Surfacing Iceberg Table management capabilities in Spark SQL

Harish Butani

Iceberg [1], [2], [3], [4] introduces the concept of *Table formats* (as opposed to File Formats) that defines an access and management model for Tables in Big Data systems. At its core it is a well documented and portable specification of versionable Table metadata(both physical and logical metadata). On top of this it provides a set of capabilities: Snapshot isolation, significant speed and simplicity in access of Table metadata critical for Query Planning overhead(even in the case of datasets with millions of files and partitions), schema evolution, and partition layout isolation(hence the ability to change physical layout without changing Applications).

It defines clear contracts for underlying file formats used for data files, such has how schema and statistics are integrated from these formats. It also prescribes how Iceberg capabilities can be integrated into existing Big Data scenarios through various packaged components such as `iceberg-parquet`, `iceberg-orc`, `iceberg-avro` for applications that directly manage `parquet, orc and avro` files. `iceberg-hive`, `iceberg-presto` and `iceberg-spark` are packaged jars that can be dropped into scenarios using `hive, presto and spark` and leverage Iceberg to manage datasets. In Appendix A we list examples of Iceberg usage taken from the Iceberg code base for Hive, Parquet and Spark.

These capabilities fill some very critical gaps of Table management in Big Data systems, and hence various open source communities have quickly adopted/integrated Iceberg functionality. Iceberg was initially developed at Netflix; subsequently(likely because of its wide appeal) Netflix has graciously incubated it as an Apache project [3].

For Apache Spark, Iceberg integration is not fully available for the SQL layer. There is work going on to surface Iceberg Table Management as a V2 Datasource table [5], but V2 Datasources itself are not fully integrated into Spark SQL [6], [7]; besides we feel it is useful to provide the Table Management for Datasource V1 tables, bringing this functionality to a large deployed base.

Apache Spark is gaining traction as a platform for Enterprise Analytical workloads: for example our Oracle SNAP Spark native OLAP Platform [8], [9] is used by a Fortune 10 company to power their Finance Data Lake. These tend to be SQL heavy(in fact almost exclusively SQL based) solutions. It is a time honored tradition to surface analytical and management functionality in SQL, for example as SQL Row and Table functions, or Database options like OLAP and Geospatial capabilities. Data Management is a critical aspect of an Analytical platform, but unfortunately is an underdeveloped component of Apache Spark. This has led customers to come up with their own data management schemes such as using Hive ACID Tables for data management with

Spark for query processing, or custom solutions using ETL platforms and tools such as Talend and Airflow. Providing Table management that is seamlessly integrated into familiar SQL verbs such as `create table`, `insert`, or `select` will simplify the task of developing Analytical solutions on Apache Spark and will drive further adoption.

These reasons led us to develop the ability to use Iceberg Table Management capabilities with Spark SQL, specifically Datasource V1 tables. Our component will:

- allow users to `create` managed tables and define source column to partition column transformations as table options.
- have SQL `insert` statements create new Iceberg Table snapshots
- have SQL `select` statements leverage Iceberg Table snapshots for partition and file pruning
- provide a new `as of` clause to the sql `select` statement to run a query against a particular snapshot of a managed table.
- extend Spark SQL with Iceberg management views and statements to view and manage the snapshots of a managed table.

Brief Summary of Iceberg

In this section we provide a summary of the **Table Metadata** structure of Iceberg and how this structure is maintained under change and used during Table Scan. The user is encouraged to refer to the Iceberg Specification [1] for details about these topics. We

Iceberg Table Metadata Structures

Table Metadata has the following core information: `current table schema`, `current partition specification`, the **current Snapshot**, a list of historical Snapshots, a change log. Table Metadata is maintained in a json file using a MVCC scheme: any changes to metadata triggers a new Metadata version, writer transactions retry if the Table has changed since they started, but readers are not impacted by writers. **Snapshots** are made of **manifest** files which capture information about a set of **data files**. The set of all *data files* in a Table are split across multiple manifest files so as to enable fast appends(just add a new manifest file the new files instead of rebuilding an entire manifest), Tables with multiple partitioning schemes will have different manifest files for each scheme, and for Tables with large number of files and/or partitions splitting into multiple manifest files will enable parallel

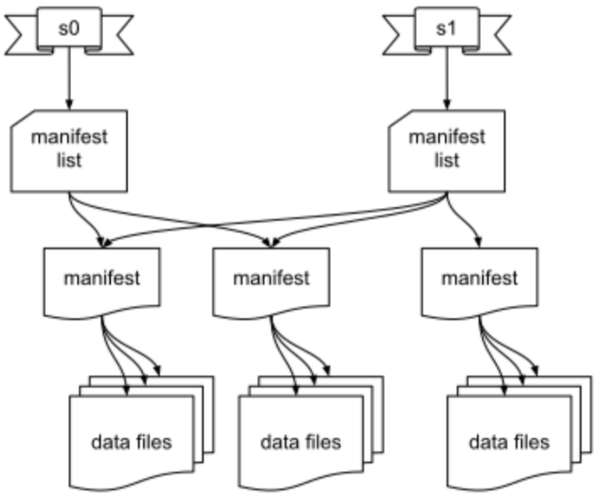

Figure 1: Iceberg Table Metadata

Changing a Table's data

Figure 2 shows a conceptual view of what happens during a Table insert or overwrite. The Command is executed by a set of `Write Tasks`. For example in the case of **Spark Datasource V2**: the execution of the `WriteToDataSourceV2Exec` command runs a job with a task for each partition of the input rdd, each task writes a new file whose information is sent to the driver in the `TaskCommitMessage`, the `DataSourceWriter` associated with the Iceberg table is passed the `TaskCommitMessages` on job commit, finally in the `commit` method the `DataSourceWriter` sets up a new `Iceberg Transaction`, applies the Table changes and commits the new Table Metadata version.

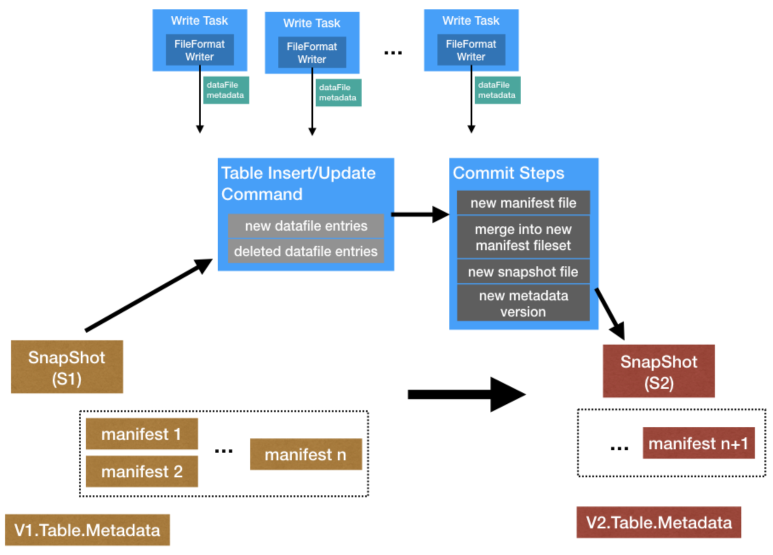

Figure 2: Iceberg Table Update Operation

Scanning a Table

Figure 3 shows a conceptual view of the role Iceberg Table Metadata plays in data reads.

Query projection and filters the Table scan is defined by the columns projected and the filters(on partition or data columns) defined. A Table Scan Command accepts these. For example in case of **Spark Datasource V2** there is a custom `Reader` class for parquet file format that implements `Supports ScanUnsafeRow`, `Supports PushDownCatalystFilters`, and `Supports PushDownRequiredColumns`.

Transformation and application on projections and filters the Query filters and projections are converted to Iceberg filter `Expressions` and `column names`. Then an Iceberg `TableScan` is created for the latest snapshot and the Iceberg filters and projectList is applied to the scan.

Build Scan Tasks next the `TableScan` is asked for the effective set of Scan Tasks. This is one per `DataFile` that remains after the application of partition filters, data filters and column projections.

Hoist Iceberg Scan Tasks into Big Data system the Iceberg Scan Task definitions are then used to create and execute Read Tasks in the Big Data system. For example **Spark Datasource V2** a parquet `ReadTask` that implements `DataReaderFactory<UnsafeRow>` uses the Iceberg Scan Task definitions to open and iterate the parquet DataFiles. These scans apply remaining filter and projections as spark expressions.

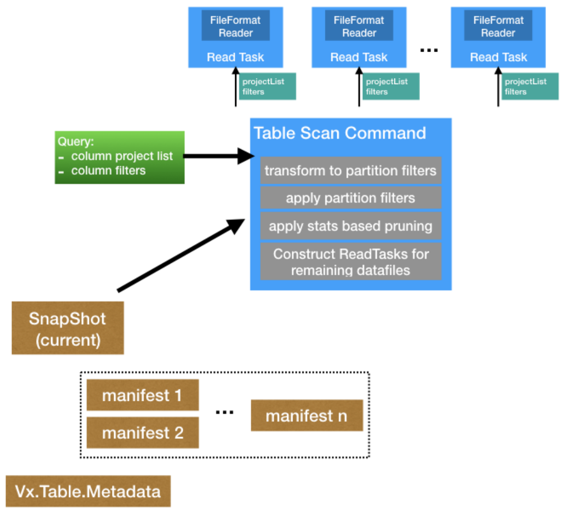

Figure 3: Iceberg Table Scan Operation

Changing a Table's metadata

Iceberg `Transaction` provides tasks for an `UpdateProperties` and `ExpireSnapshots`. These can be invoked directly by working with the Iceberg API: for example by constructing a `HadoopTables`, getting a `Table` object and performing a `Transaction` on the Table.

Spark SQL Integration

Create Statement

We will support Datasource V1 tables to be managed with Iceberg. As of the writing of this document we support management of partitioned, non bucketed tables. We plan to extend support to non-partitioned and bucketed tables shortly. When creating a Datasource V1 table the user must add an `addTableManagement=true` option to the table DDL.

For tables with `addTableManagement=true` an `partitionValueMappings` option can be used to specify a mappings between non-partition column values and partition column values. These mappings will be used to convert predicates on non-partition columns into Iceberg predicates. The value mapping should be `1-1` or `n-1`. Currently we support **Iceberg Transforms**, so users can relate non-partition columns based on `date` or `timestamp` elements, based on `truncating` values or `value buckets`. The **transforms** must be specified as a comma-separated list of *key-value pairs*. The *key* must be a non-partition column and the *value* must be a string representation of an **Iceberg Transform**.

A **CreateTableCheck** planning rule will check that table's defined as managed are supported and if `partitionValueMappings` are specified these are validated. This / Rule/ is registered as a *custom analyzer rule* via a `Spark Session Extension`.

Insert Statement

Listing 1 shows the form of a Spark Insert statement. Figure 4 shows the Query Plan for an insert statement and figure 5 shows the details of how an Insert statement is executed. It is handled by 3 components: the `InsertHadoopFsRelation` Spark Command, the `FileFormat` Writer and the `File Commit Protocol`. Table metadata information(up to the granularity all table partitions) is retrieved and updated from the Spark Catalog, whereas File information and interaction is done via the `File System` API.

InsertHadoopFsRelation orchestrates the entire operation, it also handles interaction with the Spark Catalog. It's logic executes in the Driver of the SparkContext. The actions it performs are: compute the affected partitions based on the `partition specification` in the Insert statement, setup the File Commit Protocol and the Write Job that is associated with the File Format Writer, execute and commit/abort the job, compute the set of Added and Deleted partitions, and update the Spark Catalog.

File Commit Protocol tracks changes to data files made by the job and provides rudimentary level of job isolation. It provides a set of callbacks like new Task file, Task commit/abort and Job commit/abort that the other components use to notify it of file changes. On commit it moves files into their final locations, after which other operations will see the new list of the Table files.

Write Tasks create and write new Files, notify the File Commit Protocol of new files.

Listing 1: Spark Insert Command Form

```
insertInto
    : INSERT OVERWRITE TABLE tableIdentifier (partitionSpec (IF NOT EXISTS)?)?
    | INSERT INTO TABLE? tableIdentifier partitionSpec?
;

partitionSpec
    : PARTITION '(' partitionVal (',' partitionVal)* ')'
    ;
```

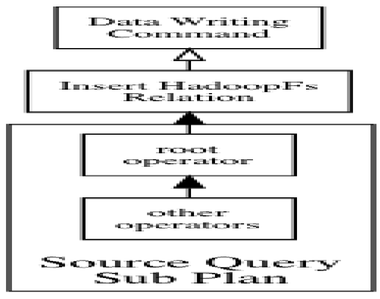

Figure 4: Spark Insert Plan

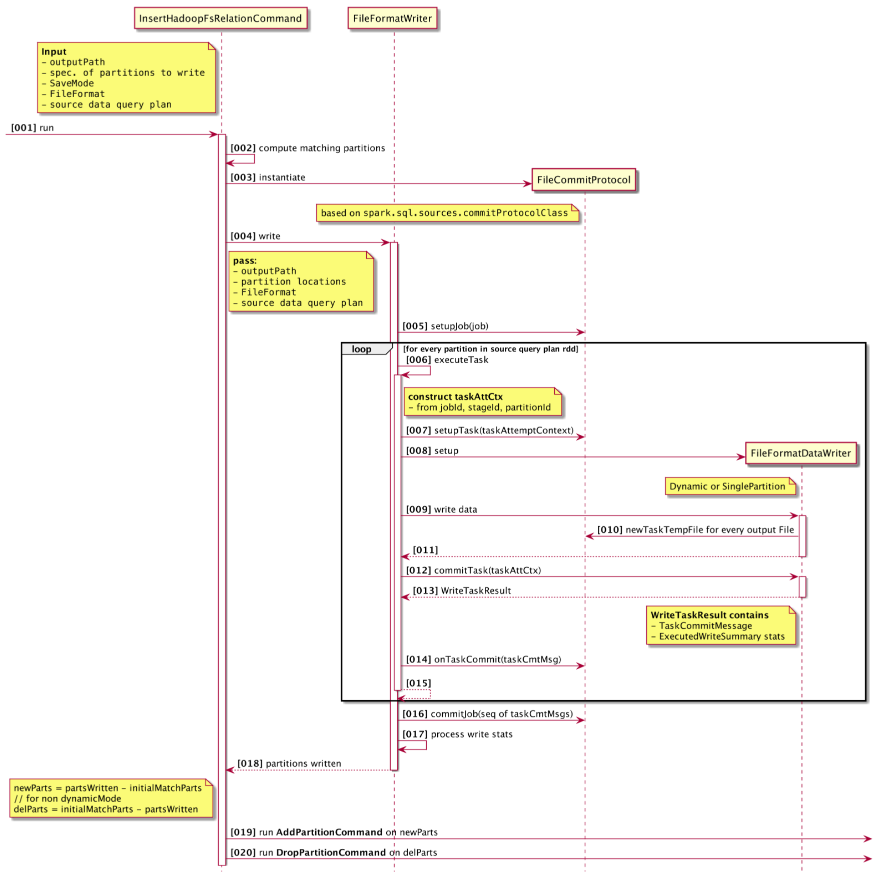
InsertHadoopFsRelationCommand
FileFormatWriter
Input
- outputPath
- spec. of partitions to write
- SaveMode
- FileFormat
- source data query plan
[001] run
[002] compute matching partitions
[003] instantiate
FileCommitProtocol
based on spark.sql.sources.commitProtocolClass
[004] write
pass:
- outputPath
- partition locations
- FileFormat
- source data query plan
[005] setupJob(job)
loop
[for every partition in source query plan rdd]
[006] executeTask
construct taskAttCtx
- from jobId, stageId, partitionId
[007] setupTask(taskAttemptContext)
[008] setup
FileFormatDataWriter
Dynamic or SinglePartition
[009] write data
[010] newTaskTempFile for every output File
[011]
[012] commitTask(taskAttCtx)
[013] WriteTaskResult
WriteTaskResult contains
- TaskCommitMessage
- ExecutedWriteSummary stats
[014] onTaskCommit(taskCmtMsg)
[015]
[016] commitJob(seq of taskCmtMsgs)
[017] process write stats
[018] partitions written
newParts = partsWritten - initialMatchParts
// for non dynamicMode
delParts = initialMatchParts - partsWritten
[019] run AddPartitionCommand on newParts
[020] run DropPartitionCommand on delParts

Figure 5: Spark Insert Command execution

Integrating with Iceberg

As part of the completion of an Insert we need to create a new Table metadata snapshot. Also the setup of the Table partitions(which ones are added or deleted will use the information from the latest Table snapshot). In order to achieve this we define a new **InsertIntoIcebergTable** command and an **Iceberg File Commit Protocol**.

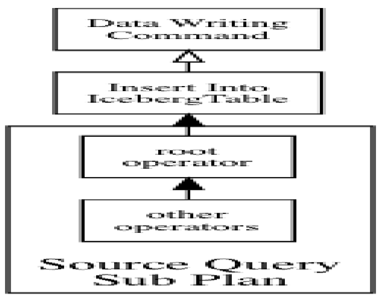

Figure 6: Spark Insert Iceberg Plan

The Spark Insert Plan in Figure 4 will be replaced by a plan show in Figure 6. An **Iceberg Management** *Spark Optimizer Rule* will be responsible for this rewrite. This *Optimizer Rule* is registered as a *customOperatorOptimization Rule* via a `Spark Session Extension`. These *customOperatorOptimization Rules* are the final optimizer rules applied during logical planning; so they don't alter Spark Planning behavior, only extend it.

Command to Insert Into Iceberg Table A drop-in replacement for InsertIntoHadoopFsRelationCommand setup by the `IcebergTableWriteRule`. By and large follows the same execution flow as `InsertIntoHadoopFsRelation` Command with the following behavior overrides.

- The write must be on a CatalogTable. So catalogTable parameter is not optional.
- Since this is a iceberg managed table we load the IceTable metadata for this table.
- `initialMatchingPartitions` is computed from the IceTable metadata
- since data files must be managed by iceberg custom partition locations cannot be configured for this table.
- an `IcebergFileCommitProtocol` is setup that wraps the underlying FileCommitProtocol. This mostly defers to the underlying commitProtocol instance; in the process it ensures iceberg DataFile instances are created for new files on task commit which are then delivered to the Driver `IcebergFileCommitProtocol` instance via `TaskCommitMessages`.
  - The underlying `FileCommitProtocol` is setup with `dynamicPartitionOverwrite` mode set to false. Since IceTable metadata is used by scan operations to compute what files to scan we don't have to do an all-or-nothing replacement of files in a partition that is needed for dynamic partition mode using the FileCommitProtocol.
- in case of dynamicPartitionOverwrite mode we don't clear specified source Partitions, because we want the current files to be able execute queries against older snapshots.
- once the job finishes the Catalog is updated with 'new' and 'deleted' partitions just as it is in a regular InsertIntoHadoopFsRelationCommand
- then based on the 'initial set' of DataFile and the set of DataFile created by tasks of this job a new iceberg Snapshot is created.
- finally cache invalidation and stats update actions happen just like in a regular InsertIntoHadoopFsRelationCommand.

Iceberg File Commit Protocol Provides the following function on top of the 'normal' Commit Protocol. Commit actions are simply deferred to the 'designate' except in the following:

- track files created for each Task in a TaskPaths instance. This tracks the temporary file location and also the location that the file will be moved to on a commit.
- on Task Commit build an Iceberg DataFile instance. Currently only if the file is a parquet file we will also build column level stats.
    - The TaskCommitMessage we send back has a payload of IcebergTaskCommitMessage, which encapsulates the TaskCommitMessage build by the 'designate' and the DataFile instances.
- we ignore deleteWithJob invocations, as we want to keep historical files around. These will be removed via a clear snapshot command.
- on a commitJob we extract all the DataFile instances from the IcebergTaskCommitMessage messages and expose a addedDataFiles list which is used by IceTableScanExec to build the new Iceberg Table Snapshot.

Select Statement

A table scan for a DataSource V1 table is performed by a `FileSourceScanExec` operator in Apache Spark. The `FileSourceScanExec` is setup by the `FileSource Strategy` optimizer rule that converts an `Project-Filter-Scan` Logical Plan into a FileSource Scan. It also splits any filters on the Table into partition and data column filters. A `FileSourceScanExec` is given all of this information such as the data and partition filters, the FileIndex, and table metadata. The `FileSourceScanExec` computes the partitions to scan based on the partition filters and then sets up an input RDD that has tasks for the files in the selected partitions.

The `FileSourceScanExec` will be replaced by a `IcebergTableScanExec`, that will setup an *Iceberg Table Scan* and try to push partition and data filters to the scan; the returned *Iceberg Scan Tasks* will be converted into a list of partitions(and within each partition the list of files) to scan. Apart from this the plan will execute with no other changes.

We will attempt to push data and partition Spark filters as *Iceberg Expressions* in a manner similar to the logic in `SparkExpressions` class in the `iceberg-spark` sub-project.

As of Select

Managing Iceberg Tables

Packaging and Deploying

We are providing this functionality as a drop-in jar that is available at [10]. In order to use it in your deployment you will have to add the following to your spark configuration:

```
spark.driver.extraClassPath=<location of downloaded icebergSparkSQL.jar>
spark.sql.extensions=org.apache.spark.sql.iceberg.SparkSessionExtensions
```

As of the writing of this document this will be available for `Apache Spark 2.4.2`, but we plan to back port this to `2.3.x` and `2.2.x` shortly.

References

[1] *Iceberg Table Spec.* https://iceberg.apache.org/spec/.

[2] *Iceberg, A modern table format for big data.* https://tinyurl.com/y6z6rcg4.

[3] *Iceberg Apache Github.* https://github.com/apache/incubator-iceberg.

[4] *Iceberg Netflix Github.* https://github.com/Netflix/iceberg.

[5] *Apache Spark Data Source V2.* https://databricks.com/session/apache-spark-data-source-v2.

[6] *Data source API v2.* https://tinyurl.com/y5u576gk.

[7] *Data Source V2 improvements.* https://tinyurl.com/yylna72p.

[8] Peter Jeffcock. *Interactive Data Lake Queries At Scale.* https://tinyurl.com/y8hbyp9q.

[9] Greg Pavlik and Diwakar Goel. *Innovating with the Oracle Platform for Data Science and Big Data.* https://tinyurl.com/y59k23bf.

[10] *Iceberg Spark SQL Github.* https://github.com/hbutani/icebergSparkSQL.

Appendix A: Iceberg Examples

Listing 2: Iceberg Hive Example

```
/*  HIVE EXAMPLE */

// HiveTables is the bridge between Hive Metastore and the Iceberg Tables interface

// a. CREATE TABLE
new HiveTables(this.hiveConf).create(schema, partitionSpec, DB_NAME, TABLE_NAME);

// b.ALTER TABLE, ADD COLUMN
com.netflix.iceberg.Table icebergTable = new HiveTables(hiveConf).
            load(DB_NAME, TABLE_NAME);
```

Listing 3: Iceberg Parquet Example

```
/* PARQUET EXAMPLE */

// HadoopTables is an implementation of Iceberg Tables interface
// that relies entirely on metadata stored on disk
// this example shows using HadoopTables to manage Iceberg Tables
// using the Parquet file format

private static final Tables TABLES = new HadoopTables(CONF);

// a. CREATE TABLE
this.sharedTable = TABLES.create(
          SCHEMA, PartitionSpec.unpartitioned(),
          ImmutableMap.of(TableProperties.DEFAULT_FILE_FORMAT, format.name()),
          sharedTableLocation);

// b. WRITE DATA
// a FileAppender for Parquet bridge to ParquetFileWriter mechanics
// converts ParquetMetadata from footer to Iceberg DataFile metrics
FileAppender<Record> appender = Parquet.write(fromPath(path, CONF))
                .schema(SCHEMA)
                .createWriterFunc(GenericParquetWriter::buildWriter)
                .build());
appender.addAll(records);

// b2. COMMIT NEW SNAPSHOT
sharedTable.newAppend()
            .appendFile(DataFiles.builder(PartitionSpec.unpartitioned())
                .withInputFile(file1)
                .withMetrics(new Metrics(3L,
                    null, // no column sizes
                    ImmutableMap.of(1, 3L), // value count
                    ImmutableMap.of(1, 0L), // null count
                    ImmutableMap.of(1, longToBuffer(0L)), // lower bounds
                ImmutableMap.of(1, longToBuffer(2L)))) // upper bounds)
            .build())
        .commit();

// c. SCAN
Iterable<Record> result =
  IcebergGenerics.read(sharedTable).where(lessThan("id", 3)).build();
// under the covers
// - Parquet.ReadBuilder sets up a ParquetReader
// - ParquetReader is the bridge to ParquetFileReader
Parquet.ReadBuilder pRB = Parquet.read(input)
            .project(projection)
            .createReaderFunc(fileSchema -> buildReader(projection, fileSchema))
            .split(task.start(), task.length());
// in pRB.build call
// org.apache.parquet.hadoop.ParquetReader is setup with
// column pruning and predicate pushdown applied
```

Listing 4: Iceberg Spark Example

```
/* SPARK EXAMPLE */
// Iceberg is surfaced as a DataSourceV2 with ReadSupport,
//      WriteSupport, DataSourceRegister

// a. WRITE EXAMPLE

// df is some DataFrame that is source of data being written
df.select("id", "data").write()
        .format("iceberg")
        .mode("append")
        .save(location.toString());

// b. READ EXAMPLE
Dataset<Row> df = spark.read()
        .format("iceberg")
        .load(location.toString());

// c. CREATE TABLE EXAMPLE
// this would be based on the CatalogPlugin concept of DataSource v2
```

Skip

intro

skip

This is a specification for the Iceberg table format that is designed to manage a large, slow-changing collection of files in a distributed file system or key-value store as a table.

Iceberg defines a **management scheme** for a dataset that is stored as a large, slow-changing collection of data files in a distributed file system. It provides:

of core capabilities for managing Table snapshots, schema evolution, efficient table data access using predicate and projection pushdown, hidden partition layouts and layout evolution.

The components are packaged into various libraries that are used in different settings: the core runtime component(this provides the Table capabilities used by all other components), various components to write Iceberg Tables using parquet, orc or avro file formats, a component to manage Hive tables as Iceberg Tables, a component to surface Iceberg Tables in Pig, integration with Presto, and a component that surfaces Iceberg Tables as a Spark Datasource V2 [5] table.

spark insert description

- executed as an `InsertHadoopFsRelationCommand`

- based on `partition specification` on the command a set of *matching partitions* is computed.
- A FileCommitProtocol is setup to record the filesystem actions of the Command.
- The input is shuffled and sorted to match the partition specification of the output Table.
- A WriteTask is executed for each input RDD partition
- each new file needed is registered with the FileCommitProtocol
- when Task finishes successfully the FileCommitProtocol is asked to commit the task; which returns a `WriteTaskResult`
- when all Tasks of the job finish the FileCommitProtocol is asked to commit the Job, it is passed all the `WriteTaskResults` from the Task commits
- based on the updated partitions a list of new and deleted partitions are computed and Add Partition and Drop Partition Commands are issued for these.