Proposal: Support for reading iceberg v2 table

Goals

1. Support for reading V2 Format

Non-Goals

1. Pushdown optimization of Delete files

BackGround

Iceberg V2 format is a format that supports row-level delete, which can support deleting row data that meets certain conditions. You can refer to https://iceberg.apache.org/spec/ to view the specific delete related spec. V2 mainly introduces delete files, including position delete and equality delete files. At present, there are more and more customers of V2, and the demand for V2 analysis of StarRocks is also increasing.

Position Delete Files

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	required struct<...> [1]	Deleted row values. Omit the column when not storing deleted rows.

Equality Delete Files

For example, a table with the following data:

```
 1: id | 2: category | 3: name
-------|-------------|---------
 1     | marsupial   | Koala
```

```
 2    | toy        | Teddy
 3    | NULL       | Grizzly
 4    | NULL       | Polar
```
The delete id = 3 could be written as either of the following equality delete files:

```
equality_ids=[1]

 1: id
-------
 3
equality_ids=[1]
```

```
 1: id | 2: category | 3: name
-------|-------------|---------
 3    | NULL       | Grizzly
```
The delete id = 4 AND category IS NULL could be written as the following equality delete file:

```
equality_ids=[1, 2]
```

```
 1: id | 2: category | 3: name
-------|-------------|---------
 4    | NULL       | Polar
```

Design

Iceberg internal implementation

1. DeleteFilter provides filter API, which can be called directly by the engine side; at the same time, it provides posAccessor that can return the recorded position information for subsequent processing; Delete Filter also support applyPosDeletes and applyEqDeletes

```
public abstract class DeleteFilter<T> {
 private static final long DEFAULT_SET_FILTER_THRESHOLD = 100_000L;
 private static final Schema POS_DELETE_SCHEMA = new Schema(
     MetadataColumns.DELETE_FILE_PATH,
     MetadataColumns.DELETE_FILE_POS);

 private final long setFilterThreshold;
 private final String filePath;
 private final List<DeleteFile> posDeletes;
 private final List<DeleteFile> eqDeletes;
 private final Schema requiredSchema;
 private final Accessor<StructLike> posAccessor;

private List<Predicate<T>> applyEqDeletes() {
 List<Predicate<T>> isInDeleteSets = Lists.newArrayList();
 if (eqDeletes.isEmpty()) {
   return isInDeleteSets;
 }

 Multimap<Set<Integer>, DeleteFile> filesByDeleteIds =
```

```
Multimaps.newMultimap(Maps.newHashMap(), Lists::newArrayList);
 for (DeleteFile delete : eqDeletes) {
   filesByDeleteIds.put(Sets.newHashSet(delete.equalityFieldIds()),
delete);
 }

 for (Map.Entry<Set<Integer>, Collection<DeleteFile>> entry :
filesByDeleteIds.asMap().entrySet()) {
   Set<Integer> ids = entry.getKey();
   Iterable<DeleteFile> deletes = entry.getValue();

   Schema deleteSchema = TypeUtil.select(requiredSchema, ids);

   // a projection to select and reorder fields of the file schema to match
the delete rows
   StructProjection projectRow = StructProjection.create(requiredSchema,
deleteSchema);

   Iterable<CloseableIterable<Record>> deleteRecords =
Iterables.transform(deletes,
       delete -> openDeletes(delete, deleteSchema));

   // copy the delete records because they will be held in a set
   CloseableIterable<Record> records = CloseableIterable.transform(
       CloseableIterable.concat(deleteRecords), Record::copy);

   StructLikeSet deleteSet = Deletes.toEqualitySet(
       CloseableIterable.transform(
           records, record -> new
InternalRecordWrapper(deleteSchema.asStruct()).wrap(record)),
       deleteSchema.asStruct());

   Predicate<T> isInDeleteSet = record ->
deleteSet.contains(projectRow.wrap(asStructLike(record)));
   isInDeleteSets.add(isInDeleteSet);
 }

 return isInDeleteSets;
}

public CloseableIterable<T> findEqualityDeleteRows(CloseableIterable<T>
records) {
 // Predicate to test whether a row has been deleted by equality deletions.
 Predicate<T> deletedRows = applyEqDeletes().stream()
     .reduce(Predicate::or)
     .orElse(t -> false);

 Filter<T> deletedRowsFilter = new Filter<T>() {
   @Override
   protected boolean shouldKeep(T item) {
     return deletedRows.test(item);
   }
```

```
  };
  return deletedRowsFilter.filter(records);
}

private CloseableIterable<T> applyEqDeletes(CloseableIterable<T> records) {
  // Predicate to test whether a row should be visible to user after
applying equality deletions.
  Predicate<T> remainingRows = applyEqDeletes().stream()
      .map(Predicate::negate)
      .reduce(Predicate::and)
      .orElse(t -> true);

  Filter<T> remainingRowsFilter = new Filter<T>() {
    @Override
    protected boolean shouldKeep(T item) {
      return remainingRows.test(item);
    }
  };

  return remainingRowsFilter.filter(records);
}

private CloseableIterable<T> applyPosDeletes(CloseableIterable<T> records)
{
  if (posDeletes.isEmpty()) {
    return records;
  }

  List<CloseableIterable<Record>> deletes = Lists.transform(posDeletes,
this::openPosDeletes);

  // if there are fewer deletes than a reasonable number to keep in memory,
use a set
  if (posDeletes.stream().mapToLong(DeleteFile::recordCount).sum() <
setFilterThreshold) {
    return Deletes.filter(records, this::pos,
Deletes.toPositionIndex(filePath, deletes));
  }

  return Deletes.streamingFilter(records, this::pos,
Deletes.deletePositions(filePath, deletes));
}

private CloseableIterable<Record> openPosDeletes(DeleteFile file) {
  return openDeletes(file, POS_DELETE_SCHEMA);
}
}
```

2. StructLike provides get API for reading specific fields

```
public interface StructLike {
  int size();
```

```
  <T> T get(int pos, Class<T> javaClass);

  <T> void set(int pos, T value);
}
```

3. Position Delete
    - The Position Delete file stores deleted row information
    - In the call to DeleteFilter.filter, applyPosDeletes will be done
    - The actual logic of applyPosDeletes is to filter the data according to the position information stored in the delete file and the recorded position information.
4. Equality Delete
    - The Equality Delete file stores the corresponding predicate conditions
    - In the call to DeleteFilter.filter, applyEqDeletes will be done
    - applyEqDeletes first generates a memory filter condition based on the Equality Delete file, which is an and predicate; then use this predicate to compare the record content to filter the data

Solution

// Comments welcome

1. BE defines StarRocksRow, which holds Chunk objects to avoid row-column conversion
2. FE obtains the relevant information of the delete file and sends it to BE to generate DeleteFilter
3. Timing StarRocksDeleteFilter, there are two schemes
    a. JNI calls Java API
    b. C++ rewrites the logic of Delete Filter

Reference

- **Trino** https://github.com/trinodb/trino/pull/11642/files
- Iceberg v2 design doc https://docs.google.com/document/d/1Pk34C3diOfVCRc-sfxfhXZfzvxwum1Odo-6Jj9mwK38/edit#heading=h.g4del2n8m0hv