The Iceberg Java and Python APIs

This chapter is a hands-on exploration of utilizing Apache Iceberg with Java and Python, providing practical insights into table management, schema, and datafile operations. It covers aspects such as creating tables, schemas, and partition specs, as well as reading and updating data. The chapter aims to equip readers with the necessary skills to efficiently interact with Apache Iceberg using these APIs.

The Java API

The Java API provides a programmatic way to manage Iceberg table metadata and data from a custom Java application, such as the schema, partition spec, and datafiles. We will navigate through such operations as creating tables and schemas, partitioning data, and reading and updating data in this part of the chapter so that you can quickly get started with this API.

Create a Table

As with any other compute engine, the first step to creating and interacting with Apache Iceberg tables is to create a catalog. In general, with the Java API, tables are created using either a `Catalog` or a `Table` interface implementation. In this chapter, you will see how to use two of the built-in catalogs to create Iceberg tables. However, you can also implement and use a custom catalog of your own.

Initializing a Hive catalog involves providing a name and catalog-specific properties. Here is an example of creating an Iceberg table using the Hive catalog:

```
import org.apache.iceberg.hive.HiveCatalog;
import org.apache.iceberg.Table;
import org.apache.iceberg.catalog.TableIdentifier;

HiveCatalog catalog = new HiveCatalog();
catalog.setConf(spark.sparkContext().hadoopConfiguration());
```

```
Map <String, String> properties = new HashMap<String, String>();
properties.put("warehouse", "...");
properties.put("uri", "...");

catalog.initialize("hive", properties);

TableIdentifier name = TableIdentifier.of("hr", "employee");
Table table = catalog.createTable(name, schema, spec);
```

The preceding code configures a Hive catalog that interacts with the Hive Metastore to track Iceberg tables. It sets the required Hive configurations and initializes the catalog with properties such as `warehouse` and `URI`. Then it uses the `createTable()` method to create a new Iceberg table named `employee` in the `hr` namespace with a defined schema and partitioning spec.

A Hadoop catalog can be used with filesystems that support atomic renaming, such as the Hadoop Distributed File System (HDFS). Here is how to create a table with a Hadoop catalog:

```
import org.apache.hadoop.conf.Configuration;
import org.apache.iceberg.hadoop.HadoopCatalog;
import org.apache.iceberg.Table;
import org.apache.iceberg.catalog.TableIdentifier;

Configuration conf = new Configuration();
String warehouseLoc = "hdfs://host:8020/path";
HadoopCatalog catalog = new HadoopCatalog(conf, warehouseLoc);

TableIdentifier name = TableIdentifier.of("hr", "employee");
Table table = catalog.createTable(name, schema, spec);
```

Create a Schema

To create a table schema, you will need to create an object of the `Schema` class in Iceberg. Here is an example of defining a schema:

```
import org.apache.iceberg.Schema;
import org.apache.iceberg.types.Types;

Schema schema = new Schema(
  Types.NestedField.required(1, "id", Types.LongType.get()),
  Types.NestedField.required(2, "department", Types.StringType.get()),
  Types.NestedField.required(3, "role", Types.StringType.get()),
  Types.NestedField.required(4, "salary", Types.LongType.get()),
  Types.NestedField.required(5, "region", Types.StringType.get())
);
```

The preceding code creates a schema for the table with five fields.

Create a Partitioning Spec

A partition specification in the Java API is defined using a builder pattern. In Iceberg, `PartitionSpec.builderFor(schema)` starts the building process for a partition specification. This is based on the table's schema. Then, through the builder, you specify how you want to partition the data. Here is an example:

```
import org.apache.iceberg.PartitionSpec;

PartitionSpec spec = PartitionSpec.builderFor(schema)
      .day("hire_date")
      .build();
```

Here, we use `.day("hire_date")` to define the partitioning strategy for our table. This means Iceberg will create a separate partition for each distinct day.

Read a Table

The `Table` interface in the Java API provides methods to interact with an Iceberg table to get information on things such as the schema, partition spec, metadata, and datafiles. Here are a few common methods:

`schema()`
: Returns the current table schema

`spec()`
: Provides the current table partition spec

`currentSnapshot()`
: Retrieves the current snapshot of the table

`location()`
: Provides the base location of the table

File-level scan

Reading from an Iceberg table begins with creating a `TableScan` object using the `newScan()` method. An Iceberg `TableScan` returns a list of tasks (datafiles, delete files, etc.) that are related to the desired scan. This is useful if you want your compute engine to support Apache Iceberg, as it can use a table scan to generate the list of files for the query engine to scan and process without having to reimplement the process of reading the Iceberg metadata. Here is an example:

```
TableScan scan = table.newScan();
```

You can also use the `filter()` method on the `TableScan` object to filter specific data:

```
TableScan filteredScan = scan.filter(Expressions.equal("role", "Engineer"));
```

The preceding code will read the table metadata and return a list of any files in the current snapshot what would be applicable to a scan for records where `role = Engineer`.

To retrieve specific data, the API provides a `select()` method to be used on the `TableScan` object, as shown in the following code:

```
TableScan scan = table.newScan()
    .filter(Expressions.equal("role", "Engineer"))
    .select("id", "salary");
```

Row-level scan

With the Java API, initiating a row-level scan that returns individual records of data begins by creating a `ScanBuilder` object with `IcebergGenerics.read()`. This may not be practical for very large datasets, which should use a query engine. Those query engines will likely use the `TableScan` construct previously discussed to help with their queries. Here is an example:

```
ScanBuilder scanBuilder = IcebergGenerics.read(table);
```

You can then use the `where()` method on the `ScanBuilder` object to filter out records:

```
CloseableIterable<Record> result = IcebergGenerics.read(table)
        .where(Expressions.lessThan("id", 5))
        .build();
```

In the preceding code, an expression, `Expressions.lessThan("id", 5)`, is used to determine whether a row should be included in the result.

Note that `ScanBuilder()` provides a more direct way to pull rows from an Iceberg table into your Java application by applying filters such as the one demonstrated in the example, whereas `TableScan()` allows you to generate a list of tasks/files to be scanned and passed to the computing engine of choice.

Update a Table

Apache Iceberg's `Table` interface exposes methods that allow updates to the table. These operations follow a builder pattern. For example, to update a table's schema, you have to first call `updateSchema()`, then add the updates to the builder, and finally call `commit()` to push the changes to the table. An example follows:

```
table.updateSchema()
    .addColumn("location", Types.StringType.get())
    .commit();
```

Here, we add a new column called `location` of type `String` to the Iceberg table using the `addColumn()` method. Other available methods include `updateLocation`, `updateProperties`, `newAppend`, `newOverwrite`, and `rewriteManifests`.

Create a Transaction

Transactions in Iceberg support committing multiple changes to a table atomically. Individual operations within a transaction are created using factory methods, and all these operations are collectively committed using the `commitTransaction()` method.

For instance, in the following code, we first overwrite some of the files based on a filter expression and then replace them with a new datafile, as well as update the location of the table with `updateLocation()` within a single transaction. In the end, we commit the operations atomically:

```
Transaction t = table.newTransaction();

// Overwrite files
t.newOverwrite()
    .overwriteByRowFilter(Expressions.equal("id", 5))
    .addFile(newDataFile)
    .commit();

// Update the table location
t.updateLocation()
    .setLocation("new-location-path")
    .commit();

// Commit all the changes to the table
t.commitTransaction();
```

The Python API

The Python API for Apache Iceberg provides a way to interact with Iceberg tables from a Python application without leveraging an engine (Rust and Go libraries in early stages at the time of this writing). It also allows for working with Apache Iceberg table catalogs and metadata for operations such as creating tables and planning scans. In this section, we will walk through a couple of hands-on exercises to see how to configure catalogs, load a table, query data, and create a table.

Install PyIceberg

The latest version of PyIceberg can be installed using PyPi:

```
pip3 install "pyiceberg[s3fs,hive]"
```

The preceding command will install PyIceberg with s3fs as a `FileIO` implementation and support for the Hive Metastore. Note that you can specify various dependencies to be installed with PyIceberg within the brackets, [ ], including AWS Glue, Amazon Simple Storage Service (Amazon S3) readers, PyArrow, and more. Also, you will need to install either s3fs, adlfs, or PyArrow to access files.

Configure the Catalog

Configuring the catalog is the first step to using the API for Iceberg tables. PyIceberg currently supports AWS Glue, Hive, and REST catalogs natively. There are a couple of ways to configure a catalog: using a *~/.pyiceberg.yaml* file, using environment variables, or passing the datalog configuration directly via the CLI or Python code.

Let's take a look at how you can configure different catalogs using PyIceberg. We will use the *pyiceberg.yaml* config file to define these configurations. Make sure you place this file in your operating system's home directory, the location of which may differ depending on your operating system and environment variable settings.

The Hive catalog

The Hive catalog configuration requires the URI for the Hive Metastore and the storage system details. For example, if you are using Amazon S3 as the storage, you will need to provide the S3-specific configurations (endpoint, access key ID, and secret access key) to perform actions on S3. Here is how the config file may look:

```
catalog:
  default:
    uri: thrift://localhost:9083
    s3:
      endpoint: http://localhost:9000
      access-key-id: admin
      secret-access-key: password
```

The Glue catalog

To use Glue as a catalog, the AWS credentials must be either passed directly through the Python API or set up locally. Here is an example config:

```
catalog:
  default:
    type: glue
    aws_access_key_id: <ACCESS_KEY_ID>
    aws_secret_access_key: <SECRET_ACCESS_KEY>
    aws_session_token: <SESSION_TOKEN>
    region_name: <REGION_NAME>
```

The REST catalog

The REST catalog configuration involves specifying the URI that identifies the REST server and the credentials needed to authenticate to the service. Here is a sample config:

```
catalog:
  default:
    uri: http://rest-catalog/ws/
    credential: t-1234:secret
```

Additional parameters that can be used with the REST catalog config include the bearer token for the `Authorization` header, and `rest.sigv4-enabled` to indicate whether to use AWS Signature v4 (SigV4) to authenticate requests.

Load a Table

You can load a table with PyIceberg either from a catalog or directly from a metadata file. Let's see how to do this in action.

From a catalog

To load a table with PyIceberg from a catalog, start by connecting to a catalog. The following code connects to a Glue catalog:

```
from pyiceberg.catalog import load_catalog

catalog = load_catalog("glue_catalog")
catalog.list_namespaces()
```

This will return the available namespaces. In this example case, it returns the namespace `company`.

Now, to list all the tables in the `company` namespace, you can call the `list_tables()` method:

```
catalog.list_tables("company")
```

This will return all the tables that exist within the namespace as a list with tuples—for example, `[("company", "employees")]`.

Finally, to load the `employees` table, you can use the `load_table()` method:

```
catalog.load_table("company.employees")

# Alternatively:
catalog.load_table(("company", "employees"))
```

From the metadata file

Loading a table directly from the metadata file can be beneficial for situations where you would like to bypass the catalog that can be used for one-off read-only transactions. Any updates to the table should always be done via the catalog to maintain consistency. This is facilitated by the `StaticTable` class. Here is an example:

```
from pyiceberg.table import StaticTable

ice_table = StaticTable.from_metadata(    "s3a://my-bucket/test.db/salesnew/
metadata/00001-91773d96-d1f4-490e-a894-9bc0652b6500.metadata.json"
)
```

The method `from_metadata()` takes the location of the metadata file from your storage and loads it as a table.

Create a Table

To create a table with PyIceberg, you need to define the schema, any partition specification or sort order, and the catalog from which you're creating the table. The `create_table()` method then takes these arguments and creates the table. Here is an example that creates an `employee` table with five fields:

```
from pyiceberg.catalog import load_catalog
from pyiceberg.schema import Schema
from pyiceberg.types import DoubleType, StringType, NestedField

# Define the schema
schema = Schema(
    NestedField(required=False, field_id=1, name="id", field_type=DoubleType()),
    NestedField(required=False, field_id=2, name="role", field_type=String
Type()),
    NestedField(required=False, field_id=3, name="salary", field_type=Double
Type()),
    NestedField(required=False, field_id=3, name="department", field_type=String
Type()),
    NestedField(required=False, field_id=3, name="region", field_type=String
Type())
)

# Load the catalog
catalog = load_catalog("glue_catalog")

# Create the table
catalog.create_table(
    identifier="company.employee", location="/Users/user/Desktop/docker-spark-
iceberg/wh/employees/", schema=schema
)
```

Update Table Properties

PyIceberg has limited support for write operations. However, you can update table properties through the Transaction API. Let's look at an example:

```
# setting a property
with table.transaction() as transaction:
    transaction.set_properties(prop="value")
assert table.properties == {"prop": "value"}
```

This code block starts a transaction on a table, sets a property, and then checks whether the property has been set correctly. If the process fails, the entire transaction is rolled back, and the table will not be modified with the property.

Query Data

You will need to do a table scan to query a table using PyIceberg, which will return a list of files similar to the `tableScan` class in the Java API. There are several arguments that you can pass to the `table_scan()` method, including filters, limits, and snapshot IDs. Here is an example that shows how to load the `employees` table from `glue_catalog` with a filter for employees with a salary greater than or equal to $50,000:

```
from pyiceberg.catalog import load_catalog
from pyiceberg.expressions import GreaterThanOrEqual

catalog = load_catalog("glue_catalog")
table = catalog.load_table("employee")

scan = table.scan(
    row_filter=GreaterThanOrEqual("salary", 50000.0),
    selected_fields=("id", "role", "salary"),
    limit=100
)
```

After you perform a table scan, you must pass the list of tasks/files to something that can process and read the list of files. PyIceberg has integrations to pass the scan plan to PyArrow, DuckDB, and Ray to query and process the data.

Conclusion

This chapter covered the Java and Python APIs for working with Apache Iceberg tables. These native table APIs allow you to create tables, create scan plans for compute engines, create transactions, and more. New native table APIs in more languages, such as Rust and GO, are being worked on, expanding the reach of Apache Iceberg across different languages and tools created in those languages.