Writing Iceberg V2 tables in Impala

Zoltán Borók-Nagy (boroknagyz@cloudera.com)

Overview

This doc describes the design choices of writing Iceberg V2 tables via Apache Impala. Please note that INSERT INTO / INSERT OVERWRITE already works for V2 tables, this document focuses on the row-level modifications, i.e. the DELETE and the UPDATE statement.
We also restrict ourselves to position delete files.

Non-goals

- Writing equality delete files
- Modifying rows via “copy-on-write” mode
- Multi-statement transactions

Background

The major new feature of the Iceberg format version 2 is the row-level modifications via merge-on-read. In Iceberg V1 when the user wanted to delete/update a record they had to rewrite the whole data file that contained the record (aka “copy-on-write” mode). With Iceberg V2 we can collect information about the deleted records in delete files. There are two kinds of delete files, both of them store some identifiers of the deleted records:

- ***Position delete files***: store the file URIs and ordinal positions of the deleted records
- ***Equality delete files***: store a subset of column values of the deleted records (typically unique IDs). Different delete files might use different subset of columns

UPDATE operations can be split into an atomic pair of DELETE + INSERT operations.

DELETE statements

Let’s start with DELETEs because they are obviously simpler than UPDATEs. We need to write DELETE files, so let’s see what they look like. Their file format can be controlled via the table property “write.delete.format.default” based on the Iceberg Configuration page. By default the format is the same as the data file format, i.e. Iceberg tables with Parquet data files have Parquet delete files unless specified otherwise. But since Impala can only write Parquet, it limits our options. Their schema is based on the Iceberg spec:

Field ID	Name	Type	Description
2147483546	file_path	STRING	Full URI of a data file with FS scheme. This must

			match the file_path of the target data file in a manifest entry
2147483545	pos	LONG	Ordinal position of a deleted row in the target data file identified by file_path, starting at 0
2147483544	row	STRUCT<...>	Deleted row values. Omit the column when not storing deleted rows.

The row column can be omitted, and we will omit this as Impala cannot write STRUCTs. Also, the delete files might be consuming too much space if the row field was stored. The downside of not storing the deleted records is that we won't have column statistics about them. This means that predicates pushed down to Iceberg will not filter the delete files, therefore we might end up evaluating more delete files than necessary.

The rows in the delete file must be sorted by `file_path` then `position`.

To write such files we can simply rewrite a DELETE statement to an INSERT INTO statement:

```
DELETE FROM ice_t WHERE <cond>;
```

To

```
INSERT INTO ice_t.DELETE (file_path, pos)
SELECT input__file__name, file__position, <part_cols>
FROM ice_t
WHERE <cond>;
```

`ice_t.DELETE` can be a virtual table created on the fly by the query. It has a schema corresponding to the schema of Iceberg position delete files. <part_cols> (partition columns) are added to the statement so we can write delete files according to the Iceberg table's partitioning. We should shuffle the delete rows across executors based on their partitions so each partition is written out by one executor, hence we end up with as few delete files as possible.

We might also need to make the virtual table `ice_t.DELETE` partitioned by the same partition spec that ice_t uses. In this case we need to tweak the delete writer to ignore the partition columns when it writes out the rows, and only use partitioning for clustering the files.

UPDATE statements

UPDATE statements can be split into two: a DELETE statement and an INSERT statement:

```
UPDATE ice_t SET col_i = col_i + 1 WHERE <cond>;
```

To

```
DELETE FROM ice_t WHERE <cond>;

INSERT INTO ice_t
SELECT ..., col_i + 1, ...
FROM ice_t
WHERE <cond>;
```

The two statements must run on the same snapshot and commit together, so they don't interfere with each other.

But a DELETE statement is also an INSERT INTO statement (see above). Let's apply the rewrite:

```
INSERT INTO ice_t.DELETE (file_path, pos)
SELECT input__file__name, file__position, <part_cols>
FROM ice_t
WHERE <cond>;

INSERT INTO ice_t
SELECT ..., col_i + 1, ...
FROM ice_t
WHERE <cond>;
```

Seems like we could just merge them into a single INSERT INTO statement:

```
INSERT INTO ice_t.UPDATE (file_path, pos, ...)
SELECT input__file__name, file__position, ..., col_i + 1, ...
FROM ice_t
WHERE <cond>;
```

The update writer would split each row to two:

- file_path, pos
- ..., col_i + 1, ...

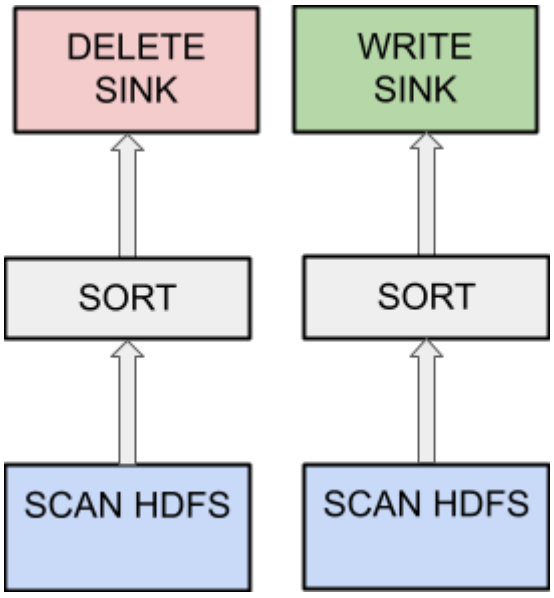
And write the first split into a delete file, and the second split into a regular data file.

Unfortunately it doesn't work in all cases. There is no problem if we update non-partition columns. We can shuffle data around by the partition columns which will be the same for the delete files and data files. But if a partition column gets updated, e.g.:

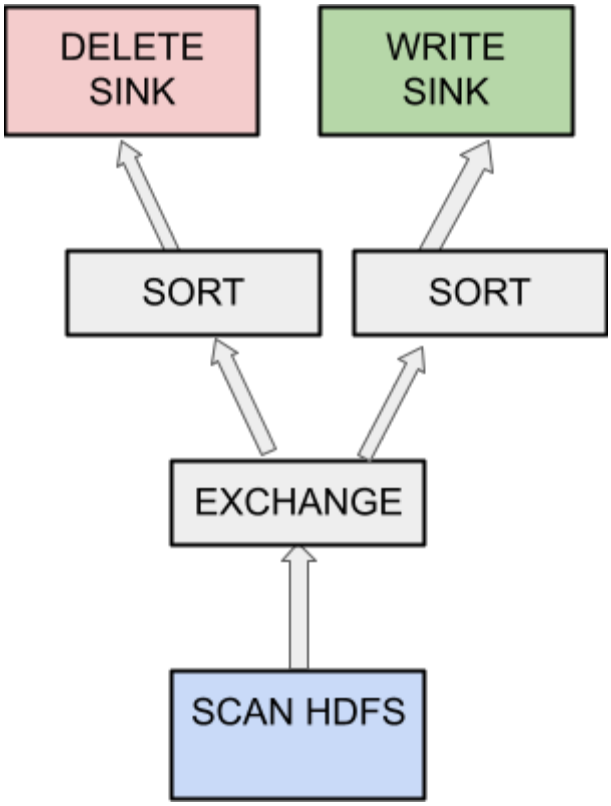
```
UPDATE ice_t SET part_col = part_col * 10 WHERE <cond>;
```

Then the delete records and the newly inserted records might belong to completely different partitions.
It's not easy to overcome this problem. We either

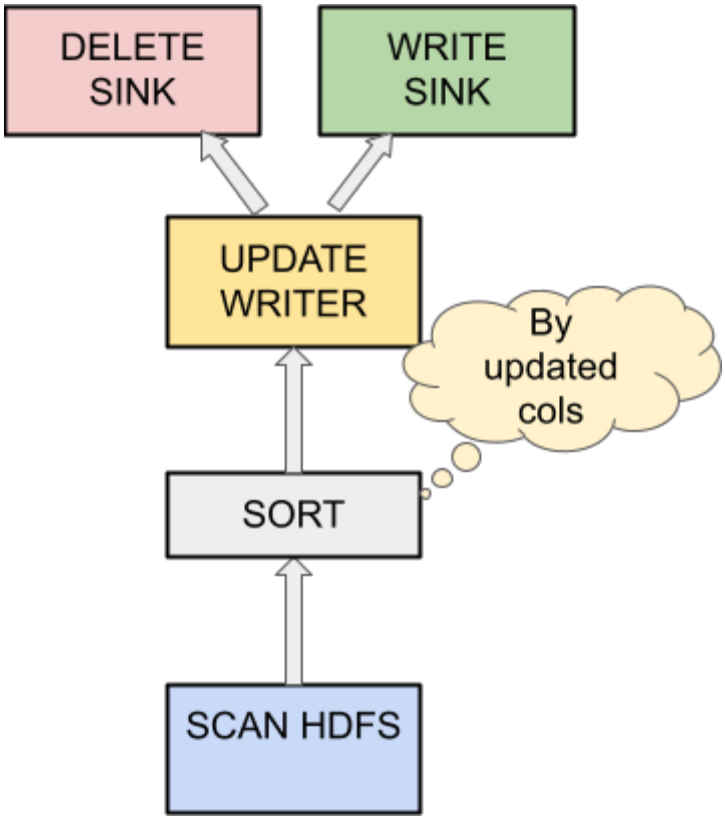
- Return back to the original DELETE + INSERT statements
    - It's wasteful, need to do things twice



- Fork the transmission of each row, driven by two partitioning
    - We don't have such operators in Impala
    - Not trivial to create such plans



- Shuffle only by the partitions of the newly inserted records
    - This is what we plan to implement, at least in the short term. Following parts of the document dive into this.
    - Each data partition is written by a single executor, i.e. as few data files are written as possible
    - Delete partitions might be written by multiple executors, i.e. more delete files are written than necessary
    - Hive does this: https://github.com/apache/hive/pull/3251/files



HIVE-26183: Implement UPDATE statements for Iceberg tables

So https://github.com/apache/hive/pull/3251/files does the latter, i.e. Hive update queries are converted to an insert statement where the result records contain the new column values and the deleted column values.
Rows are shuffled and sorted based on the new column values and HiveIcebergUpdateWriter writes both the new records and the deleted records. Deleted records are getting buffered in HiveIcebergBufferedDeleteWriter which outputs them during close().

Changes needed in Impala

(With the assumption that option “Shuffle only by the partitions of the newly inserted records” is chosen)
For DELETE statements, we could reuse the IcebergPositionDeleteTable virtual table which could be used as the target table of the DELETE - > INSERT statements. We might need to add the partition columns to its schema so that the INSERTs can write the delete files in separate partitions.
We need to create a delete writer in the backend. This probably needs to be tweaked so that it only outputs the file_path and pos columns. We might also need to SORT BY the file_path and position as well to write out rows one-by-one. Alternatively we can buffer the incoming rows then write out everything sorted at the end of each partition. Probably we should do the latter as we’ll need a buffering logic for UPDATEs anyway.

For UPDATE statements we can follow Hive's behavior (Shuffle only by the partitions of the newly inserted records). This means we need to create an UpdateWriter that splits the incoming tuples into two parts:

- New data part
  - <new columns>
- Position delete part
  - file_path
  - pos
  - <old columns>

(Probably we don't need to physically split the tuples, to avoid copying, we just need to track which slot belongs to where).

The incoming tuples to the UpdateWriter will be ordered based on the partitioning of the "New data part". I.e. the tuples can be passed one-by-one to a file writer, and the files need to be closed when a new tuple belongs to a new partition, and a new file is created for that partition, the same way we currently do partitioned inserts.

The "Position delete part" can be out of order. We can deal with it similarly to HIVE-26183, i.e. buffer the position delete data, and output the delete files at the very end. The memory requirements for this shouldn't be too high, as we would just need to store file names and the corresponding file positions. We could store them in an

```
unordered_map<Partition, map<FileName, vector<Position>>>
```

like structure. If this gets too large and we have tight memory limits then we could flush out the data belonging to the largest partitions.

Edge cases

Delete everything from a table:

```
DELETE FROM <tbl>;
```

This should be translated to:

```
TRUNCATE TABLE <tbl>;
```

So we don't need to write any new data files, just create a new empty snapshot.

Update every row:

```
UPDATE table_name SET column_name1 = new_value1,  column_name2 = new_value2 …;
```

This could be basically translated to:

```
INSERT OVERWRITE table_name SELECT new_value1, new_value2 … FROM table_name;
```

So we wouldn't write any delete files, just write the new data files. INSERT OVERWRITE does a dynamic overwrite with the help of Iceberg's ReplacePartitions API. It means only the affected partitions are replaced. Therefore, if we want to change columns that are involved in the

partitioning, then the above INSERT OVERWRITE statement would not work as expected (some old partitions might remain).

Therefore we need to invoke our truncateTable() method, then append the newly written data files to our table.