FLIP-282: Introduce Delete & Update API

Status

Current state: Accepted

Discussion thread: https://lists.apache.org/thread/6h64v7v6g916pkvmc3q3vxxccr46r3

Vote thread: https://lists.apache.org/thread/3m91zys8dgnzvypq88qkb2t55iqvgh1

JIRA: [FLINK-30648 - [Umbrella]FLIP-282 Introduce Delete/Update API] [CLOSED]

Released: 1.17

Please keep the discussion on the mailing list rather than commenting on the wiki (wiki discussions get unwieldy fast).

Motivation

Row-Level SQL Delete & Update are becoming more and more important in modern big data workflow. The use cases include deleting a set of rows for regulatory compliance, updating a set of rows for data correction, etc. Many popular engines such as Trino, Hive have supported them. But they are still missing in Flink.

So, in this FLIP, we try to introduce the Delete & Update API to Flink works in batch mode which will be exposed to connectors. And then to support delete & update the rows in the external storage system, the corresponding connectors need to implement these interfaces.

Note: Considering the use cases of delete & update are mainly for batch scenario and the semantic in stream scenario should be discussed separately, we only consider to support them in batch mode in this FLIP. For stream mode, it will throw TableException with the error message "DELETE /UPDATE TABLE is not supported for streaming mode now" for delete/update statement.

Delete & Update Examples

Here, before further discussion, we list some SQL examples for Delete & Update.

Delete

```sql
DELETE FROM user WHERE id = -1;
DELETE FROM user WHERE id > (SELECT count(*) FROM employee);
```

Update

```sql
UPDATE user SET name = "u1" WHERE id > 10;
UPDATE user SET name = "u1" WHERE id > (SELECT count(*) FROM employee);
```

Public Interfaces

We propose to introduce the following interfaces for delete & update:
/**
 * The context for scan the table to do row-level modification like updated/delete, designed to provide some
 * necessary information that the sink may need to know to perform update/delete. It'll be generated by table
 * source which implements {@link SupportsRowLevelModificationScan}, and then passed to the sink
 * which implements {@link SupportsRowLevelDelete} or {@link SupportsRowLevelUpdate} for
 * UPDATE/DELETE statement in compile phase.
 * <p>This mechanism enables the coordination between the source and sink for the corresponding
 * table to be updated/deleted.
 * <p>The connector can implement this interface to provide custom information.
 */

@PublicEvolving
public interface RowLevelModificationScanContext {}
public interface SupportsRowLevelModificationScan {

    RowLevelModificationScanContext applyRowLevelModificationScan(
        RowLevelModificationType rowLevelModificationType,
        @Nullable RowLevelModificationScanContext previousContext);

    enum RowLevelModificationType {
        UPDATE, DELETE
    }
}

The above interfaces are what the source is expected to implement. But it's optional, it's not necessary that the source implements these interfaces.

Although the code comments for the above interfaces has explained what the interfaces are for. But in here we would like to explain a bit more for why we introduce these interfaces.

First, after we introduce delete / update, the table scan will have diverse purposes, it'll be beneficial to tell the table source scan whether the table is for update/delete. The table scan may have behavior for different purpose. Also it's import for back compatibility.

For example, in the current implementation, the iceberg table source will get a snapshot id in the running phase and use the snapshot id to scan the table. But for delete or update, it may expect to use the snapshot id generated in compiling phase. So, if the table source scan can know the purpose, it can avoid the compatibility issue.

Second, for the row-level modification such as delete/update, it's important that the table source scan have the ability to pass some values to the sink. Take delete for Iceberg table as an example, iceberg source scan will first load the current snapshot id and then use this snapshot id to scan data, and then when the iceberg sink does the commit to finish delete, it'll need the snapshot id that the source used to do validation.
Delete

/**
 * Enable to delete existing data in a [DynamicTableSink] directly according to the filter
 * expressions in [DELETE] clause.
 *<p>Given the following SQL:
 *<pre>@code
 * DELETE FROM t WHERE (a = '1' OR a = '2') AND b IS NOT NULL;
 *</pre>
 *<p>In the example above, [a = '1' OR a = '2'] and [b IS NOT NULL] are the filters.
 *<p>Flink will get the filters in conjunctive form and push down the filters into sink by calling
 * method [applyDeleteFilters] in the planner phase. If it returns true, Flink will then call
 * method [executeDeletion] to execute the actual delete in execution phase.
 *<p>Note: in the case that the filter expression is not available, e.g., sub-query or [applyDeleteFilters]
 * returns false, if the sink implements [SupportsRowLevelDelete], Flink
 * will try to rewrite the delete operation and produce row-level changes, see [SupportsRowLevelDelete]
 * for more details. Otherwise, Flink will throw [UnsupportedOperationException].
 */
@PublicEvolving
public interface SupportsDeletePushDown {

    /**
     * Provides a list of filters from delete operation in conjunctive form in planning phase. A
     * sink can either return true if it can accept all filters or return false if it cannot
     * accept.
     *
     * If it returns true, Flink will then call the method [executeDeletion] in execution
     * phase to do the actual deletion.
     *
     * If it returns false, and the sink still implements [SupportsRowLevelDelete], Flink
     * will rewrite the delete operation and produce row-level changes. Otherwise, Flink will throw
     * [UnsupportedOperationException].
     */
    boolean applyDeleteFilters(List<ResolvedExpression> filters);

    /**
     * Do the actual deletion in the execution phase, and return how many rows has been deleted,
     * Optional.empty() is for unknown delete rows.
     *
     * Note that this method will be involved if and only if [applyDeleteFilters] returns true. So, please make sure the implementation for this method
     * will do delete the data correctly.
     *
     * Note that in this method, the sink won’t get the filters since they have been passed to
     * the method [applyDeleteFilters] before. The sink may need to keep these filters, so
     * that it can get the filters if necessary to finish the deletion.
     */
    Optional<Long> executeDeletion();
}

/**
 * Enable to delete existing data in a [DynamicTableSink] according to row-level changes.
 *<p>The planner will call method [applyRowLevelDelete] to get the [RowLevelDeleteInfo]
 * that the sink returns, and rewrite the delete operation based on gotten

* {@link RowLevelDeleteInfo} to produce rows (may rows to be deleted or remaining rows after the delete operation depending on sink's implementation) to {@link DynamicTableSink}. The sink is expected to consume these rows to achieve rows delete purpose.

* `<p>Note: if the sink also implements {@link SupportsDeletePushDown}, the planner will always prefer {@link SupportsDeletePushDown}, and only the filters aren't available or {@link SupportsDeletePushDown#applyDeleteFilters} returns false, this interface will be considered and to rewrite the delete operation to produce the rows to the sink.

*/

@PublicEvolving
public interface SupportsRowLevelDelete {

/**
* Apply row level delete with {@link RowLevelModificationScanContext} passed by table source, and return {@link RowLevelDeleteInfo} to guide the planner on how to rewrite the delete operation.
* `<p>Note: if the table source doesn't implement {@link SupportsRowLevelModificationScan}, the `{@param context}` will be null.
*/
RowLevelDeleteInfo applyRowLevelDelete(@Nullable RowLevelModificationScanContext context);

/** The information that guides the planer on how to rewrite the delete operation. */
@PublicEvolving
interface RowLevelDeleteInfo {

/**
* The required columns that the sink expects for deletion, the rows consumed by sink will contain the columns with the order consistent with the order of returned columns. If return Optional.empty(), it will select all columns.
*/
default Optional<@Column> requiredColumns() {
    return Optional.empty();
}

/**
* Planner will rewrite delete to query base on the {@link RowLevelDeleteInfo}, keeps the query of delete unchanged by default(in `DELETE_ROWS` mode), or change the query to the complementary set in REMAINING_ROWS mode.
* `<p>Note: All rows will have RowKind#DELETE when RowLevelDeleteMode is DELETED_ROWS, and RowKind#INSERT when RowLevelDeleteMode is REMAINING_ROWS.
*/
default RowLevelDeleteMode getRowLevelDeleteMode() {
    return RowLevelDeleteMode.DELETED_ROWS;
}

/**
* Type of delete modes that the sink expects for delete purpose.
* `<p>Currently, two modes are supported:
* `<ul>
* `<li>DELETED_ROWS - in this mode, the sink will only get the rows that need to be deleted.
* `<li>REMAINING_ROWS - in this mode, the sink will only get the remaining rows as if the delete operation had been done.
*/
Update

/**
* Enable to update existing data in a [link DynamicTableSink] according to row-level changes.
* The planner will call method [link #applyRowLevelUpdate] to get the [link RowLevelUpdateInfo] that the sink returns, and rewrite the update operation based on gotten [link RowLevelUpdateInfo] to produce rows (may rows to be updated with update after values or all rows including both rows be updated and the other rows without update need depending on sink's implementation) to [link DynamicTableSink]. The sink is expected to consume these rows to achieve rows update purpose.
*/
@PublicEvolving
public interface SupportsRowLevelUpdate {

    /**
     * Apply row level update with providing the updated columns and [link RowLevelModificationScanContext] passed by table source, then return [link RowLevelUpdateInfo] to guide the planner on how to rewrite the update operation.
     * @param updatedColumns the columns updated by update operation in table column order.
     * @param rowLevelModificationScanContext the context passed by table source which implement [link SupportsRowLevelModificationScan], if the table source doesn't implement this interface, the rowLevelModificationContext will be null.
     */
    RowLevelUpdateInfo applyRowLevelUpdate(List<Column> updatedColumns, @Nullable RowLevelModificationScanContext context);

    /**
     * The information that guides the planner on how to rewrite the update operation.
     */
    @PublicEvolving
    interface RowLevelUpdateInfo {

        /**
         * The required columns that the sink expects for deletion, the rows consumed by sink will contain the columns with the order consistent with the order of returned columns. If return Optional.empty(), it will select all columns.
         */
        default Optional<List<Column>> requiredColumns() {
            return Optional.empty();
        }
    }

    /**
     * Planner will rewrite the update operation to query base on the [link RowLevelUpdateMode], keeps the query of update unchanged by default (in `UPDATED_ROWS` mode), or change the query to union the updated rows and un-updated rows (in `ALL_ROWS` mode).
     * <p>Take the following SQL as an example:
     * <pre><![CDATA[
     * UPDATE t SET x = 1 WHERE y = 2;
     *]]></pre>
     * If returns [link RowLevelUpdateMode#UPDATED_ROWS], the sink will get the update after rows which match the filter \[y = 2\].
     * If returns [link RowLevelUpdateMode#ALL_ROWS], the sink will get the update after rows which match the filter \[y = 2\] as well as the other rows that don’t match the filter \[y = 2\].
     */
}
The above interfaces are what the sink is expected to implement.

Proposed Changes

The basic idea to implement delete & update is first query the rows needed for delete & update purpose and then produce these rows to the sink which is responsible to perform delete & update according to the produced rows.

How to produce needed rows

We will translate the delete & update statement into a plan that inserts into the target table with the rows that need to be deleted & updated or all the rows assuming the delete & update has been performed. The connector sink will have the flexibility to define what columns the rows will contain and what type of rows. The detail explanation is as follows:

Delete

If the interface SupportsDeletePushDown is implemented, in the planning phase, the planner will first call the method SupportsDeletePushDown#applyDeleteFilters to try to push down the filters corresponding to the delete statement to the connector. If the connector can directly perform delete according to the pushed filters, it should return true. And then in execution phase, Flink will call the method SupportsDeletePushDown#delete in which method the sink is expected to do the actual delete.

Otherwise SupportsDeletePushDown isn’t implemented, or filters are not available, or SupportsDeletePushDown#applyDeleteFilters returns false, the planner will then call the method SupportsRowLevelDelete#applyRowLevelDelete to get the needed information wrapped in RowLevelDeleteInfo to do the plan rewrite to produce row-level changes.

There’re two things the planner may need to know:

1: The first one is RowLevelDeleteInfo#requiredColumns, it means which columns need to be selected. The default implementation will return Optional.empty(), if return Optional.empty() the planner will get all the columns. But in most cases, to perform delete, the connector only needs to select the primary key columns, so the connector implementation can overwrite it and just return the primary key columns.

2: The second one is RowLevelDeleteInfo#getRowLevelDeleteMode, it means in which mode we rewrite the plan. Currently, two modes are supported:

• DELETED_ROWS

   In this mode, the planner will only select the rows that need to be deleted. For example

   DELETE FROM user WHERE id = -1;
will produce the rows like they're generated by the following SQL:

```sql
SELECT id, name FROM user WHERE id = -1;
```

If the `RowLevelDeleteMode#requiredColumns` returns column `id`, then it looks like:

```sql
SELECT id FROM user WHERE id = -1;
```

Considering this mode is more common and more straightforward, we use this mode as the default delete mode.

- **REMAINING_ROWS**

  In this mode, the planner will only select the rows that won't be deleted. They're the rows remained as if the delete statement had been executed. For example:

```sql
DELETE FROM user WHERE id = -1;
```

will produce the rows like they're generated by the following SQL:

```sql
SELECT id, name
FROM user
WHERE NOT(id = -1);
```

Such a mode is useful for copy-on-write tables or writing the remaining rows costs less than writing delete rows.

**Update**

Update is similar to delete, but the planner will also pass the updated columns in UPDATE statement to the connector in case some connectors may need the updated columns. The planner will rewrite the update operation to produce row-level changes.

There're two things the planner may need to know:

1: The first one is `SupportsRowLevelUpdate#requiredColumns`, it means which columns need to be selected. The default implementation will return `Optional.empty()`, if return `Optional.empty()` the planner will get all the columns. But in some partial update cases, to perform the update, the connector only needs to get the primary key columns and the updated columns, so the connector implementation can overwrite it and just return the primary key columns + updated columns.

2: The second one is `RowLevelUpdateInfo#getRowLevelUpdateMode`, it means in which mode we rewrite the plan. Currently, two modes are supported:

- **UPDATED_ROWS**

  In this mode, the planner will only select the rows that need to be updated. The rows will be the update after rows. For example:

```sql
UPDATE user SET name = "u1" WHERE id > 10;
```

will produce the rows like they're generated by the following SQL:

```sql
SELECT id, "u1" FROM user WHERE id > 10;
```

Considering this mode is more common use and more straightforward, so here, we use this mode as the default mode for update.

- **ALL_ROWS**

  In this mode, the planner will select both update after rows to be updated and the other rows that don't need to be updated. Those are all the rows as if the update statement had been executed. For example:

```sql
UPDATE user SET name = "u1" WHERE id > 10;
```

will produce the rows like they're generated by the following SQL:

```sql
SELECT id, IF(id > 10, "u1", name) FROM user;
```
How the connector's sink do the delete & update

As said in the previous section, Flink will produce the rows needed to the connector's sink just like inserting rows into a sink, and it's the connector's responsibility to determine how to handle these rows to achieve delete & update.

Delete

For delete, the sink can implement SupportsDeletePushDown, SupportsRowLevelDelete or both of them. If SupportsDeletePushDown is implemented, the method SupportsDeletePushDown#applyDeleteFilters will be called firstly, then the sink can determine whether it can perform deleting rows directly according to the filters. If the method SupportsDeletePushDown#applyDeleteFilters returns true, Flink will call method SupportsDelete#delete in execution phase. So in this method, the sink needs to do the actual delete if SupportsDelete#applyDeleteFilters return true.

The sink can also implement SupportsRowLevelDelete for the case that SupportsDeletePushDown is not applicable. For the cases, if SupportsRowLevelDelete is implemented, the method SupportsRowLevelDelete#applyRowLevelDelete will be called, the connector sink is expected to return RowLevelDeleteInfo to provide guidance for Flink to produce rows. By default, Flink will produce all the rows to be deleted with all columns. Also, in this method, the DynamicTableSink may need to mark it as the coming rows are for delete purpose.

Then, the sink will get a datastream of RowData, and the sink can do deletion according to these rows. The code may look like:

```java
public class TSink implements DynamicTableSink, SupportsRowLevelDelete {

    ....

    @Override
    public SinkRuntimeProvider getSinkRuntimeProvider(Context context) {
        return new DataStreamSinkProvider() {
            @Override
            public DataStreamSink<?> consumeDataStream(ProviderContext providerContext, DataStream<RowData> dataStream) {
                // consume the rows for delete purpose
            }
        };
    }
}
```

Update

For update, the sink needs to implement SupportsRowLevelUpdate. The implementation for update is similar to delete. The method SupportsRowLevelUpdate#applyRowLevelUpdate(List<Column> updatedColumns) will be called. The connector sink is expected to return a RowLevelUpdateInfo to provide guidance for Flink to produce rows. By default, Flink will produce all the update after rows with all columns.

Also, in this method, the DynamicTableSink may need to mark it as the coming rows are for update purpose.

Then, the sink will get a datastream of RowData, and the sink can do update according to these rows. The code may look like:

```java
public class TSink implements DynamicTableSink, SupportsDelete {

    ....

    @Override
    public SinkRuntimeProvider getSinkRuntimeProvider(Context context) {
        return new DataStreamSinkProvider() {
            @Override
            public DataStreamSink<?> consumeDataStream(ProviderContext providerContext, DataStream<RowData> dataStream) {
                // consume the rows for delete purpose
            }
        };
    }
}
```
Message passing from source to sink

In the case that source may need pass some messages to sink, the source can implement the interface SupportsRowLevelModificationScan to provide a RowLevelModificationScanContext which contains required by sink, the code will look like:

```java
-- Custom a ModificationScanContext
public MyTableModificationScanContext implements RowLevelModificationScanContext {
    private Map<ObjectIdentifier, Long> snapshots = new HashMap<>();

    Map<ObjectIdentifier, Long> getSnapshots() {
        return snapshots;
    }
}

-- Source
public class TSource implements ScanTableSource, SupportsRowLevelModificationScan {
    private RowLevelModificationType type;

    RowLevelModificationScanContext applyRowLevelModificationScan(
        RowLevelModificationType rowLevelModificationType,
        @Nullable RowLevelModificationScanContext previousContext)
    {
        if (previousContext == null) {
            // no previous context, first table source
            previousContext = new MyTableModificationScanContext();
        }

        // mark the scan type
        this.type = rowLevelModificationType;
        // put some values
        scanContext.getSnapshots().put(xxx, xxx);
    }
}

-- Sink
public class TSink implements DynamicTableSink, SupportsRowLevelDelete {

    RowLevelDeleteInfo applyRowLevelDelete(@Nullable RowLevelModificationScanContext context) {
        MyTableModificationScanContext myContext = (MyTableModificationScanContext) context;
        // get the snapshot info
        myContext.getSnapshots().get(xxx);
    }
}
```

Compatibility, Deprecation, and Migration Plan

It won't break any compatibility.

Test Plan

1: The changes will be covered by unit and IT cases.
2: Also, we expect to choose one connector, maybe the JDBC connector to implement the delete and update.

Rejected Alternatives

1: Add a sort requirements in SupportsRowLevelDelete

Some external storage systems may expect the data is sorted to do better delete. For example, Iceberg expects the rows are sorted by `<file_path, position>` to write position-delete file. Although we can add an interface to make the connector capable of telling Flink how to sort these rows, it's a little weird to add it to this FLIP. Also, sort requirements are orthogonal to delete. Ideally, we need another FLIP that supports different distribution requirements before coming to sink.
So, in this FLIP, we remove the sort requirement interface to avoid complexity and future potential conflicts.

2: About message passing from source to sink.

Again, take passing the snapshot id from source to sink as an example. In interfaces design, we can add a parameter in method `SupportsRowLevelDelete#applyRowLevelDelete(ScanTableSource scanTable)` to pass the tableScan to sink. And then, the sink can tell tableScan to generate a snapshot id and get the snapshot id.

But it's hack and error prone. Also, it'll expose too much to the sink.