

```
  String hash = Hexagon.encode(-73.700052, 42.678120, 10).getHash();
```

Gets the geometry for a hexagon cell with a given hash value

```
  IFeatureGeometry geometry =
Hexagon.decode("PF7041126164").asFeatureGeometry();
```

Gets the geohash value for a rectangle cell at a given location and precision.

```
  String hash = GeoHash.encode(-73.700052, 42.678120, 10).getHash();
```

Gets the bounds of a rectangular grid cell for a given hash value.

```
  Envelope bounds = GeoHash.decode("dredfcm606").getBounds();
```

Gets the hash for a square cell given a location and precision.

```
  String hash = Square.encode(-73.700052, 42.678120, 10).getHash();
```

Gets the bounds of a square grid cell for a given hash value.

```
  Envelope bounds = Square.decode("0302323112").getBounds();
```

7 - Samples

In this section

Point In Polygon Sample

Using an example is a good way to get started using the Location Intelligence SDK. This is a walk through of a point in polygon operation on a dataset containing US states. It has been broken down into various sections to show some of the core concepts and operations used in the Location Intelligence SDK such as connecting to your data, using geometries, performing operations, and filtering results. Use the walk-through point in polygon example below and follow the links for the various detailed sections on how to use the Location Intelligence SDK.

For a complete working example, see the `PointInPolygonExamples.java` file in the samples included with the Location Intelligence SDK.

> **Note:** For descriptions and functionality of the SDK, see the JavaDocs included in your deployment.

This example uses a point geometry created using the geometry factory, and then determines if that point is within the geometries located in a TAB file, limiting the results using an envelope filter, and only returning a select list of attributes (column data).

First, get the TAB file that contains all of the polygon data, and open the TAB file. When using TAB files, you are using the native data provider. For more information on using TAB files and the native data provider, see **MapInfo TAB**.

```
  try (IDataSourceTable table =
    NativeDataProvider.getInstance().openTable(path)) {
   // ...
  }
```

> **Note:** When referencing the table directory, it may be relative to the process current working directory.

Once the TAB file is open, you can get all of the table attributes information. This information can be used later to request a particular attribute (column).

```
  for (IAttributeDefinition attributeDefinition :
    table.getMetadata().getAttributeDefinitions()) {
   //System.out.println("Attribute name: " +
attributeDefinition.getName());
  }
```

The attributes allow you to specify the list of columns to return in the search. In this example, since the point in polygon is doing a search on a states table, it can return the state name, state, and the state geometry (Obj column).

```
  SelectList selectList = new SelectList("State_name", "State", "Obj");
```

Next, create the envelope geometry to be used in the envelope filter. To create an envelope, you must define the coordinate system, and then the geometry. For more information on creating geometries, see **Geometry API** on page 6.

```
SpatialInfo wgs84SpatialInfo =
  SpatialInfo.create(CoordSysConstants.longLatWGS84);

Envelope searchingEnvelope = new Envelope(
  pos(-90.593531, 27.968251), pos(-68.088187, 49.483763),
  wgs84SpatialInfo);
```

Once the envelope is created, you can use that to create the filter. In this case, an envelope filter. The envelope filter is limiting the search area of the table to a given area from the geometry column in the table (the OBJ column is the geometry column in TAB files). For more information on filters, see **Using Filters** on page 20.

```
EnvelopeFilter envelopeFilter =
  new EnvelopeFilter("Obj", searchingEnvelope);
FilterSearch filterSearch = new FilterSearch(selectList);
filterSearch.setFilter(envelopeFilter);
```

Apply the envelope filter and do the search, returning a cursor that contains the list of features that meets the criteria of the filter (the states within the envelope).

```
ICursor cursor = table.search(filterSearch);
```

Next, create the point that will be used for the within operation. If the point has a different coordinate system as the data in the table, a transformation will be done as part of the search. To increase performance, it is advised that both the searching geometries and the source geometries are in the same coordinate system.

```
GeometryFactory factory =
  GeometryFactory.forSpatialInfo(wgs84SpatialInfo);

IPoint pointWGS84 = factory.newPoint(-74.0059, 40.7128);
```

Iterate through all of the features returned by the filter search and perform the point in polygon (using the point you created) against the returned feature geometries (states) using the within geometry operation. For more information on geometry operations, see **Geometry Operations** on page 8.

```
try {
 while (cursor.hasNext()) {
  //Get a feature
  IFeature feature = cursor.next();

  //Get the geometry column
 IFeatureGeometry featureGeometry = feature.getFeatureGeometry("Obj");


  //Perform PIP operation using GeometryOperations.within
```

```
    if (GeometryOperations.within(pointWGS84, featureGeometry)) {
     System.out.println("Point: " + pointWGS84.toString()
       + " is WITHIN " + feature.getString("State") + " State.");
    }
```

When using data providers, the cursor may hold on to resources (i.e., file locks, file readers, etc.) It is important to always dispose of the cursor in order to release these resources.

```
  cursor.dispose();
```

Vector Tiles Sample

The Location Intelligence SDK includes a sample application that demonstrates how the vector tiles can be served over the web. The sample is bundled as an executable binary jar file and the source code is also included alongside the binary.

The source comprises a minimalist `SpringBootApplication` called application and a REST service called the `VectorTileService` under the `/samples/src/com/pb/lisdk/sample/vectortile` folder inside the package. It also contains the client applications using OpenLayers and Leaflet within the `/samples/src/static` folder inside the package

To launch the example, unzip the package and open a command prompt. Change to the `/samples/bin` folder inside the root where the package was unzipped. You should see a single jar file at this location called `li-sdk-samples-XXXX.jar` where `XXXX` represents a version number. You can launch the application by entering the command `java -jar li-sdk-samples-XXXX.jar`.

> **Note:** The web application, by default listens at port 8080. To change the port number use a switch on the command line: `--server.port=NNNN`. For example, `java -jar li-sdk-samples-XXXX.jar --server.port=8888` will launch the service and listen on port 8888.

Open a browser and navigate to `http://localhost:8080` and browse the two client applications to see the vector tile service in action. If the browser is opened on a different machine to the one where the web application was started in the step above, substitute localhost with appropriate host name

Copyright