..

Using Apache Iceberg

Date published: 2022-03-15
Date modified: 2026-08-28



https://docs.cloudera.com/

Legal Notice

© Cloudera Inc. 2026. All rights reserved.

The documentation is and contains Cloudera proprietary information protected by copyright and other intellectual property rights. No license under copyright or any other intellectual property right is granted herein.

Unless otherwise noted, scripts and sample code are licensed under the Apache License, Version 2.0.

Copyright information for Cloudera software may be found within the documentation accompanying each component in a particular release.

Cloudera software includes software from various open source or other third party projects, and may be released under the Apache Software License 2.0 (“ASLv2”), the Affero General Public License version 3 (AGPLv3), or other license terms. Other software included may be released under the terms of alternative open source licenses. Please review the license and notice files accompanying the software for additional licensing information.

Please visit the Cloudera software product page for more information on Cloudera software. For more information on Cloudera support services, please visit either the Support or Sales page. Feel free to contact us directly to discuss your specific needs.

Cloudera reserves the right to change any products at any time, and without notice. Cloudera assumes no responsibility nor liability arising from the use of products, except as expressly agreed to in writing by Cloudera.

Cloudera, Cloudera Altus, HUE, Impala, Cloudera Impala, and other Cloudera marks are registered or unregistered trademarks in the United States and other countries. All other trademarks are the property of their respective owners.

Disclaimer: EXCEPT AS EXPRESSLY PROVIDED IN A WRITTEN AGREEMENT WITH CLOUDERA, CLOUDERA DOES NOT MAKE NOR GIVE ANY REPRESENTATION, WARRANTY, NOR COVENANT OF ANY KIND, WHETHER EXPRESS OR IMPLIED, IN CONNECTION WITH CLOUDERA TECHNOLOGY OR RELATED SUPPORT PROVIDED IN CONNECTION THEREWITH. CLOUDERA DOES NOT WARRANT THAT CLOUDERA PRODUCTS NOR SOFTWARE WILL OPERATE UNINTERRUPTED NOR THAT IT WILL BE FREE FROM DEFECTS NOR ERRORS, THAT IT WILL PROTECT YOUR DATA FROM LOSS, CORRUPTION NOR UNAVAILABILITY, NOR THAT IT WILL MEET ALL OF CUSTOMER’S BUSINESS REQUIREMENTS. WITHOUT LIMITING THE FOREGOING, AND TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, CLOUDERA EXPRESSLY DISCLAIMS ANY AND ALL IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, QUALITY, NON-INFRINGEMENT, TITLE, AND FITNESS FOR A PARTICULAR PURPOSE AND ANY REPRESENTATION, WARRANTY, OR COVENANT BASED ON COURSE OF DEALING OR USAGE IN TRADE.

Contents

Apache Iceberg features **5**
Alter table feature 7
Create table feature 8
Create table as select feature 11
Create partitioned table as select feature 11
Create table … like feature 12
Data compaction 12
Delete data feature 14
Describe table metadata feature 15
Drop table feature 16
Drop partition feature 17
Expire snapshots feature 17
Insert table data feature 19
Load data inpath feature 20
Load or replace partition data feature 20
Materialized view feature 21
Materialized view rebuild feature 22
Merge feature 24
Migrate Hive table to Iceberg feature 25
Securing Iceberg table data 26
Flexible partitioning 26
Partition evolution feature 27
Partition transform feature 27
Insert into/overwrite partition feature 29
Truncate partition feature 30
Query metadata tables feature 30
Rollback table feature 32
Row-level operations 32
Row lineage feature [Technical Preview] 34
Puffin statistics feature 34
Select Iceberg data feature 34
Schema evolution feature 35
Schema inference feature 36
Snapshot management 37
Branching feature 37
Tagging feature 38
Table repair feature 39
LIKE predicate pushdown feature 39
Time travel feature 40
Truncate table feature 41
Update data feature 41
Best practices for Iceberg in Cloudera **42**
Making row-level changes on V2 tables only 43
Performance tuning **43**
Caching manifest files 44
Configuring manifest caching in Cloudera Manager 44
Unsupported features and limitations **45**
Accessing Iceberg tables **47**
Opening Ranger in Cloudera Data Hub 47
Editing a storage handler policy to access Iceberg files on the file system 48

Creating a SQL policy to query an Iceberg table....51
Creating an Iceberg table....52
Creating an Iceberg partitioned table....53
Expiring snapshots....54
Inserting data into a table....54
Migrating a Hive table to Iceberg....55
Selecting an Iceberg table....55
Running time travel queries....56
Updating an Iceberg partition....57
Test driving Iceberg from Impala....57
Hive demo data....59
Test driving Iceberg from Hive....62
Iceberg data types....63
Iceberg table properties....64

Apache Iceberg features

You can quickly build on your past experience with SQL to analyze Iceberg tables.

From Hive or Impala, you run SQL queries to create and query Iceberg tables. Impala queries are table-format agnostic. For example, Impala options are supported in queries of Iceberg tables. You can run nested, correlated, or analytic queries on all supported table types. Most Hive queries are also table-format agnostic.

You can use Impala for manipulating Iceberg tables. The provided examples show how to run queries on Iceberg tables from Impala but do not list every supported Impala query.

If your environment uses HDFS HA, you must enable it before creating Iceberg tables. For instructions, see Enable HDFS HA before you create Iceberg tables.

Support for Apache Iceberg library version 1.10.0

Starting with Cloudera Runtime 7.3.2 SP1, Apache Iceberg library version 1.10.0 is supported for Hive and Impala. Existing Iceberg V1 and V2 table operations are unchanged; you do not need new SQL syntax for current workflows.

Variant data type

Hive supports the Iceberg variant data type on Iceberg V3 tables. In Hive, use PARSE_JSON to convert a JSON string into a variant value and VARIANT_GET to extract a nested field using a JSON path and return type.

For more information, see Iceberg data types.

Deletion vector support for Hive

Hive supports deletion vectors on Iceberg V3 tables. Deletion vectors are the Iceberg V3 encoding for position deletes. When you delete rows from a V3 table, Hive records which rows are deleted by position in the referenced data file.

Deletion vectors are a a bitmap of deleted row positions within a single data file. On V3 tables, Hive writes deletion vectors instead of V2 position delete files for new delete operations. Read operations merge active rows with the deletion vectors to return query results. You need not run a new SQL syntax to write deletion vectors. Create an Iceberg table with 'format-version'='3' and run a standard DELETE statement. Hive encodes the delete operation using deletion vectors automatically.

For more information, see Delete data feature and Row-level operations.

Deletion vector support for Impala [Technical Preview]



Note: This feature is in Technical Preview and is not ready for production deployments. Cloudera recommends trying this feature in test or development environments and encourages you to provide feedback on your experiences.

Impala supports deletion vectors on Iceberg V3 tables. Deletion vectors are the Iceberg V3 encoding for position deletes. When you delete rows from a V3 table from Impala, Impala records which rows are deleted by position in the referenced data file.

On V3 tables, Impala writes deletion vectors instead of V2 position delete files for new delete operations. Read operations merge active rows with the deletion vectors to return query results. You need not run a new SQL syntax to write deletion vectors. Create an Iceberg table with 'format-version'='3' and run a standard DELETE statement. Impala encodes the delete operation using deletion vectors automatically.

For more information, see Delete data feature and Row-level operations.

Row lineage support for Impala [Technical Preview]

Note: This feature is in Technical Preview and is not ready for production deployments. Cloudera recommends trying this feature in test or development environments and encourages you to provide feedback on your experiences.

Impala supports row lineage tracking on Iceberg format version 3 (V3) tables. Row lineage is an Iceberg V3 capability that assigns a unique row identifier to each row and records which snapshot last updated a row. This metadata supports change data capture (CDC) use cases. To use row lineage from Impala, create an Iceberg V3 table by setting 'format-version'='3'. When Impala writes to the table, Impala maintains the row lineage metadata required; you need not run a new SQL syntax to enable row lineage. Row lineage applies to newly written rows on V3 tables. It does not apply to rows updated using equality deletes.

For more information, see Row lineage feature.

Puffin statistics support for Impala

Impala supports writing Puffin statistics files on Iceberg V3 tables. Puffin is a lightweight container format that stores binary metadata such as deletion vector sidecar files required by the Iceberg V3 specification. To use Puffin statistics from Impala, create an Iceberg V3 table by setting 'format-version'='3'. When Impala performs supported V3 write operations, Impala writes Puffin files as required by the Iceberg V3 specification; you need not run a new SQL syntax to enable Puffin statistics writes.

For more information, see Puffin statistics feature.

Table repair support for Impala

Impala supports the repair_metadata() function for Iceberg tables. Use this function to recover tables that are inaccessible due to missing data files after manual file deletions in the underlying storage. The function synchronizes Iceberg metadata by removing references to missing data files; it does not restore deleted content.

For more information, see Table repair feature.

LIKE predicate pushdown support for Impala

Impala pushes eligible SQL LIKE predicates on Iceberg tables to Iceberg startsWith() and equal() expressions during query planning. This optimization uses Iceberg manifest statistics for file and partition pruning.

For more information, see LIKE predicate pushdown feature.

Supported ACID transaction properties

Iceberg supports atomic and isolated database transaction properties. Writers work in isolation, not affecting the live table, and perform a metadata swap only when the write is complete, making the changes in one atomic commit.

Iceberg uses snapshots to guarantee isolated reads and writes. You see a consistent version of table data without locking the table. Readers always see a consistent version of the data without the need to lock the table. Writers work in isolation, not affecting the live table, and perform a metadata swap only when the write is complete, making the changes in one atomic commit.

Iceberg partitioning

The Iceberg partitioning technique has performance advantages over conventional partitioning, such as Apache Hive partitioning. Iceberg hidden partitioning is easier to use. Iceberg supports in-place partition evolution; to change a partition, you do not rewrite the entire table to add a new partition column, and queries do not need to be rewritten for the updated table. Iceberg continuously gathers data statistics, which supports additional optimizations, such as partition pruning.

Iceberg uses multiple layers of metadata files to find and prune data. Hive and Impala keep track of data at the folder level and not at the file level, performing file list operations when working with data in a table. Performance problems occur during the execution of multiple list operations. Iceberg keeps track of a complete list of files within a table

using a persistent tree structure. Changes to an Iceberg table use an atomic object/file level commit to update the path to a new snapshot file. The snapshot points to the individual data files through manifest files.

The manifest files track several data files across many partitions. These files store partition information and column metrics for each data file. A manifest list is an additional index for pruning entire manifests. File pruning increases efficiency.

Iceberg relieves Hive metastore (HMS) pressure by storing partition information in metadata files on the file system/object store instead of within the HMS. This architecture supports rapid scaling without performance hits.

Alter table feature

In Hive or Impala, you can use ALTER TABLE to set table properties. From Impala, you can use ALTER TABLE to rename a table, to change the table owner, or to change the role of the table owner. From Hive, you can alter the metadata location of the table if the new metadata does not belong to another table; otherwise, an exception occurs.

You can convert an Iceberg v1 table to v2 by setting a table property as follows:'format-version' = '2'.

Hive or Impala syntax

```
ALTER TABLE table_name SET TBLPROPERTIES table_properties;
```

- table_properties

  A list of properties and values using the following syntax:

  ```
  ('key' = 'value', 'key' = 'value', ... )
  ```

Impala syntax

```
ALTER TABLE table_name RENAME TO new_table_name;

ALTER TABLE table_name SET OWNER USER user_name;

ALTER TABLE table_name SET OWNER ROLE role_name;
```

Hive example

```
ALTER TABLE test_table SET TBLPROPERTIES('metadata_location'='s3a://bucketNa
me/ice_table/metadata/v1.metadata.json');
ALTER TABLE test_table2 SET TBLPROPERTIES('format-version' = '2');
```

Impala examples

```
ALTER TABLE t1 RENAME TO t2;

ALTER TABLE ice_table1 set OWNER USER john_doe;

ALTER TABLE ice_table2 set OWNER ROLE some_role;
ALTER TABLE ice_8 SET TBLPROPERTIES ('read.split.target-size'='268435456');

ALTER TABLE ice_table3 SET TBLPROPERTIES('format-version' = '2');
```

Create table feature

You use CREATE TABLE from Impala or CREATE EXTERNAL TABLE from Hive to create an external table in Iceberg. You learn the subtle differences in these features for creating Iceberg tables from Hive and Impala. You also learn about partitioning.

Hive and Impala handle external table creation a little differently, and that extends to creating tables in Iceberg. By default, Iceberg tables you create are v1. To create an Iceberg v2 table from Hive or Impala, you need to set a table property as follows:'format-version' = '2'.

The VARIANT data type is an Iceberg format version 3 feature. To create a Hive Iceberg table that supports the VARIANT data type, set the table property 'format-version'='3', similar to how ice_6 sets 'format-version'='2' in the Hive examples below.

Enable HDFS HA before you create Iceberg tables

Learn about how Apache Iceberg stores absolute paths in its metadata manifest files, and how HDFS HA ensures that the table is accessible regardless of which NameNode is active.

Background

Apache Iceberg stores absolute paths in its metadata manifest files for Iceberg tables. Opposed to this, Apache Hive writes the relative path to the table into the metadata, while building the absolute path based on the HDFS configuration.

The format of these paths depends on your HDFS configuration at the time of table creation. Without HDFS HA, Iceberg writes the paths using the specific hostname and port of the active NameNode. With HDFS HA, Iceberg writes the paths using the logical nameservice. Using the HDFS nameservice ensures that the paths remain valid regardless of which NameNode is active.

To avoid possible data loss when switching from one HDFS configuration to another, Cloudera highly recommends enabling HDFS HA in the environment.

Explanation

Writing a table in Hadoop in either Hive or Iceberg format includes the data itself and the metadata of the table.

While Cloudera stores the metadata in the Hive Metastore (HMS) for both Hive and Iceberg tables, the method that table location is written in the metadata varies:

- Hive uses a relative path, where the table location is concatenated with the relative path in the metadata and the HDFS definition for the NameNode or NameService that is active.
- Iceberg uses an absolute path in the table metadata to describe the location of the table at the time of its creation. If HDFS runs in non-HA mode, the cluster uses either the Active NameNode and Standby NameNode, and the used one is written to the table information in the hdfs://node4.cloudera.com:8020/warehouse/tablespace/external/hive/test.db/ice_table/ format.

  That path is accessible as long as that NameNode (node4.cloudera.com) is active. However, if the node becomes unavailable, and the secondary (standby) node takes over, the table becomes inaccessible because the original node that was used to write the table is not active. This applies to both the table information itself and the manifest JSON file associated with it, containing the absolute path to the table location.

When you enable HDFS HA, the service removes the NameNode references, and uses nameservice as the only reference. The nameservice internally manages its NameNodes, such as Load Balancer, exposing only the nameservice as the access point.

Tables created after HDFS HA is enabled use the following format (assuming the nameservice is named nameservice01): hdfs://nameservice01/warehouse/tablespace/external/hive/test.db/ice_table/

This format ensures that the table is accessible regardless of which NameNode is active, as the NameService takes care of the path, thereby supporting the failover cases.

Note: If the solution includes another cluster to which the primary cluster synchronizes data, such as in the case of Disaster Recovery, the same NameService name must be maintained on the destination cluster.

Iceberg table creation from Hive

From Hive, CREATE EXTERNAL TABLE is recommended to create an Iceberg table in Cloudera.

When you use the EXTERNAL keyword to create the Iceberg table, by default only the schema is dropped when you drop the table. The actual data is not purged. Conversely, if you do not use EXTERNAL, by default the schema and actual data is purged. You can override the default behavior. For more information, see the Drop table feature.

From Hive, you can create a table that reuses existing metadata by setting the metadata_location table property to the object store path of the metadata. The operation skips generation of new metadata and re-registers the existing metadata. Use the following syntax:

```
CREATE EXTERNAL TABLE ice_fm_hive (i int) STORED BY ICEBERG TBLPROPERTIES ('
metadata_location'='<object store or file system path>')
```

See examples below.

Iceberg table creation from Impala

From Impala, CREATE TABLE is recommended to create an Iceberg table in Cloudera. Impala creates the Iceberg table metadata in the metastore and also initializes the actual Iceberg table data in the object store.

The difference between Hive and Impala with regard to creating an Iceberg table is related to Impala compatibility with Kudu, HBase, and other tables. For more information, see the Apache documentation, "Using Impala with Iceberg Tables".

Iceberg table creation from a non-Iceberg table (Impala)

From Impala, you can use `CREATE TABLE LIKE` with STORED BY ICEBERG to create an Iceberg table that uses the column schema of an existing non-Iceberg table in the metastore. Supported source tables include HDFS-backed formats such as Parquet, ORC, Avro, and Text.

The operation creates Iceberg table metadata only; it does not copy table data. Use `INSERT INTO` or `INSERT OVERWRITE` to load data into the new Iceberg table after you create it.

```
CREATE TABLE my_ice_tbl LIKE functional_parquet.complextypestbl STORED BY IC
EBERG;
```

Metadata storage of Iceberg tables

When you create an Iceberg table using CREATE EXTERNAL TABLE in Hive or using CREATE TABLE in Impala, HiveCatalog creates an HMS table and also stores some metadata about the table on your object store, such as S3. Creating an Iceberg table generates a metadata.json file, but not a snapshot. In the metadata.json, the snapshot-id of a new table is -1. Inserting, deleting, or updating table data generates a snapshot. The Iceberg metadata files and data files are stored in the table directory under the warehouse folder. Any optional partition data is converted into Iceberg partitions instead of creating partitions in the Hive Metastore, thereby removing the bottleneck.

To create an Iceberg table from Hive or from Impala, you associate the Iceberg storage handler with the table using one of the following clauses, respectively:

- Hive: STORED BY ICEBERG
- Impala: STORED AS ICEBERG or STORED BY ICEBERG

Supported file formats

You can write Iceberg tables in the following formats:

- From Hive: Parquet (default), Avro, ORC
- From Impala: Parquet

Impala supports writing Iceberg tables in only Parquet format. Impala does not support defining both file format and storage engine. For example, CREATE TABLE tbl ... STORED AS PARQUET STORED BY ICEBERG works from Hive, but not from Impala.

You can read Iceberg tables in the following formats:

- From Hive: Parquet, Avro, ORC
- From Impala: Parquet, Avro, ORC



Note: Reading Iceberg tables in Avro format from Impala is available as a technical preview. Cloudera recommends that you use this feature in test and development environments. It is not recommended for production deployments.

Hive syntax

```
CREATE [EXTERNAL] TABLE [IF NOT EXISTS] [db_name.]table_name
  [(col_name data_type, ... )]
  [PARTITIONED BY [SPEC]([col_name][, spec(value)][, spec(value)]...)]]
   STORED BY ICEBERG
  [STORED AS file_format]
  [TBLPROPERTIES ('key'='value', 'key'='value', ...)]
```

Impala syntax

```
CREATE TABLE [IF NOT EXISTS] [db_name.]table_name
  [(col_name data_type, ... )]
  [PARTITIONED BY [SPEC]([col_name][, spec(value)][, spec(value)]...)]]
   STORED {AS | BY} ICEBERG
  [TBLPROPERTIES (property_name=property_value, ...)]
```

Hive examples

```
CREATE EXTERNAL TABLE ice_1 (i INT, t TIMESTAMP, j BIGINT) STORED BY ICEBERG
;
CREATE EXTERNAL TABLE ice_2 (i INT, t TIMESTAMP) PARTITIONED BY (j BIGINT)
 STORED BY ICEBERG;
CREATE EXTERNAL TABLE ice_4 (i int) STORED BY ICEBERG STORED AS ORC;
CREATE EXTERNAL TABLE ice_5 (i int) STORED BY ICEBERG TBLPROPERTIES ('metad
ata_location'='s3a://bucketName/ice_table/metadata/v1.metadata.json')
CREATE EXTERNAL TABLE ice_6 (i int) STORED BY ICEBERG STORED AS ORC TBLPRO
PERTIES ('format-version' = '2');
```

```
CREATE EXTERNAL TABLE ice_3 (v VARIANT) STORED BY ICEBERG TBLPROPERTIES ('fo
rmat-version'='3');
```

Impala examples

```
CREATE TABLE ice_7 (i INT, t TIMESTAMP, j BIGINT) STORED BY ICEBERG; //creat
es only the schema
CREATE TABLE ice_8 (i INT, t TIMESTAMP) PARTITIONED BY (j BIGINT) STORED BY
 ICEBERG; //creates schema and initializes data
CREATE TABLE ice_v2 (i INT, t TIMESTAMP) PARTITIONED BY (j BIGINT) STORED BY
 ICEBERG TBLPROPERTIES ('format-version' = '2'); //creates a v2 table
```

Related Information

Drop table feature
Partition transform feature

Create table as select feature

You can create an Iceberg table based on an existing Hive or Impala table.

The create table as select (CTAS) query can optionally include a partitioning spec for the table being created.

Hive examples

```
CREATE EXTERNAL TABLE ctas STORED BY ICEBERG AS SELECT i, t, j FROM ice_1;
```

Impala examples

```
CREATE TABLE ctas STORED BY ICEBERG AS SELECT i, b FROM ice_11;
```

Create partitioned table as select feature

You can create a partitioned Iceberg table by selecting another table. You see an example of how to use PARTITIONED BY and TBLPROPERTIES to declare the partition spec and table properties for the new table.

You see an example of using a partition transform with the PARTITIONED BY SPEC clause.

The newly created table does not inherit the partition spec and table properties from the source table in SELECT. The Iceberg table and the corresponding Hive table is created at the beginning of the query execution. The data is inserted / committed when the query finishes. So for a transient period the table exists but contains no data.

Hive syntax

```
CREATE [EXTERNAL] TABLE prod.db.sample
    USING iceberg
    PARTITIONED BY (part)
    TBLPROPERTIES ('key'='value')
    AS SELECT ...
```

Hive examples

```
CREATE EXTERNAL TABLE ctas STORED BY ICEBERG AS SELECT i, t, j FROM ice_1;

CREATE EXTERNAL TABLE ctas_part PARTITIONED BY(z) STORED BY ICEBERG TBLPROPE
RTIES ('format-version'='2')
AS SELECT x, ts, z FROM t;
CREATE EXTERNAL TABLE ctas_part_spec PARTITIONED BY SPEC (month(d)) STORED
BY ICEBERG TBLPROPERTIES ('format-version'='2')
AS SELECT x, ts, d FROM source_t;
```

Impala examples

```
CREATE TABLE ctas STORED BY ICEBERG AS SELECT i, b FROM ice_11;

CREATE TABLE ctas_part PARTITIONED BY(b) STORED BY ICEBERG AS SELECT i, s, b
 FROM ice_11;
```

```
CREATE TABLE ctas_part_spec PARTITIONED BY SPEC (month(d)) STORED BY ICEBERG
 TBLPROPERTIES ('format-version'='2')
AS SELECT x, ts, d FROM source_t;
```

Create table … like feature

You learn by example how to create an empty table based on another table.

From Hive or Impala, you can create an Iceberg table schema based on another table. The table contains no data. The table properties of the original table are carried over to the new table definition. The following examples show how to use this feature:

Hive example

```
CREATE EXTERNAL TABLE target LIKE source STORED BY ICEBERG;
```

Impala example

```
CREATE TABLE target LIKE source STORED BY ICEBERG;
```

Data compaction

From Hive and Impala, you can compact Iceberg tables and optimize them for read operations. Compaction is an essential table maintenance activity that creates a new snapshot, which contains the table content in a compact form.

Hive Iceberg compaction supports WHERE clause predicates only on partition columns.

Frequent updates and row-level modifications on Iceberg tables can result in many small data files and delete files, which have to be merged-on-read. This degrades the query performance over time. You can use the following Hive and Impala SQL statements to compact Iceberg tables and optimize the table for reading.

Impala syntax

```
OPTIMIZE TABLE [db_name.]table_name (FILE_SIZE_THRESHOLD_MB=<value>);
```

The FILE_SIZE_THRESHOLD_MB enables you to specify the maximum size of files (in MB) that should be considered for compaction. Data files larger than the specified limit are rewritten only if they are referenced from delete files.



Note:

- If FILE_SIZE_THRESHOLD_MB is specified, only the files meeting the file size criteria are rewritten according to the latest schema and partition spec and the remaining data files might still have an older schema or partition layout. Therefore, run the OPTIMIZE TABLE statement without the file size threshold option to rewrite the entire table according to the latest schema and partititon layout.
- OPTIMIZE TABLE statement without a specified FILE_SIZE_THRESHOLD_MB rewrites the entire table and the operation can take longer to complete depending on the table size. Therefore, it is recommended that you specify a file size threshold for recurring table maintenance jobs to save compute resources.

Impala example

```
OPTIMIZE TABLE ice_table (FILE_SIZE_THRESHOLD_MB=128);
```

Hive syntax

```
ALTER TABLE [database_name.]table_name COMPACT 'compaction_type' [AND WAIT];
```

```
OPTIMIZE TABLE [database_name.]table_name REWRITE DATA;
```

Hive example

```
ALTER TABLE ice_table COMPACT 'MAJOR';
```

```
OPTIMIZE TABLE ice_table REWRITE DATA;
```



Important: When Cloudera Data Warehouse and Cloudera Data Hub are deployed in the same environment and use the same Hive Metastore (HMS) instance, the Cloudera Data Hub compaction workers can inadvertently pick up Iceberg compaction tasks. Since Iceberg compaction is not yet supported in the latest Cloudera Data Hub version, the compaction tasks will fail when they are processed by the Cloudera Data Hub compaction workers.

In such a scenario where both Cloudera Data Warehouse and Cloudera Data Hub share the same HMS instance and there is requirement to run both Hive ACID and Iceberg compaction jobs, it is recommended that you use the Cloudera Data Warehouse environment for these jobs. If you want to run only Hive ACID compaction tasks, you can choose to use either the Cloudera Data Warehouse or Cloudera Data Hub environments.

If you want to run the compaction jobs without changing the environment, it is recommended that you use Cloudera Data Warehouse. To avoid interference from Cloudera Data Hub, change the value of the hive .compactor.worker.threads Hive Server (HS2) property in Cloudera Data Hub to '0'. This ensures that the compaction jobs are not processed by Cloudera Data Hub. You can find this property in Cloudera Manager Hive Configuration .

This behavior is applicable only to the Hive Iceberg manual compaction and does not affect Cloudera Lakehouse Optimizer.

To perform table optimization, ensure that the following prerequisites are met:

- The user performing compaction must have the 'ALL' permissions on the table, which can be set through Ranger.
- Impala can only write Parquet files, therefore the write.format.default table property must be set to parquet. Hive can write both Parquet and ORC file formats.
- Impala cannot compact tables with complex data types.
- Impala cannot compact views.

The OPTIMIZE TABLE statement rewrites the entire table, performing the following tasks:

- Compact small files into larger files
- Merge delete files created due to previously run DELETE and UPDATE operations
- Rewrite all files, converting them to the latest table schema
- Rewrite all partitions according to the latest partition specification
- Compact tables with partition evolution

When an Iceberg table is optimized, a new snapshot is created where all the old files of the table are replaced with newly written files. Note that if the FILE_SIZE_THRESHOLD_MB option is specified, only the files meeting the file size threshold limit are replaced. The old table state and old files can still be queried using time travel, because the rewritten data and delete files are not removed from the system. This can lead to the accumulation of unused files that belong to old snapshots. Use the *Expire Snapshots* feature to permanently remove the old files from the file system.



Note: For full compaction, since the compaction process rewrites the entire table, the operation can take a long time to complete depending on the table size.

Delete data feature

Cloudera supports row-level deletes in Iceberg V2 tables.

From Hive and Impala, you can use the following formatting types defined by the Iceberg Spec:

- position deletes
- equality deletes



Note: The equality deletes feature is in technical preview and not recommended for use in production deployments. Cloudera recommends that you try this feature in test and development environments.

- deletion vectors (Iceberg V3 table format only)

For equality deletes, you must be aware of the following considerations:

- If you are using Apache Flink or Apache NiFi to write equality deletes, then ensure that you provide a PRIMARY KEY for the table. This is required for engines to know which columns to write into the equality delete files.
- If the table is partitioned then the partition columns have to be part of the PRIMARY KEY
- For Apache Flink, the table should be in 'upsert-mode' to write equality deletes
- Partition evolution is not allowed for Iceberg tables that have PRIMARY KEYs

Position delete files contain the following information:

- file_path, which is a full URI
- pos, the file position of the row

Delete files are sorted by file_path and pos. The following table shows an example of delete files in a partitioned table:

Partition	Partition information	Delete file
YEAR(ts)=2022	ts_year=2022/data-abcd.parquet ts_year=2022/data-bcde.parquet	ts_year=2022/delete-wxyz.parquet
YEAR(ts)=2023	ts_year=2023/data-efgh.parquet	ts_year=2023/delete-hxkw.parquet
MONTH(ts)=2023-06	ts_month=2023-06/data-ijkl.parquet	ts_month=2023-06/delete-uzwd.parquet
MONTH(ts)=2023-07	ts_month=2023-07/data-mnop.parquet	ts_month=2023-07/delete-udqx.parquet

Inserting, deleting, or updating table data generates a snapshot.

You use a WHERE clause in your DELETE statement. For example:

```
delete from tbl_ice where a <= 2,1;
```

Hive and Impala evaluate rows from one table against a WHERE clause, and delete all the rows that match WHERE conditions. If you want delete all rows, use the Truncate feature. The WHERE expression is similar to the WHERE expression used in SELECT. The conditions in the WHERE clause can refer to any columns.

Concurrent operations that include DELETE do not introduce inconsistent table states. Iceberg runs validation checks to check for concurrent modifications, such as DELETE+INSERT. Only one will succeed. On the other hand, DELETE+DELETE, and INSERT+INSERT can both succeed, but in the case of a concurrent DELETE+UPDATE, UPDATE+UPDATE, DELETE+INSERT, UPDATE+INSERT from Hive, only the first operation will succeed.

From joined tables, you can delete all matching rows from one of the tables. You can join tables of any kind, but the table from which the rows are deleted must be an Iceberg table. The FROM keyword is required in this case, to separate the name of the table whose rows are being deleted from the table names of the join clauses.

For deletion vectors, you need not run a new SQL syntax to write deletion vectors. Run a standard DELETE statement on a V3 Iceberg table. Set 'format-version'='3' when you create the table. Hive encodes position deletes as deletion vectors.

Hive or Impala examples

For Impala:

Note: This feature is in Technical Preview and is not ready for production deployments. Cloudera recommends trying this feature in test or development environments and encourages you to provide feedback on your experiences.

```
CREATE EXTERNAL TABLE ice_3 (i INT, s STRING) STORED BY ICEBERG TBLPROPERTIE
S ('format-version'='3');

INSERT INTO ice_3 VALUES (1, 'one'), (2, 'two'), (3, 'three');

DELETE FROM ice_3 WHERE i = 2;
```

Hive or Impala syntax

```
delete from tablename [where expression]
```

```
delete joined_tablename from [joined_tablename, joined_tablename2, ...] [ wh
ere expression ]
```

Hive or Impala examples

```
create external table tbl_ice(a int, b string, c int) stored by iceberg tblp
roperties ('format-version'='2');

insert into tbl_ice values (1, 'one', 50), (2, 'two', 51), (3, 'three', 5
2), (4, 'four', 53), (5, 'five', 54), (111, 'one', 55), (333, 'two', 56);

delete from tbl_ice where a <= 2,1;
```

The following example deletes 0, 1, or more rows of the table. If col1 is a primary key, 0 or 1 rows are deleted:

```
delete from ice_table where col1 = 100;
```

Describe table metadata feature

You can use certain Hive and Impala show and describe commands to get information about table metadata. You can also query metadata tables.

Note: Impala does not support COMPUTE INCREMENTAL STATS for Iceberg tables. If you run this command, Impala defaults to performing a complete COMPUTE STATS operation on the entire table.

For large Iceberg tables where a complete scan is too resource-intensive, use the TABLESAMPLE clause to compute statistics on a subset of the data.

The following table lists SHOW and DESCRIBE commands supported by Hive and Impala:

Command Syntax	Description	SQL engine support
SHOW CREATE TABLE table_name	Displays the schema that created the table.	Hive and Impala
SHOW FILES IN table_name	Lists the files related to the table.	Impala
SHOW PARTITIONS table_name	Returns the Iceberg partition spec, just the column information, not actual partitions or files.	Impala

Command Syntax	Description	SQL engine support
DESCRIBE [EXTENDED] table_name	EXTENDED: Displays all the metadata for the table in Thrift serialized form, which is useful for debugging.	Hive and Impala
DESCRIBE [FORMATTED] table_name	FORMATTED: Displays the metadata in tabular format.	Hive
DESCRIBE HISTORY table_name [BET WEEN timestamp1 AND timestamp2]	Limits the output history to a period of time, optionally.	Impala
SHOW FILES IN [db_name.]table_name [PARTITION (key_col_expression [, ...])]	Lists the data files related to the table or a specific partition.	Impala

Hive example

```
DESCRIBE t;
```

Hive output includes the following information:

col_name	data_type	comment
x	int	
y	int	
	NULL	NULL
# Partition Transform Information	NULL	NULL
# col_name	transform_type	NULL
y	IDENTITY	NULL

The output of DESCRIBE HISTORY includes the following columns about the snapshot. The first three are self-explanatory. The is_current_ancestor column value is TRUE if the snapshot is the ancestor of the table:

- creation_time
- snapshot_id
- parent_id
- is_current_ancestor

Impala examples

```
DESCRIBE HISTORY ice_t FROM '2022-01-04 10:00:00';
DESCRIBE HISTORY ice_t FROM now() - interval 5 days;
DESCRIBE HISTORY ice_t BETWEEN '2022-01-04 10:00:00' AND '2022-01-05 10:00:0
0';
```

Drop table feature

The syntax you use to create the table determines the default behavior when you drop the Iceberg table from Hive or Impala.

If you use CREATE TABLE, the external.table.purge flag is set to true. When the table is dropped, the contents of the table directory (actual data) are removed. If you use CREATE EXTERNAL TABLE from Hive, the external.table.purge flag is set to false. Dropping a table purges the schema only. The actual data is not removed. You can explicitly set the external.table.purge property to true to drop the data as well as the schema.

To prevent data loss during migration of a table to Iceberg, do not drop or move the table during migration. Exception: If you set the table property 'external.table.purge'='FALSE', no data loss occurs if you drop the table.

Hive or Impala syntax

```
DROP TABLE [IF EXISTS] table_name
```

Hive or Impala example

```
ALTER TABLE t SET TBLPROPERTIES('external.table.purge'='true');
DROP TABLE t;
```

Related Information

Create table feature

Drop partition feature

You can easily remove a partition from an Iceberg partition using an alter table statement from Hive or Impala.

Removing a partition does not affect the table schema. The column is not removed from the schema.

Prerequisites

- The filter expression in the drop partition syntax below is a combination of, or at least one of, the following predicates:
  - Minimum of one binary predicate
  - IN predicate
  - IS NULL predicate
- The argument of the predicate in the filter expression must be a partition transform, such as 'YEARS(col-name)' or 'column'.
- Any non-identity transform must be included in the select statement. For example, if you partition a column by days, the filter must be days.

Hive or Impala syntax

```
alter table table-name drop partition (<filter expression>)
```

The following operators are supported in the predicate: =, !=,<, >, <=, >=

Hive or Impala example

```
CREATE TABLE ice_table PARTITIONED BY SPEC (day(d)) STORED BY ICEBERG TBLPRO
PERTIES ('format-version'='2') AS SELECT x, ts, d FROM source_t;

INSERT INTO ice_table(d) VALUES ('2024-04-30');

ALTER TABLE ice_table DROP PARTITION (day(d) = '2024-04-30');
```

Expire snapshots feature

You can expire snapshots that Iceberg generates when you create or modify a table. During the lifetime of a table the number of snapshots of the table accumulate. You learn how to remove snapshots you no longer need.

Cloudera recommends periodically expiring snapshots to delete data files that are no longer needed and to reduce the size of table metadata. Each write to an Iceberg table from Hive creates a new snapshot, or version, of a table. Snapshots accumulate until expired.

You can expire snapshots based on the following conditions:

- (For Hive and Impala) All snapshots older than a timestamp or timestamp expression
- (For Hive) A snapshot having a given ID
- (For Hive) Snapshots having IDs matching a given list of IDs
- (For Hive) Snapshots within the range of two timestamps

You can keep snapshots you are likely to need, for example recent snapshots, and expire old snapshots. For example, you can keep daily snapshots for the last 30 days, then weekly snapshots for the past year, then monthly snapshots for the last 10 years. You can remove specific snapshots to meet GDPR “right to be forgotten” requirements.

Hive syntax

```
ALTER TABLE <table Name> EXECUTE EXPIRE_SNAPSHOTS(<timestamp expression>)

ALTER TABLE <table Name> EXECUTE EXPIRE_SNAPSHOTS('<Snapshot Id>')

ALTER TABLE <table Name> EXECUTE EXPIRE_SNAPSHOTS('<Snapshot Id1>,<Snapshot
 Id2>... ')

ALTER TABLE <table Name> EXECUTE EXPIRE_SNAPSHOTS BETWEEN (<timestamp exp
ression>) AND (<timestamp expression>)
```

Hive example

The first example removes snapshots having a timestamp older than August 15, 2022 1:50 pm. The second example removes snapshots from 10 days ago and before.

```
ALTER TABLE ice_11 EXECUTE EXPIRE_SNAPSHOTS('2022-08-15 13:50:00');
ALTER TABLE ice_t EXECUTE EXPIRE_SNAPSHOTS(CURRENT_TIMESTAMP - interval 10
days);
```

Impala syntax

```
ALTER TABLE <table Name> EXECUTE EXPIRE_SNAPSHOTS(<timestamp expression>)
```

Impala example

The first example removes snapshots having a timestamp older than August 15, 2022 1:50 pm. The second example removes snapshots from 10 days ago and before.

```
ALTER TABLE ice_11 EXECUTE EXPIRE_SNAPSHOTS('2022-08-15 13:50:00');
ALTER TABLE ice_t EXECUTE EXPIRE_SNAPSHOTS(now() - interval 10 days);
```

Preventing snapshot expiration

You can prevent expiration of recent snapshots by configuring the history.expire.min-snapshots-to-keep table property. You can use the alter table feature to set a property. The history.expire.min-snapshots-to-keep property refers to a number of snapshots, not a time delta. For example, assume you always want to keep all snapshots of your table for the last 24 hours. You configure history.expire.min-snapshots-to-keep as a safety mechanism to enforce this. If your table receives only one modification (insert / update / merge) per hour, then setting history.expire.min-snapshots-to-keep = 24 is sufficient to meet your requirement. However, if your table was consistently receiving updates every minute, then the last 24 hour period would entail 1440 snapshots, and the history.expire.min-snapshots-to-keep setting would need to be configured appropriately.

Table data and orphan maintenance

The contents of the table directory (actual data) might, or might not, be removed when you drop the table. An orphan data file can remain when you drop an Iceberg table, depending on the external.table.purge flag table property. An orphaned data file is one that has contents in the table directory, but no snapshot.

Expiring a snapshot does not remove old metadata files by default. You must clean up metadata files using writ e.metadata.delete-after-commit.enabled=true and write.metadata.previous-versions-max table properties. For more information, see "Iceberg table properties" below. Setting this property controls automatic metadata file removal after metadata operations, such as expiring snapshots or inserting data.

Related Information

Expiring snapshots

Iceberg table properties

Insert table data feature

From Hive and Impala, you can insert data into Iceberg tables using the standard INSERT INTO a single table. INSERT statements work for V1 and V2 tables.

You can replace data in the table with the result of a query. To replace data, Hive and Impala dynamically overwrite partitions that have rows returned by the SELECT query. Partitions that do not have rows returned by the SELECT query, are not replaced. Using INSERT OVERWRITE on tables that use the BUCKET partition transform is not recommended. Results are unpredictable because dynamic overwrite behavior would be too random in this case.

From Hive, Cloudera also supports inserting into multiple tables as a technical preview; however, this operation is not atomic, so data consistency of Iceberg tables is equivalent to that of Hive external tables. Changes within a single table will remain atomic.

Inserting, deleting, or updating table data generates a snapshot. A new snapshot corresponds to a new manifest list. Manifest lists are named snap-*.avro.

Iceberg specification defines sort orders. At this point, Hive doesn't support defining sort orders. But if there are sort orders defined by using other engines Hive can utilize them on write operations. For more information about sorting, see sort orders specification.

Hive or Impala syntax

```
INSERT INTO TABLE tablename VALUES values_row [, values_row ...]

INSERT INTO TABLE tablename1 select_statement1 FROM tablename2

INSERT OVERWRITE TABLE tablename1 select_statement1 FROM tablename2
```

Hive or Impala examples

```
CREATE TABLE ice_10 (i INT, s STRING, b BOOLEAN) STORED BY ICEBERG;
INSERT INTO ice_10 VALUES (1, 'asf', true);
CREATE TABLE ice_11 (i INT, s STRING, b BOOLEAN) STORED BY ICEBERG;
INSERT INTO ice_11 VALUES (2, 'apache', false);
INSERT INTO ice_11 SELECT * FROM ice_10;
SELECT * FROM ice_11;
INSERT OVERWRITE ice_11 SELECT * FROM ice_10;
```

Hive example

```
FROM customers
    INSERT INTO target1 SELECT customer_id, first_name;
```

```
    INSERT INTO target2 SELECT last_name, customer_id;
```

Load data inpath feature

From Hive or Impala, you can load Parquet or ORC data from a file in a directory on your file system or object store into an Iceberg table. For Impala, you might need to set the mem_limit or pool configuration (max-query-mem-limit, min-query-mem-limit) to accommodate the load.

Hive syntax

```
LOAD DATA [LOCAL] INPATH '<path to file>' [OVERWRITE] INTO TABLE tablename;
```

Hive example

```
LOAD DATA LOCAL INPATH 's3a://testbucket/files/part.orc' INTO TABLE ice_orc;

LOAD DATA LOCAL INPATH 's3a://testbucket/files/part.orc' OVERWRITE INTO TA
BLE ice_orc;
```



Note:

- Specifying the LOCAL keyword looks for a file path in the local file system.
- The OVERWRITE keyword deletes the records in the target table and replaces it with the files specified in the filepath, else, the command adds to the existing content in the target table.

Impala syntax

```
LOAD DATA INPATH '<path to file>' INTO TABLE tablename;
```

Impala example

In this example, you create a table using the LIKE clause to point to a table stored as Parquet. This is required for Iceberg to infer the schema. You also load data stored as ORC.

```
CREATE TABLE test_iceberg LIKE my_parquet_table STORED AS ICEBERG;
SET MEM_LIMIT=1GB;
LOAD DATA INPATH 's3a://testbucket/files/my_data.parq' INTO TABLE test_ice
berg;

LOAD DATA INPATH 's3a://testbucket/files/my_data.orc' INTO TABLE test2_ic
eberg;
```

Load or replace partition data feature

There is no difference in the way you insert data into a partitioned or unpartitioned Iceberg table.

Working with partitions is easy because you write the query in the same way for the following operations:

- Insert into, or replace, an unpartitioned table
- Insert into, or replace, an identity partitioned table
- Insert into, or replace, a transform-partitioned table

Do not use INSERT OVERWRITE on tables that went through partition evolution. Truncate such tables first, and then INSERT the tables.

Impala example

```
CREATE TABLE ice_12 (i int, s string, t timestamp, t2 timestamp) STORED BY I
CEBERG;
```

Hive or Impala examples

```
INSERT INTO ice_12 VALUES (42, 'impala', now(), to_date(now()));
INSERT OVERWRITE ice_t VALUES (42, 'impala', now(), to_date(now()));
```

Materialized view feature

Using a materialized view can accelerate query execution. Creating a materialized view on top of Iceberg tables in Cloudera can further accelerate the performance. You can create a materialized view of an Iceberg V1 or V2 table based on an existing Hive or Iceberg table.

The materialized view is stored in Hive ACID or Iceberg format. Materialized view source tables either must be native ACID tables or must support table snapshots. Automatic rewriting of a materialized view occurs under the following conditions:

- The view definition contains the following operators only:
  - TableScan
  - Project
  - Filter
  - Join(inner)
  - Aggregate
- Source tables are native ACID or Iceberg v1 or v2
- The view is not based on time travel queries because those views do not have up-to-date data by definition.

Hive example

The following example creates a materialized view of an Iceberg table from Hive.

```
drop table if exists tbl_ice;
create external table tbl_ice(a int, b string, c int) stored by iceberg st
ored as orc tblproperties ('format-version'='2');

create materialized view mat1 as
select b, c from tbl_ice for system_version as of 5422037307753150798;
```

The following example creates a materialized view of two Iceberg tables. Joined tables must be in the same table format, either Iceberg or Hive ACID.

```
drop table if exists tbl_ice2;
create table tbl_ice2(a int, b string, c int) stored as orc TBLPROPERTIES ('
transactional'='true');
INSERT INTO tbl_ice2 VALUES (2, 'apache', 3);

drop table if exists tbl_ice;
create external table tbl_ice(a int, b string, c int) stored by iceberg sto
red as orc tblproperties ('format-version'='2');
INSERT INTO tbl_acid VALUES (4, 'iceberg', 5);
create materialized view mat1 as
select tbl_ice2.b, tbl_ice2.c from tbl_ice join tbl_ice2 on tbl_ice.a = tb
l_ice2.a;
```

The following example uses explain to examine a materialized view and then creates a materialized view of an Iceberg V1 table from Hive.

```
drop materialized view if exists mat1;
drop table if exists tbl_ice;

create table tbl_ice(a int, b string, c int) stored by iceberg stored as orc
 tblproperties ('format-version'='1');
insert into tbl_ice values (1, 'one', 50), (2, 'two', 51), (3, 'three', 52),
 (4, 'four', 53), (5, 'five', 54);
explain create materialized view mat1 stored by iceberg stored as orc tblpr
operties ('format-version'='1') as
select tbl_ice.b, tbl_ice.c from tbl_ice where tbl_ice.c > 52;

create materialized view mat1 stored by iceberg stored as orc tblproperties
 ('format-version'='1') as
select tbl_ice.b, tbl_ice.c from tbl_ice where tbl_ice.c > 52;
```

Related Information

Materialized view rebuild feature

Materialized view rebuild feature

Updates to materialized view contents when new data is added to the underlying table are critical; otherwise, queries can return stale data.

An update can occur under the following conditions:

- As a row-level incremental rebuild of the view after inserting data into a table

  Source tables can be Iceberg V2 or Hive full ACID.
- As a full rebuild of the view

A full rebuild can be expensive. An incremental rebuild updates only the affected parts of the materialized view, decreasing rebuild step execution time.

An incremental rebuild occurs automatically when you insert (append) data into a source table; otherwise, after you make some other type of change, for example a delete, you must manually start a full rebuild.

You use the ALTER command to manually start a full rebuild of the materialized view from Hive as follows:

Hive syntax

```
ALTER MATERIALIZED VIEW <name of view> REBUILD;
```

Full rebuild example

In this example, first you set required properties. Next, you create Iceberg tables, a V1 table and a V2 table, from Hive. You insert data into the tables and create a materialized view of the joined tables. You insert some new values into one of the source tables, rendering the materialized view stale. Finally, you rebuild the materialized view using explain cbo to show the rebuild plan The rebuild plan indicates a full rebuild will occur, which means the definition query will be executed.

```

drop table if exists tbl_ice;
drop table if exists tbl_ice_v2;
create external table tbl_ice(a int, b string, c int) stored by iceberg sto
red as orc tblproperties ('format-version'='1');
create external table tbl_ice_v2(d int, e string, f int) stored by iceberg
stored as orc tblproperties ('format-version'='2');
```

```
insert into tbl_ice_v2 values (1, 'one v2', 50), (4, 'four v2', 53), (5, 'f
ive v2', 54);

create materialized view mat1 as
select tbl_ice.b, tbl_ice.c, tbl_ice_v2.e from tbl_ice
join tbl_ice_v2 on tbl_ice.a=tbl_ice_v2.d where tbl_ice.c > 52;
group by tbl_ice.b tbl_ice.c;

-- view should be empty
select * from mat1;

-- view is up-to-date, use it
explain cbo
select tbl_ice.b, tbl_ice.c, tbl_ice_v2.e from tbl_ice join tbl_ice_v2 on
tbl_ice.a=tbl_ice_v2.d where tbl_ice.c > 52;

-- insert some new values to one of the source tables
insert into tbl_ice values (1, 'one', 50), (2, 'two', 51), (3, 'three', 52),
 (4, 'four', 53), (5, 'five', 54);

-- view is outdated, cannot be used
explain cbo
select tbl_ice.b, tbl_ice.c, tbl_ice_v2.e from tbl_ice join tbl_ice_v2 on tb
l_ice.a=tbl_ice_v2.d where tbl_ice.c > 52;

explain cbo
alter materialized view mat1 rebuild;

-- view should contain data
select * from mat1;

-- view is up-to-date again, use it
explain cbo
select tbl_ice.b, tbl_ice.c, tbl_ice_v2.e from tbl_ice join tbl_ice_v2 on tb
l_ice.a=tbl_ice_v2.d where tbl_ice.c > 52;
group by tbl_ice.b tbl_ice.c;
```

Automatic query rewrite example

In this example, you create an Iceberg table, insert some values, and create the materialized view. The view is partitioned using a partition specification and stored in the Iceberg ORC format. The v1 format version is specified in this example (the v2 format is also supported). You then look at a description of the view and see that the query rewrite option is enabled by default. An automatic incremental rebuild is possible when this option is enabled.

```
   drop table if exists tbl_ice;
   create external table tbl_ice(a int, b string, c int) stored by iceberg
 stored as orc tblproperties ('format-version'='1');

   -- insert some new values into one of the source tables

   insert into tbl_ice values (1, 'one', 50), (2, 'two', 51), (3, 'three', 5
2), (4, 'four', 53), (5, 'five', 54);

   explain
   create materialized view mat1 partitioned on spec (bucket(16, b), trunc
ate(3, c)) stored by Iceberg stored as orc tblproperties(‘format-version’=’1
’)as
   select tbl_ice.b, tbl)ice.c from tbl_ice where tbl_ice.c > 52;

   -- the output query plan query indicates a rewrite is enabled
```

```
   POSTHOOK: query: explain
   create materialized view mat1 …
   …
   STAGE PLANS
  …
-- In stage one, the materialized view is created by calling the Iceberg
API to create the table object.
rewrite enabled
…
```

You use the DESCRIBE command to see the output query plan, which shows details about the view, including if it can be used in automatic query rewrites.

```
-- check the materialized view details
    describe formatted mat1;
    …
    #col_name     data_type      comment
    b             string
    c             int

    #Partition Transform Information
    #col_name     transform_type
    b             bucket(16)
    c             TRUNCATE[3]


    #detailed table information
    …
    Table_type:    MATERIALIZED_VIEW
    Table Parameters:
    …
    current-snapshot-id   563939E424367334713
    …
    metadata_location     …
    …
    Table_type            Iceberg
    …
    #Materialized View Information
    Original Query:  …
    Expanded Query:  …
    Rewrite Enabled: Yes
```

With the query rewrite option enabled, you insert data into the source table, and incremental rebuild occurs automatically. You do not need to rebuild the view manually before running queries.

Related Information

Materialized view feature

Merge feature

From Hive and Impala, you can perform actions on an Iceberg table based on the results of a join between a target and source v2 Iceberg table.

The MERGE statement supports multiple WHEN clauses, where each clause can specify actions like UPDATE, DELETE, or INSERT. Actions are applied based on the join conditions defined between the source and target tables.

Hive and Impala syntax

```
MERGE INTO <target table> AS T USING <source expression/table> AS S
ON <boolean expression1>
```

```
WHEN MATCHED [AND <boolean expression2>] THEN UPDATE SET <set clause list>
WHEN MATCHED [AND <boolean expression3>] THEN DELETE
WHEN NOT MATCHED [AND <boolean expression4>] THEN INSERT VALUES <value list>
```

Hive and Impala example

Use the MERGE INTO statement to update an Iceberg table based on a staging table:

```
MERGE INTO customer USING new_customer_stage source ON source.id = customer.
id
    WHEN MATCHED THEN UPDATE SET name = source.name, state = source.new_stat
e
    WHEN NOT MATCHED THEN INSERT VALUES (source.id, source.name, source.s
tate);
```

Create an Iceberg table and merge it with a non-Iceberg table.

```
create external table target_ice(a int, b string, c int) partitioned by spec
 (bucket(16, a), truncate(3, b)) stored by iceberg stored as orc tblproperti
es ('format-version'='2');
create table source(a int, b string, c int);
...

merge into target_ice as t using source src ON t.a = src.a
when matched and t.a > 100 THEN DELETE
when matched then update set b = 'Merged', c = t.c + 10
when not matched then insert values (src.a, src.b, src.c);
```

The Impala MERGE INTO statement supports the following capabilities:

- WHEN NOT MATCHED BY SOURCE merge clause - Useful in situations when a source table's rows do not match the target table rows. For example:

  ```
  MERGE INTO target_ice t using source s on t.id = s.id
  WHEN NOT MATCHED BY SOURCE THEN UPDATE set t.column = "a";
  ```

- Supports INSERT * syntax for the WHEN NOT MATCHED clause and UPDATE SET * syntax for the WHEN MATCHED clause.

  INSERT * enumerates all expressions from the source table or subquery to simplify inserting for target tables with large number of columns. The semantics is the same as the regular WHEN NOT MATCHED THEN INSERT clause.

  UPDATE SET * creates assignments for each target table column by enumerating the table columns and assigning source expressions by index.

Migrate Hive table to Iceberg feature

Cloudera supports Hive table migration from Hive to Iceberg tables using ALTER TABLE to set the table properties.

The topic "Test driving Iceberg from Impala" shows how to create Iceberg tables from Impala tables.



Note: Do not drop or move the old table during a migration operation. Doing so will delete the data files of the old and new tables. Exception: If you set the table property 'external.table.purge'='FALSE', no data loss occurs when you drop the table.

In-place table migration process

In-place table migration saves time generating Iceberg tables. There is no need to regenerate data files. Only metadata, which points to source data files, is regenerated.

To convert a Hive table to an Iceberg V1 table, use the following syntax:

```
ALTER TABLE table_name CONVERT TO ICEBERG;
```

To convert a Hive table to an Iceberg V2 table, you must run two queries. Use the following syntax:

```
ALTER TABLE table_name CONVERT TO ICEBERG

ALTER TABLE table_name SET TBLPROPERTIES ('format-version' = '2'
    ...)
```

In-place table migration saves time generating Iceberg tables. There is no need to regenerate data files. Only metadata, which points to source data files, is regenerated.

Securing Iceberg table data

Learn how to prevent Hive and Impala from reading data files from an Iceberg table that are outside of the table location.

An unauthorized user who knows the underlying file layout of a table can gain access to the data in an Iceberg table in the following way:

User1 owns a table, table1, which User2 does not have permission to read. However, User2 could execute an attack as follows: User2 creates a new table, table2, to which they have access rights. User2 then modifies the metadata files of their own table (table2) to reference data files from User1's table (table1), effectively including these files as part of table2. By accessing their own table (table2), User2 can then read the data files that belong to User1's table (table1).

You can prevent this by configuring Hive and Impala to prevent reading data files that are outside of the Iceberg table location.

Hive

Add the following property to HiveServer2 Advanced Configuration Snippet (Safety Valve) for hive-site.xml and set the value to "true":

```
hive.iceberg.allow.datafiles.in.table.location.only
```

Impala

Ensure that the following Impala startup flag is set to "true". The value of this flag is set to true by default.

```
iceberg_allow_datafiles_in_table_location_only
```

Flexible partitioning

Iceberg partition evolution, which is a unique Iceberg feature, and the partition transform feature, greatly simplify partitioning tables and changing partitions.

Partitions based on transforms are stored in the Iceberg metadata layer, not in the directory structure. You can change the partitioning completely, or just refine existing partitioning, and write new data based on the new partition layout--no need to rewrite existing data files. For example, change a partition by month to a partition by day.

Use partition transforms, such as IDENTITY, TRUNCATE, BUCKET, YEAR, MONTH, DAY, HOUR. Iceberg solves scalability problems caused by having too many partitions. Partitioning can also include columns with a large number of distinct values. Partitioning is hidden in the Iceberg metadata layer, eliminating the need to explicitly write partition columns (YEAR, MONTH for example) or to add extra predicates to queries for partition pruning.

```
SELECT * FROM tbl WHERE ts = '2023-04-21 20:56:08'
AND YEAR = 2023 AND MONTH = 4 AND DAY = 21
```

Year, month, and day can be automatically extracted from '2023-04-21 20:56:08' if the table is partitioned by DAY(ts)

Partition evolution feature

Partition evolution means you can change the partition layout of the table without rewriting existing data files. Old data files can remain partitioned by the old partition layout, while newly added data files are partitioned based on the new layout.

You can use the ALTER TABLE SET PARTITION SPEC statement to change the partition layout of an Iceberg table. A change to the partition spec results in a new metadata.json and a commit, but does not create a new snapshot.

The VOID transform might be used to replace the transform in an existing partition field so that the field is effectively dropped in v1 tables.

Hive or Impala syntax

```
ALTER TABLE table_name SET PARTITION SPEC ([col_name][, spec(value)][, spec(
value)]...)]
```

spec: The specification for a transform listed in the next topic, "Partition transform feature".

Hive or Impala examples

```
ALTER TABLE t
SET PARTITION SPEC ( TRUNCATE(5, level), HOUR(event_time),
BUCKET(15, message), price);
ALTER TABLE ice_p
SET PARTITION SPEC (VOID(i), VOID(d), TRUNCATE(3, s), HOUR(t), i);
```

Related Information

Partition transform feature

Partition transform feature

From Hive or Impala, you can use one or more partition transforms to partition your data. Each transform is applied to a single column. Identity-transform means no transformation; the column values are used for partitioning. The other transforms apply a function to the column values and the data is partitioned by the transformed values.

Using CREATE TABLE ... PARTITIONED BY you create identity-partitioned Iceberg tables. Identity-partitioned Iceberg tables are similar to the Hive or Impala partitioned tables, which are stored in the same directory structure as the data files. Iceberg stores the partitioning columns of identity-partitioned Iceberg tables in a different directory structure from the data files if the tables are migrated to Iceberg from Hive external tables. Iceberg handles the tables and files regardless of the location.

Hive and Impala support Iceberg advanced partitioning through the PARTITION BY SPEC clause. Using this clause, you can define the Iceberg partition fields and partition transforms.

The following table shows the available partition transforms and their corresponding transform specifications:

Partition transform	Transform specification	Supported SQL engine
Partition by year	years(time_stamp) | year(time_stamp)	Hive and Impala
Partition by month	months(time_stamp) | month(time_stamp)	Hive and Impala
Partition by a date value stored as int (dateint)	days(time_stamp) | date(time_stamp)	Hive
Partition by hours	hours(time_stamp)	Hive
Partition by a dateint in hours	date_hour(time_stamp)	Hive
Partition by hashed value mod N buckets	bucket(N, col)	Hive and Impala

Partition transform	Transform specification	Supported SQL engine
Partition by value truncated to L, which is a number of characters	truncate(L, col)	Hive and Impala
Partition by hashed value mod N buckets	bucket(N, col)	Hive and Impala
Partition by value truncated to L, which is a number of characters	truncate(L, col)	Hive and Impala
Partition by value	identity(col)	Impala
Partition by null values	void(col)	Impala

The BUCKET, TRUNCATE, IDENTITY, and VOID transforms now support the BINARY type.

The VOID transform might be used to replace the transform in an existing partition field so that the field is effectively dropped in v1 tables.

Strings are truncated to length L. Integers and longs are truncated to bins. For example, truncate(10, i) yields partitions 0, 10, 20, 30 …

The idea behind transformation partition by hashed value mod N buckets is the same as hash bucketing for Hive tables. A hashing algorithm calculates the bucketed column value (modulus). For example, for 10 buckets, data is stored in column value % 10, ranging from 0-9 (0 to n-1) buckets.

You use the PARTITIONED BY SPEC clause to partition a table by an identity transform.

Hive syntax

```
CREATE [EXTERNAL] TABLE [IF NOT EXISTS] [db_name.]table_name
  [(col_name data_type][, time_stamp TIMESTAMP] )]
  [PARTITIONED BY SPEC([col_name][, spec(value)][, spec(value)]...)]
  [STORED AS file_format]
   STORED BY ICEBERG
  [TBLPROPERTIES (property_name=property_value, ...)]
```

Where spec (value) represents one or more of the following transforms:

- YEARS(col_name)
- MONTHS(col_name)
- DAYS(col_name)
- BUCKET(bucket_num,col_name)
- TRUNCATE(length, col_name)

Impala syntax

```
CREATE TABLE [IF NOT EXISTS] [db_name.]table_name
  [(col_name data_type, ... )]
  [PARTITIONED BY SPEC([col_name][, spec(value)][, spec(value)]...)]
   STORED (AS | BY) ICEBERG
  [TBLPROPERTIES (property_name=property_value, ...)]
```

Where spec(value) represents one or more of the following transforms:

- YEARS(col_name)
- MONTHS(col_name)
- DAYS(col_name)
- BUCKET(bucket_num,col_name)
- TRUNCATE(length, col_name)

Hive example

The following example creates a top level partition based on column i, a second level partition based on the hour part of the timestamp, and a third level partition based on the first 1000 characters in column j.

```
CREATE EXTERNAL TABLE ice_3 (i INT, t TIMESTAMP, j BIGINT) PARTITIONED BY SP
EC (i, HOUR(t), TRUNCATE(1000, j)) STORED BY ICEBERG;
```

Impala examples

```
CREATE TABLE ice_13 (i INT, t TIMESTAMP, j BIGINT) PARTITIONED BY SPEC (i, H
OUR(t), TRUNCATE(1000, j)) STORED BY ICEBERG;
```

The following examples show how to use the PARTITION BY SPEC clause in a CREATE TABLE query from Impala.The same transforms are available in a CREATE EXTERNAL TABLE query from Hive.

```
CREATE TABLE ice_t(id INT, name STRING, dept STRING)
PARTITIONED BY SPEC (bucket(19, id), dept)
STORED BY ICEBERG
TBLPROPERTIES ('format-version'='2');
```

```
CREATE TABLE ice_ctas
PARTITIONED BY SPEC (truncate(1000, id))
STORED BY ICEBERG
TBLPROPERTIES ('format-version'='2')
AS SELECT id, int_col, string_col FROM source_table;
```

Related Information

Creating an Iceberg partitioned table

Create table feature

Partition evolution feature

Insert into/overwrite partition feature

From Hive you can insert into, or overwrite data in, Iceberg tables that are statically or dynamically partitioned.

You can also insert data into a table using static and dynamic partitions. INSERT OVERWRITE queries do not work on tables that have undergone partition evolution.

Static partition syntax

```
INSERT INTO|OVERWRITE TABLE name PARTITION(col = val) VALUES (...)
INSERT INTO|OVERWRITE TABLE name PARTITION(col = val) SELECT <expression>
```

Dynamic partition syntax

```
INSERT INTO|OVERWRITE TABLE name PARTITION(col) VALUES (...)
INSERT INTO|OVERWRITE TABLE name PARTITION(col) SELECT <expression>
```

Static and dynamic partition syntax

Insertion into the static partition must precede the insertion into the dynamic partition.

```
INSERT INTO|OVERWRITE TABLE name PARTITION(col1 = val, col2) VALUES (...)
INSERT INTO|OVERWRITE TABLE name PARTITION(col1 = val, col2) SELECT <expre
ssion>
```

Truncate partition feature

You can truncate Iceberg partitioning from Hive.

You specify columns in the TRUNCATE TABLE ... PARTITION command as shown in the following syntax. This command does not support partitioning on columns that have partition transforms other than identity-transform.

Hive syntax

```
TRUNCATE TABLE t PARTITION (parCol1 = Val1, parCol2 = Val2 ... );
```

file:/Users/nisha.sridhar/Sandbox/newco-docs/topics/iceberg-truncate-partition.xml

Query metadata tables feature

Apache Iceberg stores extensive metadata for its tables. From Hive and Impala, you can query the metadata tables as you would query a regular table. For example, you can use projections, joins, filters, and so on.

Iceberg metadata tables include information that is useful for efficient table maintenance (about snapshots, manifests, data, delete files, etc.) as well as statistics that help query engines plan and execute queries more efficiently (value count, min-max values, number of NULLs, etc.).

Note: Iceberg metadata tables are read-only. You cannot add, remove, or modify records in the tables. Also, you cannot drop or create new metadata tables.

For more information about the Apache Iceberg Iceberg metadata table types, see the *Apache Iceberg MetadataTableType enumeration*.

For more information about querying Iceberg metadata, see the *Apache Iceberg Spark documentation*.

The following sections describe how you can interact with and query Iceberg metadata tables:

List metadata tables

The SHOW METADATA TABLES statement lists all metadata tables belonging to an Iceberg table. You can also filter the tables according to a specific pattern.

Note: The SHOW METADATA TABLES statement is only available in Impala and is not supported in Hive.

Impala Syntax:

```
SHOW METADATA TABLES IN [database_name.]table_name [[LIKE] "pattern"];
```

Impala Example:

```
SHOW METADATA TABLES IN default.ice_table;

Output:
+----------------------+
| name                 |
+----------------------+
| all_data_files       |
| all_delete_files     |
| all_entries          |
| all_files            |
| all_manifests        |
| data_files           |
| delete_files         |
| entries              |
| files                |
```

```
| history              |
| manifests            |
| metadata_log_entries |
| partitions           |
| position_deletes     |
| refs                 |
| snapshots            |
+----------------------+
```

Query metadata tables

You can use the regular SELECT statement from Hive or Impala to query Iceberg metadata tables. To reference a metadata table, use the fully qualified path as shown in the syntax.

Hive or Impala Syntax:

```
SELECT … FROM database_name.table_name.metadata_table_name;
```

Hive or Impala Example:

```
SELECT * FROM default.ice_table.files;
```

You can select any subset of the columns or all of them using '*'. In comparison to regular tables, running a SELE CT * from Impala on metadata tables always includes complex-typed columns in the result. The Impala query option, EXPAND_COMPLEX_TYPES only applies to regular tables. However, Hive always includes complex columns irresepctive of whether SELECT queries are run on regular tables or metadata tables.

For Impala queries that have a mix of regular tables and metadata tables, a SELECT * expression where the sources are metadata tables always includes complex types, whereas for SELECT * expressions where the sources are regular tables, complex types are included only if the EXPAND_COMPLEX_TYPES query option is set to 'true'.

In the case of Hive, columns with complex types are always included.

You can also filter the result set using a WHERE clause, use aggregate functions such as MAX or SUM, JOIN metadata tables with other metadata tables or regular tables, and so on.

Example:

```
SELECT
    s.operation,
    h.is_current_ancestor,
    s.summary
FROM default.ice_table.history h
JOIN default.ice_table.snapshots s
  ON h.snapshot_id = s.snapshot_id
WHERE s.operation = 'append'
ORDER BY made_current_at;
```

Limitations

- Impala does not support the DATE and BINARY data types. NULL is returned instead of their actual values.
- Impala does not support unnesting collections from metadata tables.

DESCRIBE metadata tables

Like regular tables, Iceberg metadata tables have schemas that can be explored using the DESCRIBE statement. The DESCRIBE statement displays metadata about a table, such as the column names and their data types.

To reference the metadata table, use the fully qualified path as shown in the syntax.

Note: DESCRIBE FORMATTED|EXTENDED is not available for metadata tables. In Impala, using this statement results in an error whereas Hive displays the same output as the regular DESCRIBE statement.

Hive or Impala Syntax:

```
DESCRIBE database_name.table_name.metadata_table_name;
```

Hive or Impala Example:

```
DESCRIBE default.ice_table.history;
Output:
+---------------------+-----------+---------+----------+
| name                | type      | comment | nullable |
+---------------------+-----------+---------+----------+
| made_current_at     | timestamp |         | true     |
| snapshot_id         | bigint    |         | true     |
| parent_id           | bigint    |         | true     |
| is_current_ancestor | boolean   |         | true     |
+---------------------+-----------+---------+----------+
```

Related Information

Apache Iceberg MetadataTableType

Apache Spark documentation

Rollback table feature

In the event of a problem with your table, you can reset a table to a good state as long as the snapshot of the good table is available. You can roll back the table data based on a snapshot id or a timestamp.

When you modify an Iceberg table, a new snapshot of the earlier version of the table is created. When you roll back a table to a snapshot, a new snapshot is created. The creation date of the new snapshot is based on the Timezone of your session. The snapshot id does not change.

Hive and Impala syntax

```
ALTER TABLE test_table EXECUTE rollback(snapshotID);
ALTER TABLE test_table EXECUTE rollback('timestamp');
```

Hive and Impala examples

The following example rolls back to an earlier table, creating a new snapshot having a new creation date timestamp, but keeping the same snapshot id 3088747670581784990.

```
ALTER TABLE ice_t EXECUTE ROLLBACK(3088747670581784990);
```

The following example rolls the table back to the latest snapshot having a creation timestamp earlier than '2022-08-08 00:00:00'.

```
ALTER TABLE ice_7 EXECUTE ROLLBACK('2022-08-08 00:00:00')
```

Row-level operations

Hive supports the copy-on-write (COW) as well as merge-on-read (MOR) modes for handling Iceberg row-level updates and deletes. Impala supports only the MOR mode and will fail if configured for copy-on-write. Impala does support reading copy-on-write tables.

Depending on the COW or MOR setting, Hive performs updates and deletes to Iceberg tables as follows:

- COW: Hive creates a new version of files for each update/delete. Use COW when updating/deleting a large number of rows, or when reading data frequently.
- MOR (default): Updates/deletes are logged to delta files, which tends to be faster than creating new versions of the files. Later a compaction can eliminate the delete files and rewrite the affected data files.

Impala uses only the MOR method. Impala does not support copy-on-write and will fail if configured for copy-on-write. Impala does support reading copy on write tables.

Configuring Hive tables to support COW or MOR

You use the CREATE TABLE or ALTER commands to set either COW or MOR mode in table properties as shown in the following examples:

```
ALTER TABLE ice01 SET TBLPROPERTIES ('write.delete.mode'='copy-on-write');
ALTER TABLE ice01 SET TBLPROPERTIES ('write.update.mode'='copy-on-write');
```

When to use COW or MOR

Set either COW or MOR based on your use case and rate of data change. Consider the following advantages and disadvantages of the modes:

- MOR
  - Writes are efficient.
  - Reads are inefficient due to read amplification, but regularly scheduled compaction can reduce inefficiency.
  - A good choice when streaming.
  - A good choice when frequently writing or updating, such as running hourly batch jobs.
  - A good choice when the percentage of data change is low.

  COW
  - Reads are efficient.
  - A good choice for bulk updates and deletes, such as running a daily batch job.
  - Writes are inefficient due to write amplification, but the need for compaction is reduced.
  - A good choice when the percentage of data change is high.

Position or equality delete files

By default, Hive and Impala delete Iceberg V2 table data using position delete files. Hive and Impala can read equality deletes, which you might encounter in a table created and written to by Flink or another engine that supports writing equality deletes. Hive and Impala cannot write equality deletes.

Starting with Cloudera Runtime 7.3.2 SP1, Hive and Impala [Technical Preview] writes position deletes on Iceberg V3 tables as deletion vectors instead of position delete files. For more information, see Delete data feature.

Choose the MOR or COW mode based on your use case. The MOR mode keeps track of deleted rows, so it's expensive when you're making many deletions.

The position and equality deletes are summarized below:

Position delete: DELETE operations find the data records that need to be deleted, then write out their file path and position into delete files. Postponing the rewrite of the files speeds writes of updates/deletes. Read operations need to merge the data and delete files to retrieve the active rows.

Equality delete: DELETE operations write out column values (for example, primary keys) that identify the records to be deleted. If you already know the column values of the records you want to delete, then there is no need to read the whole table, which means equality deletes can be written faster than position deletes. Read operations need to merge the data and delete files to retrieve the active rows. Processing equality deletes is typically slower than processing position deletes.

Row lineage feature [Technical Preview]

Impala supports row lineage tracking on Iceberg V3 tables. Row lineage is an Iceberg V3 capability. Row lineage assigns a unique row identifier to each row and records which snapshot last updated a row. This metadata supports change data capture (CDC) and row provenance use cases.

Note: This feature is in Technical Preview and is not ready for production deployments. Cloudera recommends trying this feature in test or development environments and encourages you to provide feedback on your experiences.

To use row lineage from Impala, create an Iceberg V3 table by setting 'format-version'='3'. When Impala writes to the table, Impala maintains the row lineage metadata that the specification requires; you need not use new SQL syntax to enable row lineage.

Row lineage applies to newly written rows on V3 tables. It does not apply to rows updated using equality deletes.

Impala example

```
CREATE TABLE ice_3 (i INT, s STRING) STORED BY ICEBERG TBLPROPERTIES
  ('format-version'='3');

INSERT INTO ice_3 VALUES (1, 'one'), (2, 'two');
```

Puffin statistics feature

Impala supports writing Puffin statistics files for Iceberg V3 tables. Puffin statistics files store deletion vectors as sidecar files and provide a standardized container for column statistics metadata.

Puffin is a lightweight container format for arbitrary binary blobs. Iceberg Puffin statistics support is a prerequisite for Iceberg V3 capabilities in Impala.

To use Puffin statistics from Impala, create an Iceberg V3 table by setting 'format-version'='3'. When Impala performs supported V3 write operations, Impala writes Puffin files as required by the Iceberg V3 specification; you need not run a new SQL syntax to enable Puffin statistics writes.

For deletion vectors on V3 tables, see Delete data feature.

Impala example

```
CREATE TABLE ice_3 (i INT, s STRING) STORED BY ICEBERG TBLPROPERTIES
  ('format-version'='3');

INSERT INTO ice_3 VALUES (1, 'one'), (2, 'two');

DELETE FROM ice_3 WHERE i = 2;
```

Select Iceberg data feature

You can read Iceberg tables from Impala as you would any table. Joins, aggregations, and analytical queries, for example, are supported.

Impala supports reading V2 tables with position deletes.

Hive or Impala example

```
SELECT * FROM ice_t;
```

```
SELECT count(*) FROM ice_t i LEFT OUTER JOIN other_t b
ON (i.id = other_t.fid)
WHERE i.col = 42;
```

Schema evolution feature

You learn that the Hive or Impala schema changes when the associated Iceberg table changes; You see examples of changing the schema.

Although you can change the schema of your table over time, you can still read old data files because Iceberg uniquely identifies schema elements. A schema change results in a new metadata.json and a commit, but does not create a new snapshot.

The Iceberg table schema is synchronized with the Hive/Impala table schema; A change to the schema of the Iceberg table by an outside entity, such as Spark, changes the corresponding Hive/Impala table. You can change the Iceberg table using ALTER TABLE to make the following changes:

From Hive: dsdf

apples, tables and servers

- Add a column
- Replace a column
- Change a column type or its position in the table

From Impala:

- Add a column
- Rename a column
- Drop a column
- Change a column type

An unsafe change to a column type, which would require updating each row of the table for example, is not allowed; The following type changes are safe:

- int to long
- float to double
- decimal(P, S) to decimal(P', S) if precision is increased

You can drop a column by changing the old column to the new column.

Hive syntax

```
ALTER TABLE table_name ADD COLUMNS (col_name type[, …])
ALTER TABLE table_name CHANGE COLUMN col_old_name col_new_name type

ALTER TABLE table_name CHANGE COLUMN col_old_name col_new_name type [FIRST|
AFTER col_name] [existing_col_name
]
ALTER TABLE table_name REPLACE COLUMNS (col_name type)
```

Impala syntax

```
ALTER TABLE table_name ADD COLUMNS(col_name type[, …])

ALTER TABLE table_name CHANGE COLUMN col_old_name col_new_name type
ALTER TABLE table_name DROP COLUMN col_name
```

Hive examples

```
ALTER TABLE t ADD COLUMNS(message STRING, price DECIMAL(8,1));

ALTER TABLE t REPLACE COLUMNS (i int comment '...', a string, ...);
ALTER TABLE t CHANGE COLUMN col_x col_x DECIMAL (22, 3) AFTER col_y;
```

Impala examples

```
ALTER TABLE ice_12 ADD COLUMNS(message STRING, price DECIMAL(8,1));
ALTER TABLE ice_12 DROP COLUMN i;

ALTER TABLE ice_12 CHANGE COLUMN s str STRING;
```

Schema inference feature

From Hive or Impala, you can base a new Iceberg table on a schema in a Parquet file. You see a difference in the Hive and Impala syntax and examples. kilometre -er/-re manoeuvre

From Hive, you must use FILE in the CREATE TABLE LIKE ... statement. From Impala, you must omit FILE in the CREATE TABLE LIKE … statement. The column definitions in the Iceberg table are inferred from the Parquet data file when you create a table such as Parquet from Hive or Impala. Set the following table property for creating the table:

```
hive.parquet.infer.binary.as = <value>
```

Where <value> is binary (the default) or string.

This property determines the interpretation of the unannotated Parquet binary type. Some systems expect binary to be interpreted as string. flavour

Hive syntax

```
CREATE [EXTERNAL] TABLE [IF NOT EXISTS] [db_name.]table_name LIKE FILE PARQU
ET 'object_storage_path_of_parquet_file'
[PARTITIONED BY [SPEC]([col_name][, spec(value)][, spec(value)]...)]]
[STORED AS file_format]
STORED BY ICEBERG
[TBLPROPERTIES (property_name=property_value, ...)]
```

Impala syntax

```
CREATE TABLE [IF NOT EXISTS] [db_name.]table_name LIKE PARQUET 'object_stora
ge_path_of_parquet_file'
[PARTITIONED BY [SPEC]([col_name][, spec(value)][, spec(value)]...)]]
STORED (AS | BY) ICEBERG
[TBLPROPERTIES (property_name=property_value, ...)]
```

Hive example

```
CREATE TABLE ctlf_table LIKE FILE PARQUET 's3a://testbucket/files/schema.par
q'
STORED BY ICEBERG;
```

Impala example

```
CREATE TABLE ctlf_table LIKE PARQUET 's3a://testbucket/files/schema.parq'
STORED BY ICEBERG;
```

Snapshot management

In addition to time travel queries, expiring a snapshot, and using a snapshot to rollback to a version of a table, you can also set any snapshot to be the current snapshot from Hive.

Hive syntax

```
ALTER TABLE TBL_ICEBERG_PART EXECUTE SET_CURRENT_SNAPSHOT (<snapshot ID>)
```

Hive example

```
ALTER TABLE TBL_ICEBERG_PART EXECUTE SET_CURRENT_SNAPSHOT (75212489901265493
11)
```

Branching feature

Branches are references to snapshots that have a lifecycle of their own. You can create a branch by basing the branch on a snapshot ID, a timestamp, or the state of your table. Using the SNAPSHOT RETENTION clause, you can create a branch that limits the number of snapshots of a table.

Iceberg branching is available in Hive and Spark. Iceberg branches are not supported in Impala.

List snapshots and branches of a table

The following syntax lists timestamps and IDs of snapshots of an Iceberg table. You can use the list of snapshots to create branches and tags.

```
SELECT * FROM <database>.<table name>.HISTORY
```

The following syntax lists the branches and tags of a table.

```
SELECT * from <database>.<table name>.REFS
```

Create a branch

Use either system version or system time in the following syntax from Hive to create a branch. Tables must be Iceberg V2 tables.

```
ALTER TABLE <table name> CREATE BRANCH <branch name> FOR SYSTEM_VERSION AS O
F <SNAPSHOT_ID>
ALTER TABLE <table name> CREATE BRANCH <branch name> FOR SYSTEM_TIME AS OF
'time_stamp' [expression]
```

If you do not have the ID or timestamp of a snapshot, you can also create a branch using the table name only and omitting the FOR clause

```
ALTER TABLE <table name> CREATE BRANCH <branch name>
```

This syntax creates a branch having the same state as the table.

Limit retained snapshots

When you create a branch, you can limit the number of snapshots a branch retains.

```
ALTER TABLE <table name> CREATE BRANCH <branch name> FOR SYSTEM_TIME AS OF <
timestamp> WITH SNAPSHOT RETENTION <integer limit> SNAPSHOTS;
```

Query a branch

From Hive, you can ingest data into an Iceberg branch using dot notation as you would a SQL table. The branch name prefix branch_ must be lowercase.

```
INSERT INTO TABLE <database name>.<table name>.branch_<branch name> VALUES (
<column name>[, <column name> ...]
```

You can use SQL syntax to read, update, and delete data in a branch.

```
SELECT <column name 1>, <column name 2>, ... FROM <database name>.<table nam
e>.branch_<branch name>
UPDATE TABLE <database name>.<table name>.branch_<branch name> SET <column
name>=<new value>, <column name>=<new value> ... WHERE <condition>

DELETE FROM <database name>.<table name>.branch_<branch name> WHERE <conditi
on>
```

Fast forward a branch

Fast forwarding a branch updates the state of one branch to another branch within its hierarchy. For example, you can fast-forward branch x to branch z as shown in the following example:

```
ALTER TABLE <table name> EXECUTE FAST-FORWARD 'x' 'z';
```

Branch x must be an ancestor of branch z. If you omit the second branch name, the named branch is fast-forwarded to the current branch.

Delete a branch

You can delete a branch related to a particular table, using the following syntax:

```
ALTER TABLE <table name> DROP BRANCH [IF EXISTS] <branch name>
```

Tagging feature

Tags identify snapshots you need for auditing and conforming to GDPR. You can tag a snapshot to help you track retention for a certain period of time.

Iceberg tagging is available in Hive and Spark. Iceberg tagging is not available in Impala.

Create a tag

You can create a tag based on SYSTEM_VERSION, SYSTEM_TIME, or the current branch. Tables must be Iceberg V2 tables.

```
ALTER TABLE <table name> CREATE TAG <tag name> FOR SYSTEM_VERSION AS OF <sna
pshot ID>

ALTER TABLE <table name> CREATE TAG <tag name> FOR SYSTEM_TIME AS OF '<ti
mestamp>'
```

```
ALTER TABLE <table name> CREATE TAG <tag name>
```

Query a tag

You can run SQL read queries on tags using the name of the tag as follows:

```
<database name>.<table name>.tag_<tag name>
```

For example:

```
SELECT * from mydb.mytable.tag_mytag;
```

Delete a tag

Use the following syntax to delete a tag.

```
ALTER TABLE <table name> DROP tag [IF EXISTS] <tag name>
```

Table repair feature

The repair_metadata() function provides a self-service recovery path to recover Iceberg tables that are inaccessible due to missing data files after manual file deletions in the underlying storage.

To maintain table integrity, you must always interact with Iceberg tables through the engine or the Iceberg API. For example, to remove partitions, you must use the DROP PARTITION statement instead of manually deleting directories from the file system, for example, S3, ADLS, or HDFS).

Impala can repair the table only if the missing files are data files. Impala cannot repair the table if there are missing delete files. If delete files are missing, the repair_metadata() function does not remove references to those files, and the table remains inaccessible.

When you delete files directly from the storage without updating the Iceberg metadata, the table becomes corrupted, causing the subsequent queries to fail with the following error message: Iceberg table [...] cannot be fully loaded due to unavailable files



Note: The repair_metadata() function does not restore the deleted content. It only cleans up the metadata references. Run this command only if you do not want to restore the original data files.

This issue occurs because the Iceberg manifest files still contain references to data files that no longer exist. If you cannot restore the missing files from a backup or filesystem trash, you can use the command. The following command synchronizes the metadata by removing references to the missing data files, making the table functional again:

Impala example

```
ALTER TABLE ice_tbl EXECUTE repair_metadata();
```

LIKE predicate pushdown feature

Impala pushes eligible SQL LIKE predicates on Iceberg tables to Iceberg expressions for file and partition pruning.

The Iceberg specification supports expression-based filtering through manifest statistics. Impala translates eligible LIKE predicates to Iceberg startsWith() and equal() expressions during query planning. This enables Iceberg to skip files and partitions that cannot contain matching rows.

You need not run this optimization with new SQL syntax or table properties. Impala applies it automatically when the LIKE pattern is eligible for pushdown.

Direct predicate pushdown

For patterns where wildcard behavior is straightforward, Impala translates the LIKE predicate directly to an Iceberg expression:

SQL LIKE pattern	Iceberg expression
LIKE 'abc%'	startsWith('abc')
LIKE 'pre_fix%'	startsWith('pre')
LIKE 'exact'	equal('exact')
LIKE 'asd\%'	equal('asd%')

Relaxed predicate pushdown

For complex patterns that contain literal content after a wildcard (for example, LIKE 'prefix%suffix' or LIKE 'd%d'), Impala uses a two-phase approach:

- Impala pushes a relaxed prefix predicate to Iceberg (for example, startsWith('prefix')) for coarse-grained file and partition pruning.
- Impala retains the full LIKE predicate and evaluates it on the surviving rows to return correct results.

Limitations

Impala cannot push down LIKE patterns that start with a wildcard, such as LIKE '%suffix' or LIKE '_prefix', because there is no leading literal prefix to translate.

For ALTER TABLE … DROP PARTITION and SHOW FILES IN … PARTITION, Impala supports simple prefix LIKE patterns (for example, s LIKE 'd%'). Patterns with literal content after a wildcard are rejected to prevent unintended partition operations. For example, DROP PARTITION (s LIKE 'd%d') returns an error similar to the following:

```
AnalysisException: Predicate 's LIKE 'd%d'' can only be partially converted
 to Iceberg expression
```

Impala example

```
-- Direct pushdown: prefix pattern on a partitioned column
SELECT count(*) FROM ice_part WHERE action LIKE 'c%';

-- Relaxed pushdown: literal content after wildcard
SELECT s FROM ice_tbl WHERE s LIKE 'test%value';
```

Time travel feature

From Hive or Impala, you can run point in time queries for auditing and regulatory workflows on Iceberg tables. Time travel queries can be time-based or based on a snapshot ID. seeing through and/or

Iceberg generates a snapshot when you create, or modify, a table. A snapshot stores the state of a table. You can specify which snapshot you want to read, and then view the data at that timestamp. In Hive, you can use projections, joins, and filters in time travel queries. You can add expressions to the timestamps, as shown in the examples. You can expire snapshots.

Snapshot storage is incremental and dependent on the frequency and scale of updates. By default, Hive and Impala use the latest snapshot. You can query an earlier snapshot of Iceberg tables to get historical information. Hive/Impala use the latest schema to query an earlier table snapshot even if it has a different schema.

to understand the state(s) of the engines, for example tell me something



Note: and/or

Hive or Impala syntax

```
SELECT * FROM table_name FOR SYSTEM_TIME AS OF 'time_stamp' [expression]

SELECT * FROM table_name FOR SYSTEM_VERSION AS OF snapshot_id [expression]
```

- wow! time_stamp

  The state of the Iceberg table at the time specified by the UTC timestamp.
- snapshot_id

  The ID of the Iceberg table snapshot from the history output!

Hive or Impala examples

```
SELECT * FROM t FOR SYSTEM_TIME AS OF '2021-08-09 10:35:57' LIMIT 100;

SELECT * FROM t FOR SYSTEM_VERSION AS OF 3088747670581784990 limit 100;
SELECT * from ice_11 FOR SYSTEM_TIME AS OF now() - interval 30 minutes;
```

Truncate table feature

Truncating an Iceberg table removes all rows from the table. A new snapshot is created. Truncation works for partitioned and unpartitioned tables.

Although the table data and the table and column stats are cleared, the old snapshots and their data files continue to exist to support time travel in the future.

Hive syntax

```
TRUNCATE table_name
```

Impala syntax

```
TRUNCATE [TABLE] table_name
```

Hive or Impala example

```
TRUNCATE t;
```

Update data feature

From Hive or Impala, you can update data from a V2 Iceberg table.

The Iceberg V2 format allows row-level modifictions using delete files. Row-level updates are supported. Hive and Impala write position to denote deleted or updated records. For information about position delete files, see the Delete data feature and Row-level operations topics.

Updating table data generates a new snapshot for the table.

You can use a WHERE clause in your UPDATE statement. For example:

```
update tbl_ice set b='Changed' where b in (select b from tbl_ice where a < 4
);
```

Hive or Impala evaluates rows from the target table against a WHERE clause, and updates all the rows that match the WHERE condition. The WHERE condition is similar to the WHERE condition used in SELECT.

Impala prerequisites

The target table should be an Iceberg V2 table. Therefore set the following table property during table creation, or use ALTER TABLE SET TBLPROPERTIES on existing Iceberg V1 tables: 'format-version' = '2'

You must write data and delete files in Parquet format. Set the following table properties to meet this prerequisite:

- 'write.format.default' = 'PARQUET'
- 'write.delete.format.default' = 'PARQUET'

Hive or Impala syntax

```
update tablename set column = value [, column = value ...] [where expres
sion]
```

Hive examples

```
create external table tbl_ice(a int, b string, c int) stored by iceberg tblp
roperties ('format-version'='2');
insert into tbl_ice values (1, 'one', 50), (2, 'two', 51), (3, 'three', 52),
 (4, 'four', 53), (5, 'five', 54), (111, 'one', 55), (333, 'two', 56);

update tbl_ice set b='Changed' where b in (select b from tbl_ice where a <
 4);
```

Updating a table from a non-Iceberg source

From Impala, you can use UPDATE FROM to update anIceberg table based on a source table or view that is another Iceberg or non-Iceberg table. For example:

```
UPDATE tbl_ice SET tbl_ice.one = t.one, tbl_ice.two = t.two, FROM tbl_ice, t
bl_hive one where tbl_ice.pk = one.pk;
```

Best practices for Iceberg in Cloudera

Based on large scale TPC-DS benchmark testing, performance testing and real-world experiences, Cloudera recommends several best practices when using Iceberg.

Follow these key best practices listed below when using Iceberg:

- Use Iceberg as intended for analytics.

  The table format is designed to manage a large, slow-changing collection of files. For more information, see the Iceberg spec.

- Increase parallelism to handle large manifest list files in Spark.

  By default, the number of processors determines the preset value of the iceberg.worker.num-threads system property. Try increasing parallelism by setting the iceberg.worker.num-threads system property to a higher value to speed up query compilation.
- Reduce read amplification

  Monitor the growth of positional delta files, and perform timely compactions.
- Speed up drop table performance, preventing deletion of data files by using the following table properties:

```
Set external.table.purge=false and gc.enabled=false
```

- Tune the following table properties to improve concurrency on writes and reduce commit failures: commit.retry .num-retries (default is 4), commit.retry.min-wait-ms (default is 100)
- Maintain a relatively small number of data files under the iceberg table/partition directory for efficient reads. To alleviate poor performance caused by too many small files, run the following queries:

```
TRUNCATE TABLE target;
              INSERT OVERWRITE TABLE target select * from target FOR SYST
EM_VERSION AS OF <preTruncateSnapshotId>;
```

- To minimize the number of delete files and file handles and improve performance, ensure that the Spark write.distribution.mode table property value is “hash” (the default setting for Spark Iceberg 1.2.0 onwards).

Making row-level changes on V2 tables only

Learn the types of workloads best suited for Iceberg. Under certain conditions, using V2 tables versus V1 tables might improve query response.

Iceberg atomic DELETE and UPDATE operations resemble traditional RDBMS systems, but are not suitable for OLTP workloads. Iceberg is not designed to handle high frequency transactions. To handle very large datasets and frequent updates, use Apache Kudu.

Use Iceberg for managing large, infrequently changing datasets. You can update and delete Iceberg V2 tables at the row-level and not incur the overhead of rewriting the data files of V1 tables. Iceberg stores information about the deleted records in position delete files. These files store the file paths and positions of the deleted records, eliminating the need to rewrite the files. Iceberg performs a DELETE plus an INSERT operation in a single transaction. This technique speeds up queries. Query engines scan the data files and delete files associated with a snapshot and merge them, removing the deleted rows. For example, to remove all data belonging to a single customer:

DELETE FROM ice_tbl WHERE user_id = 1234;

To update a column value in a specific record:

UPDATE ice_tbl SET col_v = col_v + 1 WHERE id = 4321;

You can convert an Iceberg v1 table to v2 by setting a table property as follows:'format-version' = '2'.

Performance tuning

Impala uses its own C++ implementation to deal with Iceberg tables. This implementation provides significant performance advantages over other engines.

To tune performance, try the following actions:

- Increase parallelism to handle large manifest list files in Spark.

  By default, the number of processors determines the preset value of the iceberg.worker.num-threads system property. Try increasing parallelism by setting the iceberg.worker.num-threads system property to a higher value to speed up query compilation.

- Speed up drop table performance, preventing deletion of data files by using the following table properties:

```
Set external.table.purge=false and gc.enabled=false
```

- Tune the following table properties to improve concurrency on writes and reduce commit failures: commit.retry .num-retries (default is 4), commit.retry.min-wait-ms (default is 100)
- Read Iceberg V2 tables from Hive using vectorization when heavy table scanning occurs as in SELECT COUNT(*) FROM TBL_ICEBERG_PART.
  - set hive.llap.io.memory.mode=cache;
  - set hive.llap.io.enabled=true;
  - set hive.vectorized.execution.enabled=true;
- Use Iceberg from Impala for querying Iceberg tables when latency is a concern.

  The massively parallel SQL query engine, backend executors written in C++, and frontend (analyzer, planner) written in Java delivers query results fast.
- Cache manifest files as described in the next topic.

Caching manifest files

Apache Iceberg provides a mechanism to cache the contents of Iceberg manifest files in memory. The manifest caching feature helps to reduce repeated reads of small Iceberg manifest files from remote storage by Impala Coordinators and Catalogd.

Impala caches table metadata in CatalogD and the local Coordinator's catalog, making table metadata analysis fast if the targeted table metadata and files were previously accessed. Impala might analyze the same table multiple times across concurrent query planning and also within single query planning, so caching is very important.

Having a frontend written in Java, Impala can directly analyze many aspects of the Iceberg table metadata through the Java Library provided by Apache Iceberg. Metadata analysis such as listing the table data file, selecting the table snapshot, partition filtering, and predicate filtering is delegated through Iceberg Java API.

To use the Iceberg Java API while still maintaining fast query planning, Iceberg implements caching strategies in the Iceberg Java Library similar to those used by Apache Impala. The Iceberg manifest caching feature constitutes these caching strategies. For more information about manifest caching, see the Iceberg Manifest Caching Blog.

You can configure manifest caching in Cloudera Manager.

Configuring manifest caching in Cloudera Manager

You can enable or disable manifest caching for Impala Coordinators and Catalogd by setting properties in Cloudera Manager.

About this task

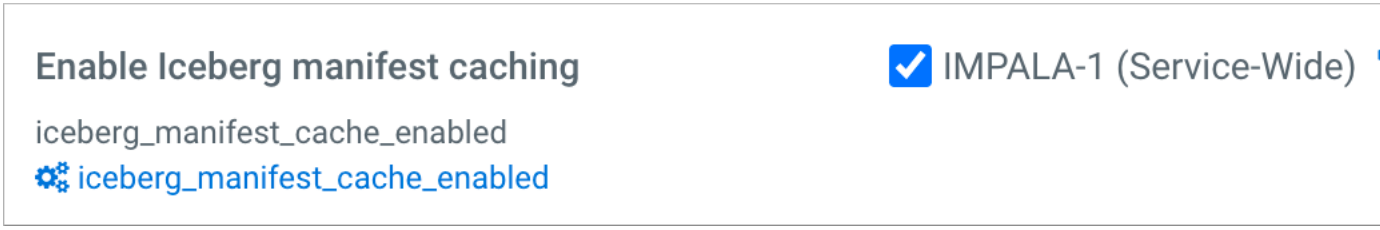
By default, only 8 catalogs can have their manifest cache active in memory.

To connect to more than 8 HDFS clusters, in Cloudera Manager, configure iceberg.io.manifest.cache.fileio-max in catalogd_java_opts and the coordinator impalad_embedded_java_opts.

In the following task, you enable Iceberg manifest caching for the Impala Coordinator and Catalog Server.

Procedure

1. Navigate to Cloudera Manager.
2. Search for Iceberg.



By default, manifest caching is enabled, but if you have turned it off, check Impala-1 (Service-Wide) to re-enable.

Unsupported features and limitations

Cloudera does not support all features in Apache Iceberg. The list of unsupported features for Cloudera differs from release to release. Also, Apache Iceberg in Cloudera has some limitations you need to understand.

Unsupported features

The following table presents feature limitations or unsupported features:

means not yet tested

N/A means will never be tested, not a GA candidate

Iceberg Feature	Hive	Impala	Spark
Branching/Tagging	#	#	#
Read equality deletes for Flink upserts	#	#	#
Read equality deletes for NiFi	#	#	#
Write equality deletes	N/A	N/A	N/A
Read outside files	N/A	N/A	N/A
Bucketing	#	#	#

The table above shows that the following features are not supported in this release of Cloudera:

- Tagging and branching

  A technical preview is supported from Hive (not Impala) in Cloudera Data Warehouse on cloud.
- Reading files outside the table directory

  Reading outside files is not supported due to the security risk. An unauthorized party who knows the underlying schema and file location outside the table location can rewrite the manifest files within one table location to point to the data files in another table location to read your data.
- CLUSTERED BY (id, partition_id) INTO 64 BUCKETS is not supported for Iceberg tables. Use partition bucketing instead.
- Using Iceberg tables as Spark Structured Streaming sources or sinks
- PyIceberg
- Migration of Delta Lake tables to Iceberg

Limitations

The following features have limitations or are not supported for Apache Iceberg:

- Impala only supports reading data files in the AVRO format but does not yet support reading delete files in AVRO format.

This means Impala can always read Iceberg V1 tables containing AVRO files because the V1 version does not have delete files. However, Iceberg V2 tables configured with the merge-on-read delete strategy might contain AVRO delete files.

If Impala encounters AVRO delete files while reading an Iceberg V2 table, it fails with an error.

Workaround: Use one of the following methods to access or modify the table:

- Use Hive or Spark to read Iceberg V2 tables that contain AVRO delete files.
- Change the delete strategy to copy-on-write or compact the table to eliminate the delete files.
- Rewrite the Iceberg table files to the Parquet file format so that Impala can read the delete files as well.

- If you upgrade an Iceberg table from format version 2 (V2) to format version 3 (V3), Impala blocks UPDATE and DELETE operations when the table still contains legacy V2 position delete files.

  Impala fails with the following AnalysisException:

  ```
  AnalysisException: UPDATE/DELETE is not allowed on Iceberg format versio
  n 3 table '[table_name]' as it has existing version 2 position delete fi
  le(s). Run 'OPTIMIZE TABLE [table_name]' to rewrite the files and remove
   deletion files before attempting UPDATE/DELETE.
  ```

  Hive and Spark might allow these operations on the same table. This restriction applies to Impala only.

  Workaround: Before you run UPDATE or DELETE on a migrated V3 table from Impala, run the following command to rewrite the data and metadata:

  ```
  OPTIMIZE TABLE database_name.table_name;
  ```

- Multiple insert overwrite queries that read data from a source table.
- When the underlying table is changed, you need to rebuild the materialized view manually, or use the Hive query scheduling to rebuild the materialized view.
- You must be aware of the following considerations when using equality deletes:
  - Equality updates and deletes are not supported.
  - If you are using Apache Flink or Apache NiFi to write equality deletes, then ensure that you provide a PRIMARY KEY for the table. This is required for engines to know which columns to write into the equality delete files.
  - If the table is partitioned then the partition columns have to be part of the PRIMARY KEY
  - For Apache Flink, the table should be in 'upsert-mode' to write equality deletes
  - Partition evolution is not allowed for Iceberg tables that have PRIMARY KEYs
  - An equality delete file in the table is the likely cause of a problem with updates or deletes in the following situations:
    - In Change Data Capture (CDC) applications
    - In upserts from Apache Flink
    - From a third-party engine
- You must be aware of the following:
  - An Iceberg table that points to another Iceberg table in the HiveCatalog is not supported.

    For example:

    ```
    CREATE EXTERNAL TABLE ice_t
    STORED BY ICEBERG
    TBLPROPERTIES ('iceberg.table_identifier'='db.tb');
    ```

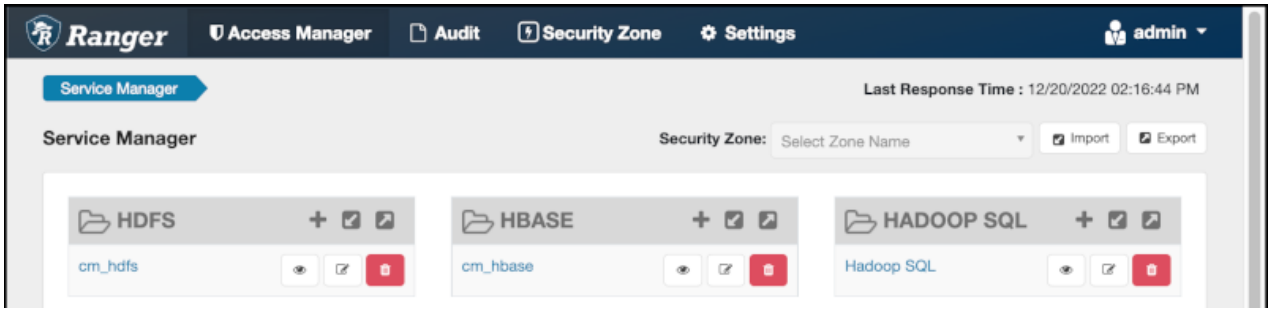
  - See also Iceberg data types.

Accessing Iceberg tables

Cloudera uses Apache Ranger to provide centralized security administration and management. The Ranger Admin UI is the central interface for security administration. You can use Ranger to create two policies that allow users to query Iceberg tables.

How you open the Ranger Admin UI differs from one Cloudera service to another. In Cloudera Management Console, you can select your environment, and then click Environment Details Quick Links Ranger .

You log into the Ranger Admin UI, and the Ranger Service Manager appears.



Policies for accessing tables on S3

The default policies that appear differ from service to service. You need to set up two Hadoop SQL policies to query Iceberg tables:

- One to authorize users to access the Iceberg files

  Follow steps in "Editing a policy to access Iceberg files" below.
- One to authorize users to query Iceberg tables

  Follow steps in "Creating a policy to query an Iceberg table" below.

Prerequisites

- Obtain the RangerAdmin role.
- Get the user name and password your Administrator set up for logging into the Ranger Admin.

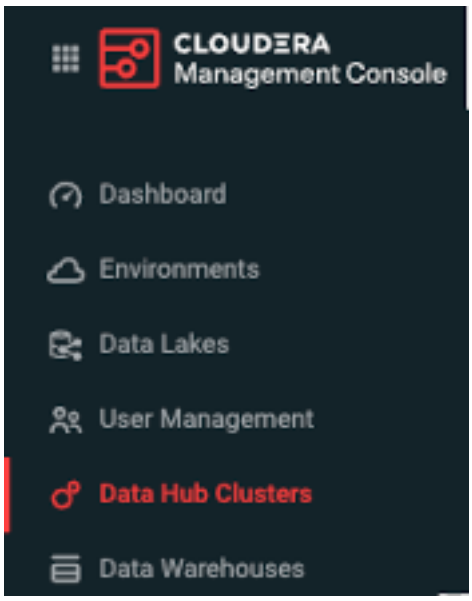
  The default credentials for logging into the Ranger Admin Web UI are admin/admin123.

Opening Ranger in Cloudera Data Hub

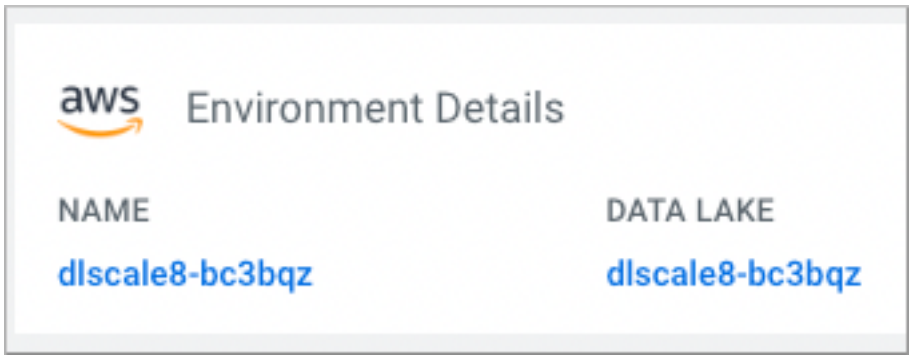
You need to navigate to Ranger Admin UI to create a policy for users to access Iceberg tables. How you navigate to the Ranger Admin UI differs from one Cloudera service to another, and typically there is more than one path. You learn one way to navigate to the Ranger Admin UI from Cloudera Data Hub.

Procedure

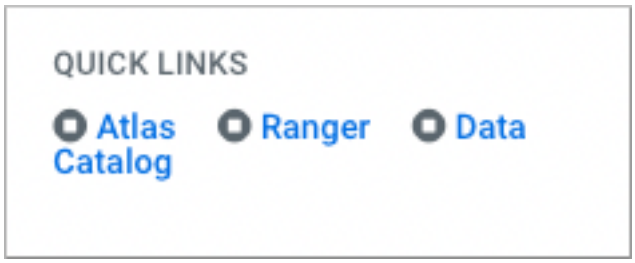
1. Log into Cloudera, and in Cloudera Management Console, click Data Hub Clusters.



2. In **Data Hubs**, select the name of your Data Hub from the list.
3. In Environment Details, click the link to your Data Lake.
   For example, click dlscale8-bc8bqz.



In Data Lake Details, in Quick Links, Ranger appears:



Editing a storage handler policy to access Iceberg files on the file system

You learn how to edit the existing default Hadoop SQL Storage Handler policy to access files. This policy is one of the two Ranger policies required to use Iceberg.

About this task

The Hadoop SQL Storage Handler policy allows references to Iceberg table storage location, which is required for creating or altering a table. You use a storage handler when you create a file stored as Iceberg on the file system or object store.

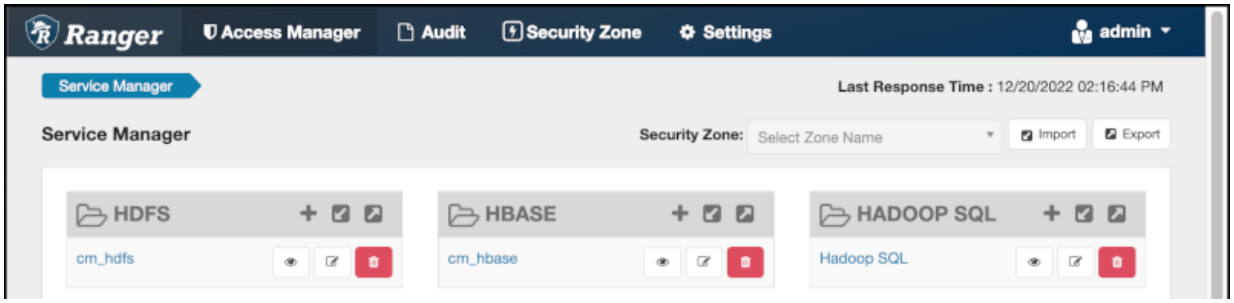
In this task, you specify Iceberg as the storage-type and allow the broadest access by setting the URL to *.

The Hadoop SQL Storage Handler policy supports only the RW Storage permission. A user having the required RW Storage permission on a resource, such as Iceberg, that you specify in the storage-type properties, is allowed only to reference the table location (for create/alter operations) in Iceberg. The RW Storage permission does not provide access to any table data. You need to create the Hadoop SQL policy described in the next topic in addition to this Hadoop SQL Storage Handler policy to access data in tables.

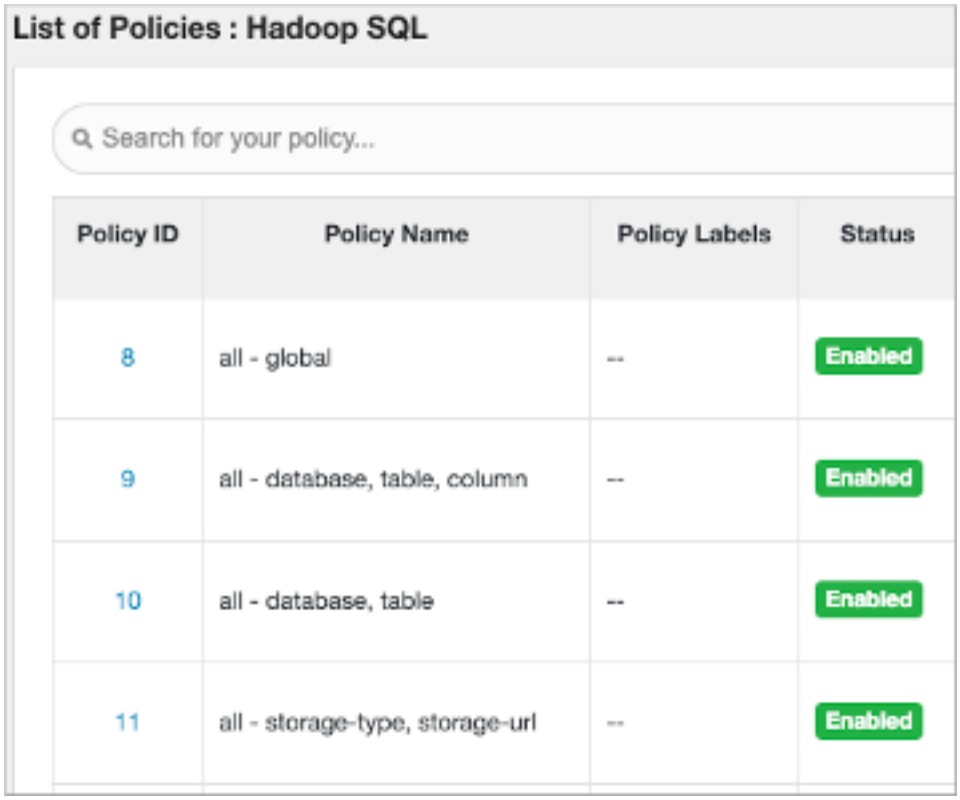
For more information about these policy settings, see Ranger Storage Handler documentation.

Procedure

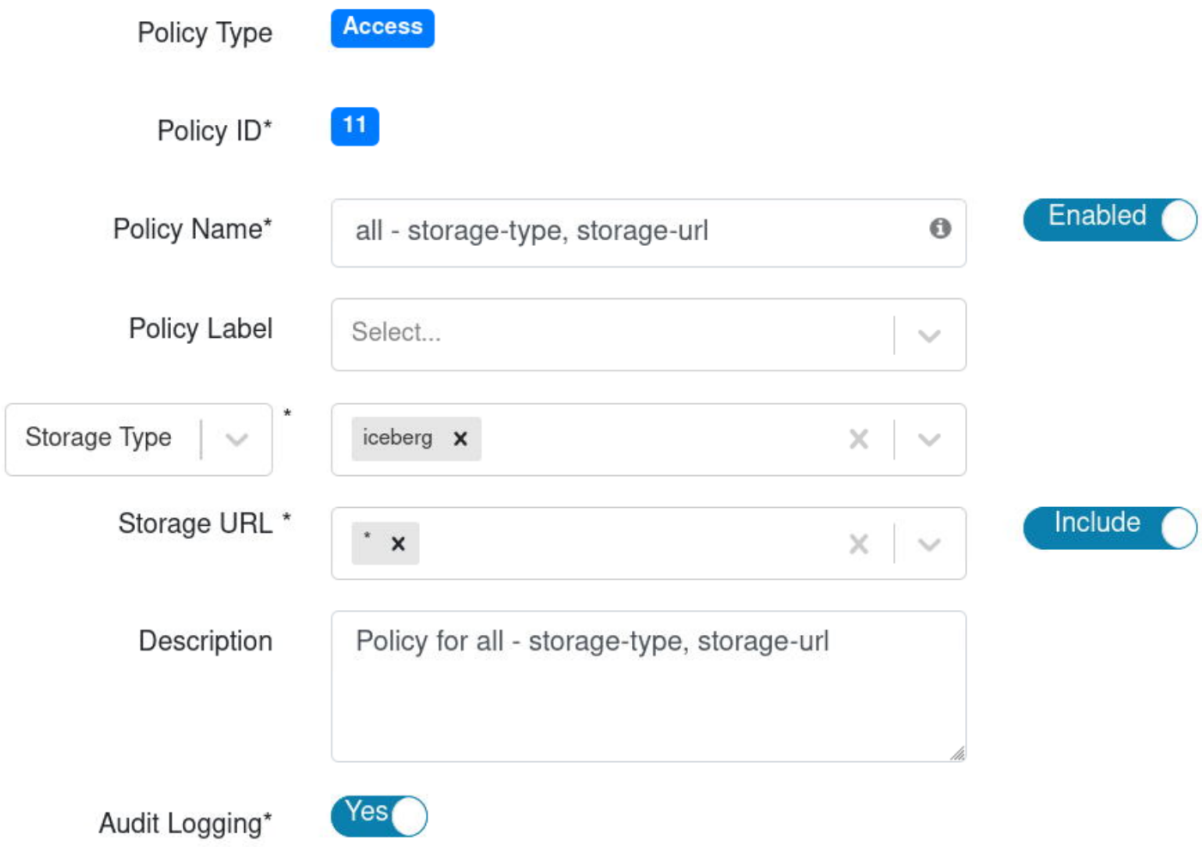
1. Log into Ranger Admin Web UI.
   The Ranger Service Manager appears:



2. In Policy Name, enable the all - storage-type, storage-url policy.



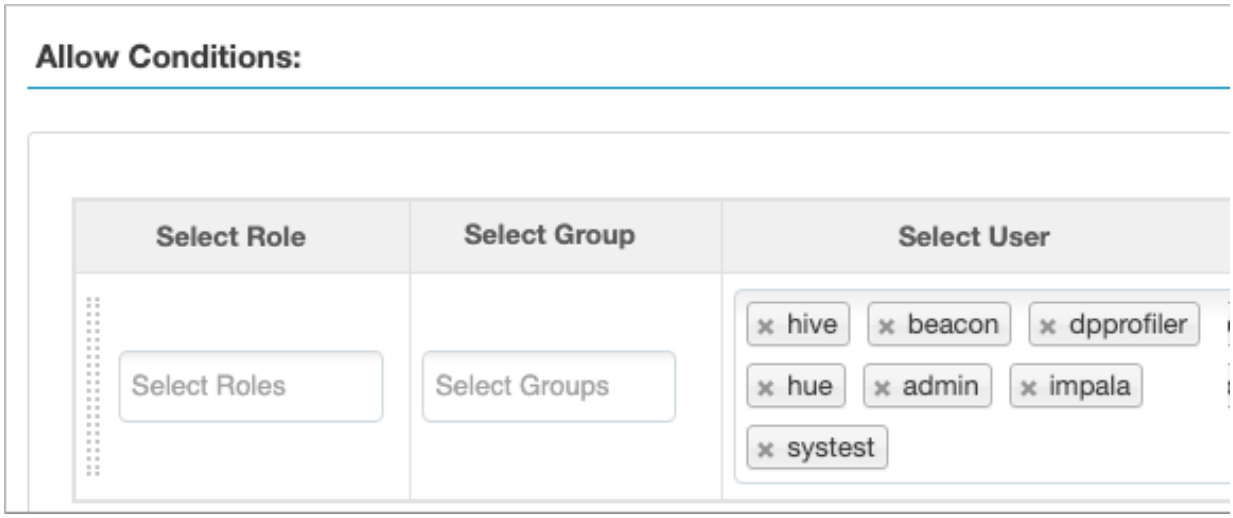
3. In **Service Manager**, in Hadoop SQL, select Edit and edit the all storage-type, storage-url policy.
4. Below Policy Label, select storage-type, and enter iceberg..
5. In Storage URL, enter the value *, enable Include.



For more information about these policy settings, see Ranger storage handler documentation.

6. In Allow Conditions, specify roles, users, or groups to whom you want to grant RW storage permissions.

   You can specify PUBLIC to grant access to Iceberg tables permissions to all users. Alternatively, you can grant access to one user. For example, add the systest user to the list of users who can access Iceberg:



   For more information about granting permissions, see Configure a resource-based policy: Hadoop-SQL.

7. Add the RW Storage permission to the policy.
8. Save your changes.

Creating a SQL policy to query an Iceberg table

You learn how to set up the second required policy for using Iceberg. This policy manages SQL query access to Iceberg tables.

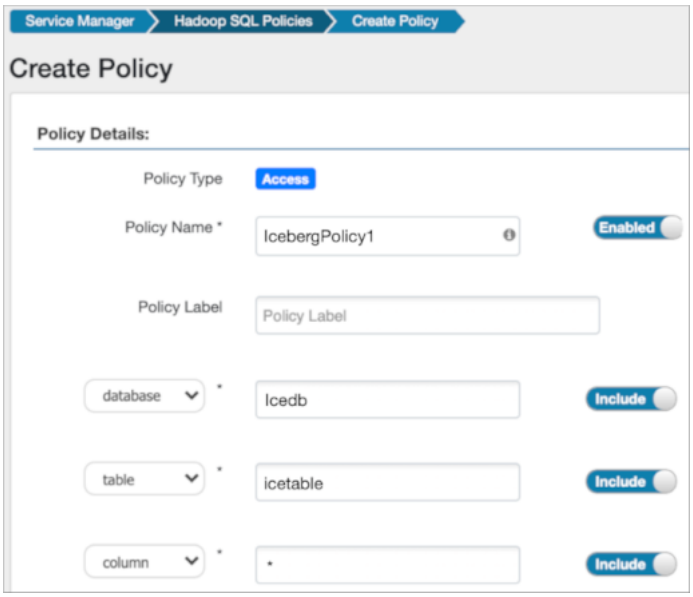
About this task

You create a Hadoop SQL policy to allow roles, groups, or users to query an Iceberg table in a database. In this task, you see an example of just one of many ways to configure the policy conditions. You grant (allow) the selected roles, groups, or users the following add or edit permissions on the table: Select, Update, Create, Drop, Alter, and All. You can also deny permissions.

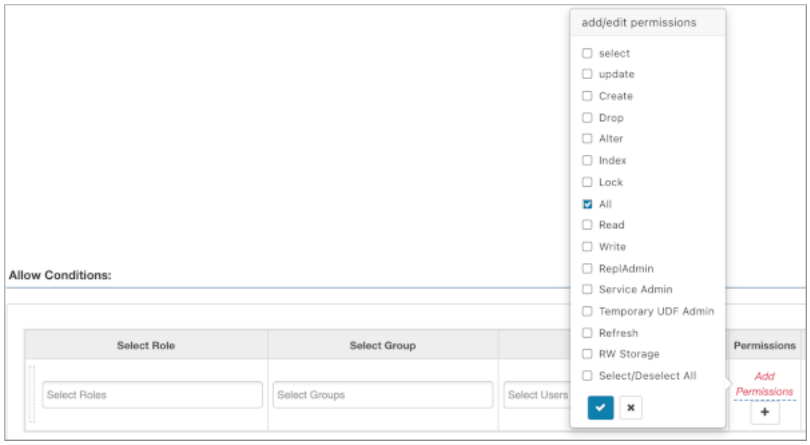
For more information about creating this policy, see Ranger documentation.

Procedure

1. Log into Ranger Admin Web UI.
   The Ranger Service Manager appears.
2. Click Add New Policy.
3. Fill in required fields.
   For example, enter the following required settings:
   - In Policy Name, enter the name of the policy, for example IcebergPolicy1.
   - In database, enter the name of the database controlled by this policy, for example icedb.
   - In table, enter the name of the table controlled by this policy, for example icetable.
   - In columns, enter the name of the column controlled by this policy, for example enter the wildcard asterisk (*) to allow access to all columns of icetable.
   - Accept defaults for other settings.



4. Scroll down to Allow Conditions, and select the roles, groups, or users you want to access the table.
   You can use Deny All Other Accesses to deny access to all other roles, groups, or users other than those specified in the allow conditions for the policy.
5. Select permissions to grant.
   For example, select Create, Select, and Alter. Alternatively, to provide the broadest permissions, select All.



Ignore RW Storage and other permissions not named after SQL queries. These are for future implementations.

6. Click Add.

Creating an Iceberg table

A step-by-step procedure describes how to create an Apache Iceberg table from a client connection to Hive or Impala, or from Data Explorer in Cloudera Data Hub. You see how to access and use the query editor Data Explorer to create an Iceberg table.

About this task

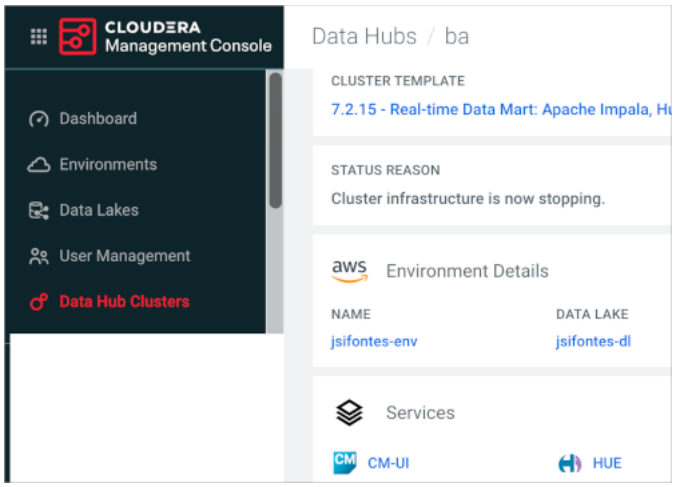
In this task, from a Data Hub cluster, you open Data Explorer, and use Hive or Impala to create a table.

Before you begin

- You must meet the prerequisites to query Iceberg tables, including obtaining Ranger access permissions.

Procedure

1. Log into Cloudera, and click Data Hub Clusters.



2. Click Hue.
3. Select a database.
4. Enter a query to create a simple Iceberg table in the default Parquet format.

Hive example:

```
CREATE EXTERNAL TABLE ice_t1 (i int, s string, ts timestamp, d date)
STORED BY ICEBERG;
```

Impala example:

```
CREATE TABLE ice_t2 (i int, s string, ts timestamp, d date)
STORED BY ICEBERG;
```

In Cloudera, CREATE EXTERNAL TABLE, and just CREATE TABLE, are valid from Hive. You use the EXTERNAL keyword from Hive to create the Iceberg table to purge the data when you drop the table. In Cloudera, from Impala, you must use CREATE TABLE to initialize the Iceberg table.

5. Click  to run the query.

Creating an Iceberg partitioned table

The ease of use of the Iceberg partitioning is clear from an example of how to partition a table using the backward compatible, identity-partition syntax. Alternatively, you can partition an Iceberg table by column values from Hive or Impala.

About this task

You can specify partitioning that is backward compatible with Iceberg V1 using the PARTITION BY clause. This type of table is called an identity-partitioned table. For more information about partitioning, see the Apache Iceberg documentation.

Procedure

1. Select, or use, a database.
2. Create an identity-partitioned table and run the query.
   Hive:

```
CREATE EXTERNAL TABLE ice_ext1 (i int, s string, ts timestamp, d date) P
ARTITIONED BY (state string)
STORED BY ICEBERG
STORED AS ORC;
```

   Impala:

```
CREATE TABLE ice_ext2 (i int, s string, ts timestamp, d date) PARTITIONED
 BY (state string)
STORED BY ICEBERG;
```

3. Create a table and specify an identity transform, such as bucket, truncate, or date, using the Iceberg V2 PARTITION BY SPEC clause.
   Hive:

```
CREATE TABLE ice_t_transforms (i int, s string, ts timestamp, d date)
PARTITIONED BY SPEC (TRUNCATE(10, i), BUCKET(11, s), YEAR(ts))
STORED BY ICEBERG;
```

   Impala:

```
CREATE TABLE ice_t_transforms (i int, s string, ts timestamp, d date)
```

```
PARTITIONED BY SPEC (TRUNCATE(10, i), BUCKET(11, s), YEAR(ts))
STORED AS ICEBERG;
```

Related Information

Partition transform feature

Expiring snapshots

You can expire snapshots of an Iceberg table using an ALTER TABLE query. Cloudera recommends periodically expiring snapshots to delete data files that are no longer needed, and reduce the size of table metadata.

About this task

Each write to an Iceberg table creates a new snapshot, or version, of a table. You can use snapshots for time-travel queries, or to roll back a table to a valid snapshot. Snapshots accumulate until they are expired by the expire_snapshots operation.

Procedure

1. Enter a query to expire snapshots older than the following timestamp: '2021-12-09 05:39:18.689000000'

```
ALTER TABLE test_table EXECUTE EXPIRE_SNAPSHOTS('2021-12-09 05:39:18.689
000000');
```

2. Enter a query to expire snapshots having between December 10, 2022 and November 8, 2023.

```
ALTER TABLE test_table EXECUTE EXPIRE_SNAPSHOTS BETWEEN ('2022-12-10 00:
00:00.000000000') AND ('2023-11-08 00:00:00.000000000');
```

Related Information

Expire snapshots feature

Inserting data into a table

You can append data to an Iceberg table by inserting values or by selecting the data from another table. You can update data, replacing the old data.

You use the INSERT command in one of the following ways to populate an Iceberg table from Hive:

- INSERT INTO t VALUES (1, ‘asf’, true);
- INSERT INTO t SELECT * FROM s;
- INSERT OVERWRITE t SELECT * FROM s;

Examples

```
INSERT INTO t VALUES (1, ‘asf’, true);
INSERT INTO t SELECT * FROM s;
INSERT OVERWRITE t SELECT * FROM s;
```

Migrating a Hive table to Iceberg

You see how to use a simple ALTER TABLE statement from Hive or Impala to migrate an external Hive table to an Iceberg table. You see how to configure table input and output by setting table properties.

About this task

Note: To prevent loss of new and old table data during migration of a table to Iceberg, do not drop, move, or change the old table during migration.

In this task, from a Cloudera Data Hub cluster, you open Data Explorer, and use Hive or Impala to create a table. In Impala, you can configure the NUM_THREADS_FOR_TABLE_MIGRATION query option to tweak the performance of the table migration. It sets the maximum number of threads to be used for the migration process but could also be limited by the number of CPUs. If set to zero then the number of available CPUs on the coordinator node is used as the maximum number of threads. Parallelism occurs on the basis of data files within a partition, which means one partition is processed at a time with multiple threads processing the files inside the partition. In case there is only one file in each partition, sequential execution occurs.

Before you begin

You must meet the prerequisites to query Iceberg tables, including obtaining Ranger access permissions.

Procedure

1. Log into Cloudera, and click Data Hub Clusters.
2. Click Hue.
3. Select a database.
4. Enter a query to use a database.
   For example:

   ```
   USE mydb;
   ```

5. Enter a query to migrate an existing external Hive table to an Iceberg v2 table.
   Hive example:

   ```
   ALTER TABLE tbl
   SET TBLPROPERTIES ('storage_handler'='org.apache.iceberg.mr.hive.HiveIc
   ebergStorageHandler',
    'format-version' = '2');
   ```

   Impala example, which requires two queries:

   ```
   ALTER TABLE table_name CONVERT TO ICEBERG;
   ALTER TABLE table_name SET TBLPROPERTIES ('format-version'='2');
   ```

   The first ALTER command converts the Hive table to an Iceberg V1 table.
6. Click to run the queries.
   An Iceberg V2 table is created, replacing the Hive table.

Selecting an Iceberg table

You see an example of how to read an Apache Iceberg table, and understand the advantages of Iceberg.

About this task

Working with timestamps in Iceberg, you do not need to know whether the table is actually partitioned by month, day or hour, based on the timestamp value. You can simply supply a predicate for the timestamp value and Iceberg converts the timestamp to month/day/hour transparently. Hive/Impala must maintain actual partition values in a separate column (for example, ts_month or ts_day). Forgetting to reference the derived partition column in your query can lead to inadvertent full table scans.

By default iceberg.table_identifier is not set in Cloudera, so you can use the familiar <db_name.<table_name> in queries.

Procedure

1. Use a database.
   For example:

   ```
   USE mydatabase;
   ```

2. Query an Iceberg table partitioned by city.
   For example:

   ```
   SELECT * FROM ice_t2 WHERE city="Bangalore";
   ```

Running time travel queries

You query historical snapshots of data using the FOR SYSTEM_TIME AS OF '<timestamp>' FOR SYSTEM_VERSION AS OF <snapshot_id> clauses in a select statement. You see how to use AS OF to specify a snapshot of your Iceberg data at a certain time.

About this task

You can inspect the history of an Iceberg table to see the snapshots. You can query the metadata of the Iceberg table using a SELECT ... AS OF statement to run time travel queries. You use history information from a query of the database to identify and validate snapshots, and then query a specific snapshot AS OF a certain Timestamp value.

Before you begin

- You must be aware of the table history.

  However, this can include commits that have been rolled back.
- You must have access to valid snapshots.

Procedure

1. View the table history.

   ```
   SELECT * FROM db.table.history;
   ```

2. Check the valid snapshots of the table.

   ```
   SELECT * FROM db.table.snapshots;
   ```

3. Query a specific snapshot by providing the timestamp and snapshot_id.

   ```
   SELECT * FROM T
   FOR SYSTEM_TIME AS OF <TIMESTAMP>;
   SELECT * FROM t
   FOR SYSTEM_VERSION AS OF <SNAPSHOT_ID>;
   ```

Updating an Iceberg partition

You see how to update Iceberg table partitioning in an existing table and then how to change the partitioning to be more granular.

About this task

Partition information is stored logically, and only in table metadata. When you update a partition spec, the old data written with an earlier spec remains unchanged. New data is written using the new spec in a new layout. Metadata for each of the partition versions is separate.

Procedure

1. Create a table partitioned by year.
   Hive

   ```
   CREATE EXTERNAL TABLE ice_t (i int, j int, ts timestamp)
   PARTITIONED BY SPEC (truncate(5, j), year(ts))
   STORED BY ICEBERG;
   ```

   Impala:

   ```
   CREATE TABLE ice_t (i int, j int, ts timestmap)
   PARTITIONED BY SPEC (truncate(5, j), year(ts))
   STORED BY ICEBERG;
   ```

2. Split the data into manageable files using buckets.

   ```
   ALTER TABLE ice_t SET PARTITION SPEC (bucket(13, i));
   ```

3. Partition the table by month.

   ```
   ALTER TABLE ice_t SET PARTITION SPEC  (truncate(5, j), month(ts));
   ```

Test driving Iceberg from Impala

You complete a task that creates Iceberg tables from Impala with mock data that you can test drive using your own queries. You learn how to work with partitioned tables.

Before you begin

- You must obtain Ranger access permissions.

Procedure

1. In Impala, use a database.
2. Create an Impala table to hold mock data for this task.

   ```
   create external table mock_rows stored as parquet as
   select x from (
   with v as (values (1 as x), (1), (1), (1), (1))
   select v.x from v, v v2, v v3, v v4, v v5, v v6
   ) a;
   ```

3. Create another Impala table based on mock_rows.

```
create external table customer_demo stored as parquet as
select
FROM_TIMESTAMP(DAYS_SUB(now() , cast ( TRUNC(RAND(7)*365*1) as bigint)), '
yyyy-MM') as year_month,
DAYS_SUB(now() , cast ( TRUNC(RAND(7)*365*1) as bigint)) as ts,
CONCAT(
  cast ( TRUNC(RAND(1) * 250 + 2) as string), '.' ,
  cast ( TRUNC(RAND(2) * 250 + 2) as string), '.',
  cast ( TRUNC(RAND(3) * 250 + 2) as string), '.',
  cast ( TRUNC(RAND(4) * 250 + 2) as string)
) as ip,
CONCAT("USER_", cast ( TRUNC(RAND(4) * 1000) as string),'@somedomain.com')
 as email,
CONCAT("USER_", cast ( TRUNC(RAND(5) * 1000) as string)) as username,
CONCAT("USER_", cast ( TRUNC(RAND(6) * 100) as string)) as country,
cast( RAND(8)*10000 as double) as metric_1,
cast( RAND(9)*10000 as double) as metric_2,
cast( RAND(10)*10000 as double) as metric_3,
cast( RAND(11)*10000 as double) as metric_4,
cast( RAND(12)*10000 as double) as metric_5
from mock_rows
;
```

4. Create another Impala table based on mock_rows.

```
create external table customer_demo2 stored as parquet as
select
FROM_TIMESTAMP(DAYS_SUB(now() , cast ( TRUNC(RAND(7)*365*1) as bigint)),
'yyyy-MM') as year_month,
DAYS_SUB(now() , cast ( TRUNC(RAND(7)*365*1) as bigint)) as ts,
CONCAT(
  cast ( TRUNC(RAND(1) * 250 + 2) as string), '.' ,
  cast ( TRUNC(RAND(2) * 250 + 2) as string), '.',
  cast ( TRUNC(RAND(3) * 250 + 2) as string), '.',
  cast ( TRUNC(RAND(4) * 250 + 2) as string)
) as ip,
CONCAT("USER_", cast ( TRUNC(RAND(4) * 1000) as string),'@somedomain.com')
 as email,
CONCAT("USER_", cast ( TRUNC(RAND(5) * 1000) as string)) as username,
CONCAT("USER_", cast ( TRUNC(RAND(6) * 100) as string)) as country,
cast( RAND(8)*10000 as double) as metric_1,
cast( RAND(9)*10000 as double) as metric_2,
cast( RAND(10)*10000 as double) as metric_3,
cast( RAND(11)*10000 as double) as metric_4,
cast( RAND(12)*10000 as double) as metric_5
from mock_rows
;
```

5. Create an Iceberg table from the customer_demo table.

```
CREATE TABLE customer_demo_iceberg STORED BY ICEBERG AS SELECT * FROM cu
stomer_demo;
```

6. Insert into the customer_demo_iceberg table the results of selecting all data from the customer_demo2 table.

```
INSERT INTO customer_demo_iceberg select * from customer_demo2;
INSERT INTO customer_demo_iceberg select * from customer_demo2;
INSERT INTO customer_demo_iceberg select * from customer_demo2;
```

7. Create an Iceberg table partitioned by the year_month column and based on the customer_demo_iceberg table.

```
CREATE TABLE customer_demo_iceberg_part PARTITIONED BY(year_month) STORED
 BY ICEBERG
AS SELECT ts, ip , email, username , country, metric_1 , metric_2 , metric
_3 , metric_4 , metric_5, year_month
FROM customer_demo_iceberg;
```

8. Split the partitioned data into manageable files.

```
ALTER TABLE customer_demo_iceberg_part SET PARTITION SPEC (year_month,BU
CKET(15, country));
```

9. Insert the results of reading the customer_demo_iceberg table into the partitioned table.

```
INSERT INTO customer_demo_iceberg_part (year_month, ts, ip, email, usern
ame, country, metric_1, metric_2, metric_3, metric_4, metric_5)
SELECT year_month, ts, ip, email, username, country, metric_1, metric_2,
 metric_3, metric_4, metric_5
FROM customer_demo_iceberg;
```

10. Run time travel queries on the Iceberg tables, using the history output to get the snapshot id, and substitute the id in the second SELECT query.

```
SELECT * FROM customer_demo_iceberg FOR SYSTEM_TIME AS OF '2021-12-09 05
:39:18.689000000' LIMIT 100;
DESCRIBE HISTORY customer_demo_iceberg;
SELECT * FROM customer_demo_iceberg FOR SYSTEM_VERSION AS OF <snapshot
id> LIMIT 100;
```

Hive demo data

To test drive Iceberg from Hive, you need to set up Hive demo data.

Iceberg Database creation and setup

The Airlines demo data for Iceberg is stored in the airline_online_iceberg database. The following queries created and set up this database.

```
create database if not exists airline_ontime_iceberg;
use airline_ontime_iceberg;
set hive.vectorized.execution.enabled=false;
set hive.stats.column.autogather=false;
```

Hive external table creation

The following Hive external tables were created in the airline_online_iceberg database:

- airports
- airlines
- planes
- flights

```
create external table if not exists airports (
    iata string,
    airport string,
    city string,
    state double,
```

```
    country string,
    lat double,
    lon double
)
stored as orc;

create external table if not exists airlines (
    code string,
    description string
)
stored as orc;
create external table if not exists planes (
    tailnum string,
    owner_type string,
    manufacturer string,
    issue_date string,
    model string,
    status string,
    aircraft_type string,
    engine_type string,
    year int
)
stored as orc;

create external table if not exists flights (
    month int,
    dayofmonth int,
    dayofweek int,
    deptime  int,
    crsdeptime int,
    arrtime int,
    crsarrtime int,
    uniquecarrier string,
    flightnum int,
    tailnum string,
    actualelapsedtime int,
    crselapsedtime int,
    airtime int,
    arrdelay int,
    depdelay int,
    origin string,
    dest string,
    distance int,
    taxiin int,
    taxiout int,
    cancelled int,
    cancellationcode string,
    diverted string,
    carrierdelay int,
    weatherdelay int,
    nasdelay int,
    securitydelay int,
    lateaircraftdelay int
)
partitioned by (year int)
stored as orc;
```

Load data into the newly created tables

```
load data inpath '${datapath}/airline_ontime_iceberg.db/airports' into table
 airports;
```

```
load data inpath '${datapath}/airline_ontime_iceberg.db/airlines' into table
 airlines;

load data inpath '${datapath}/airline_ontime_iceberg.db/planes' into table p
lanes;

load data inpath '${datapath}/airline_ontime_iceberg.db/flights/year=1995' i
nto table flights partition (year=1995);
load data inpath '${datapath}/airline_ontime_iceberg.db/flights/year=1996'
into table flights partition (year=1996);
load data inpath '${datapath}/airline_ontime_iceberg.db/flights/year=1997'
 into table flights partition (year=1997);
load data inpath '${datapath}/airline_ontime_iceberg.db/flights/year=1998'
 into table flights partition (year=1998);
load data inpath '${datapath}/airline_ontime_iceberg.db/flights/year=1999' i
nto table flights partition (year=1999);
load data inpath '${datapath}/airline_ontime_iceberg.db/flights/year=2000'
into table flights partition (year=2000);
load data inpath '${datapath}/airline_ontime_iceberg.db/flights/year=2001'
 into table flights partition (year=2001);
load data inpath '${datapath}/airline_ontime_iceberg.db/flights/year=2002'
 into table flights partition (year=2002);
load data inpath '${datapath}/airline_ontime_iceberg.db/flights/year=2003' i
nto table flights partition (year=2003);
load data inpath '${datapath}/airline_ontime_iceberg.db/flights/year=2004'
into table flights partition (year=2004);
load data inpath '${datapath}/airline_ontime_iceberg.db/flights/year=2005'
 into table flights partition (year=2005);
load data inpath '${datapath}/airline_ontime_iceberg.db/flights/year=2006'
 into table flights partition (year=2006);
load data inpath '${datapath}/airline_ontime_iceberg.db/flights/year=2007' i
nto table flights partition (year=2007);
load data inpath '${datapath}/airline_ontime_iceberg.db/flights/year=2008'
into table flights partition (year=2008);
```

Convert these existing Hive external tables to Iceberg tables

```
ALTER TABLE planes ADD CONSTRAINT planes_pk PRIMARY KEY (tailnum) DISABLE NO
VALIDATE;
ALTER TABLE flights ADD CONSTRAINT planes_fk FOREIGN KEY (tailnum) REFEREN
CES planes(tailnum) DISABLE NOVALIDATE RELY;

ALTER TABLE airlines ADD CONSTRAINT airlines_pk PRIMARY KEY (code) DISABLE
 NOVALIDATE;
ALTER TABLE flights ADD CONSTRAINT airlines_fk FOREIGN KEY (uniquecarrier)
REFERENCES airlines(code) DISABLE NOVALIDATE RELY;

ALTER TABLE airports ADD CONSTRAINT airports_pk PRIMARY KEY (iata) DISABLE N
OVALIDATE;
ALTER TABLE flights ADD CONSTRAINT airports_orig_fk FOREIGN KEY (origin)
REFERENCES airports(iata) DISABLE NOVALIDATE RELY;
ALTER TABLE flights ADD CONSTRAINT airports_dest_fk FOREIGN KEY (dest) RE
FERENCES airports(iata) DISABLE NOVALIDATE RELY;

ALTER TABLE airports SET TBLPROPERTIES ('storage_handler'='org.apache.iceb
erg.mr.hive.HiveIcebergStorageHandler');
ALTER TABLE airlines SET TBLPROPERTIES ('storage_handler'='org.apache.icebe
rg.mr.hive.HiveIcebergStorageHandler');
ALTER TABLE planes SET TBLPROPERTIES ('storage_handler'='org.apache.iceberg.
mr.hive.HiveIcebergStorageHandler');
ALTER TABLE flights SET TBLPROPERTIES ('storage_handler'='org.apache.iceber
g.mr.hive.HiveIcebergStorageHandler');
```

Test driving Iceberg from Hive

You learn how to access the Hive demo data, which you can use to get hands-on experience running Iceberg queries.

About this task

Query sample airline demo data in Data Explorer.

Before you begin

- You must meet the prerequisites to query Iceberg tables.
- You obtained the Ranger permissions to run Hive queries.

Procedure

1. Connect to Hive running in a Cloudera Data Hub cluster.
2. Run the queries in the previous topic, "Hive demo data" to set up the following databases: airline_ontime_iceberg, airline_ontime_orc, airline_ontime_parquet.
3. Use the airline_ontime_iceberg database.
4. Take a look at the tables in the airline_ontime_iceberg database.

   ```
   USE airline_ontime_iceberg;
   SHOW TABLES;
   ```

   Flights is the fact table. It has 100M rows and three dirmensions, ariline, airports, and planes. This records flights for more than 10 years in the US, and includes the following details:

   - origin
   - destination
   - delay
   - air time
5. Query the demo data from Hive.
   For example, find the flights that departed each year, by IATA code, airport, city, state, and country. Find the average departure delay.

   ```
   SELECT f.month, a.iata, a.airport, a.city, a.state, a.country
   FROM flights f,
   airports a
   WHERE f.origin = a.iata
   GROUP BY
   f.month,
   a.iata,
   a.airport,
   a.city,
   a.state,
   a.country
   HAVING COUNT(*) > 10000
   ORDER BY AVG(f.DepDelay) DESC
   LIMIT 10;
   ```

   Output appears as follows:

   ```
   +----------+---------+-------------------------------------+---------------
   +----------+------------+
   ```

```
| f.month  | a.iata  |            a.airport                |    a.city
 | a.state  | a.country  |
+----------+---------+-------------------------------------+---------------
+----------+------------+
| 12       | ORD     | Chicago O'Hare International        | Chicago
 | NULL     | USA        |
| 6        | EWR     | Newark Intl                         | Newark
 | NULL     | USA        |
| 7        | JFK     | John F Kennedy Intl                 | New York
 | NULL     | USA        |
| 6        | IAD     | Washington Dulles International     | Chantilly
 | NULL     | USA        |
| 7        | EWR     | Newark Intl                         | Newark
 | NULL     | USA        |
| 6        | PHL     | Philadelphia Intl                   | Philadelphia
 | NULL     | USA        |
| 1        | ORD     | Chicago O'Hare International        | Chicago
 | NULL     | USA        |
| 6        | ORD     | Chicago O'Hare International        | Chicago
 | NULL     | USA        |
| 7        | ATL     | William B Hartsfield-Atlanta Intl   | Atlanta
 | NULL     | USA        |
| 12       | MDW     | Chicago Midway                      | Chicago
 | NULL     | USA        |
+----------+---------+-------------------------------------+---------------
+----------+------------+
10 rows selected (103.812 seconds)
```

6. Split the partitioned data into manageable files.

```
ALTER TABLE airports SET PARTITION SPEC (iata,BUCKET(15, country));
```

Iceberg data types

References include Iceberg data types and a table of equivalent SQL data types by Hive/Impala SQL engine types.

Iceberg supported data types

Table 1:

Iceberg data type	SQL data type	Hive	Impala
binary		BINARY	BINARY
boolean	BOOLEAN	BOOLEAN	BOOLEAN
date	DATE	DATE	DATE
decimal(P, S)	DECIMAL(P, S)	DECIMAL (P, S)	DECIMAL (P, S)
double	DOUBLE	DOUBLE	DOUBLE
fixed(L)		BINARY	Not supported
float	FLOAT	FLOAT	FLOAT
int	TINYINT, SMALLINT, INT	INTEGER	INTEGER
list	ARRAY	ARRAY	Read only
long	BIGINT	BIGINT	BIGINT
map	MAP	MAP	Read only

Iceberg data type	SQL data type	Hive	Impala
string	VARCHAR, CHAR	STRING	STRING
struct	STRUCT	STRUCT	Read only
time		STRING	Not supported
timestamp	TIMESTAMP	TIMESTAMP	TIMESTAMP (see limitation below)
timestamptz	TIMESTAMP WITH LOCAL TIME ZONE	Use TIMESTAMP WITH LOCAL TIMEZONE for handling these in queries	Read timestamptz into TIMESTAMP values Writing not supported
uuid	none	STRING Writing to Parquet is not supported	Not supported
variant	VARIANT	VARIANT	Not supported

Data type limitations

An implicit conversion to an Iceberg type occurs only if there is an exact match; otherwise, a cast is needed. For example, to insert a VARCHAR(N) column into an Iceberg table you need a cast to the VARCHAR type as Iceberg does not support the VARCHAR(N) type. To insert a SMALLINT or TINYINT into an Iceberg table, you need a cast to the INT type as Iceberg does not support these types.

Iceberg supports two timestamp types:

- timestamp (without timezone)
- timestamptz (with timezone)

Starting from Spark 3.4 onwards, Spark SQL supports a timestamp with local timezone (TIMESTAMP_LTZ) type and a timestamp without timezone (TIMESTAMP_NTZ) type, with TIMESTAMP defaulting to the TIMESTAMP_LTZ type. However, this can be configured by setting the spark.sql.timestampType (the default value is TIMESTAMP_LTZ).

When creating an Iceberg table using Spark SQL, if spark.sql.timestampType is set to TIMESTAMP_LTZ, TIMESTAMP is mapped to Iceberg's timestampz type. If spark.sql.timestampType is set to TIMESTAMP_NTZ, then TIMESTAMP is mapped to Iceberg's timestamp type.

Impala is unable to write to Iceberg tables with timestamptz columns. For interoperability, when creating Iceberg tables from Spark, you can use the Spark configuration, spark.sql.timestampType=TIMESTAMP_NTZ.

Note that the timestamp and timestamptz types have different semantics.

Unsupported data types

Impala does not support the following Iceberg data types:

- TIMESTAMPTZ (only read support)
- TIMESTAMP in tables in AVRO format
- FIXED
- UUID

Iceberg table properties

The Cloudera environment for querying tables from Hive overrides some Iceberg table properties. You learn which table properties are supported for querying tables from Impala.

Iceberg documentation describes all the properties for configuring tables. This documentation focuses on key properties for working with Iceberg tables in Cloudera.

Iceberg supports concurrent writes by default. You can tune Iceberg v2 table properties for concurrent writes. You set the following properties if you plan to have concurrent writers on Iceberg v2 tables:

- commit.retry.min-wait-ms
- commit.retry.num-retries

Cloudera supports adding the Parquet compression type using table properties. For more information, see Iceberg documentation about Compression Types.

You can use the Alter Table feature to set a property. From Hive, the following Iceberg table property overrides are in effect:

- iceberg.mr.split.size overrides read.split.target-size.
- read.split.open-file-cost is overridden.

You can tune Iceberg v2 table properties for concurrent writes. From Impala, the following subset of Iceberg table properties are supported:

- history.expire.min-snapshots-to-keep

  Valid values: integers. Default = 1
- write.format.default

  Valid value: Parquet
- write.metadata.delete-after-commit.enabled

  Valid values: true or false.
- write.metadata.previous-versions-max

  Valid values: integers. Default = 100.
- write.parquet.compression-codec

  Valid values: GZIP, LZ4, NONE, SNAPPY (default value), ZSTD
- write.parquet.compression-level

  Valid values: 1 - 22. Default = 3
- write.parquet.row-group-size-bytes

  Valid values: 8388608 (or 8 MB) - 2146435072 (or 2047MB). Overiden by PARQUET_FILE_SIZE.
- write.parquet.page-size-bytes

  Valid values: 65536 (or 64KB) - 1073741824 (or 1GB).
- write.parquet.dict-size-bytes

  Valid values: 65536 (or 64KB) - 1073741824 (or 1GB)