**FLIP-308: Support Time Travel**

*Document the state by adding a label to the FLIP page with one of "discussion", "accepted", "released", "rejected".*

<table>
<thead>
<tr>
<th>Discussion thread</th>
<th><a href="https://lists.apache.org/thread/rpozdlf7469jmcm7q7vc0s08pjmmscz00">https://lists.apache.org/thread/rpozdlf7469jmcm7q7vc0s08pjmmscz00</a></th>
</tr>
</thead>
<tbody>
<tr>
<td>Vote thread</td>
<td><a href="https://lists.apache.org/thread/d8g8dmktgms014ljqtz9mx1dsch8q0">https://lists.apache.org/thread/d8g8dmktgms014ljqtz9mx1dsch8q0</a></td>
</tr>
<tr>
<td>JIRA</td>
<td><img src="FLINK-32472.png" alt="FLINK-32472 - FLIP-308: Support Time Travel RESOLVED" /></td>
</tr>
<tr>
<td>Release</td>
<td>1.18</td>
</tr>
</tbody>
</table>

- **Motivation**
- **Public Interfaces**
  - **Syntax**
    - Query the data for a past moment
    - Query the data for the current moment
    - Supported Syntax and Types of Timestamp Expression
  - **Public interfaces**
    - Catalog
    - CatalogTable
- **Proposed Changes**
  - **Flink SQL related Changes**
    - 1. Convert SQL String to SqlNode and Validate
    - 2. SqlNode To RelNode.
    - 3. Optimizer
    - 4. Convert RelNode To Transformation (Convert CommonExecTableSourceScan To Transformation)
  - **Integrate with other systems**
- **Compatibility, Deprecation, and Migration Plan**
- **Test Plan**
- **Whether to support other syntax implementations**
- **Rejected Alternatives**
  - Add SupportsTimeTravel source ability

**Motivation**

Time travel is a SQL syntax for querying historical versions of data that allows users to specify a point in time and retrieve the data and schema of a table as it appeared at that time. With time travel, users can easily analyze and compare historical versions of data.

**Public Interfaces**

**Syntax**

We propose add the following syntax for time travel statement. In the SQL 2011 standard[1], syntax related to Time Travel is defined.

The following syntax supports both batch mode and bounded stream mode.

**Query the data for a past moment.**

```
-- To specify a specific time, use the following syntax:
SELECT [column_name(s)] FROM [table_name] FOR SYSTEM_TIME AS OF TIMESTAMP '[timestamp]'

-- For example: SELECT * FROM paimon_tb FOR SYSTEM_TIME AS OF 'TIMESTAMP '2023-04-27 00:00:00'
```

```
-- To specify a constant expression, use the following syntax:
SELECT [column_name(s)] FROM [table_name] FOR SYSTEM_TIME AS OF [expression]

-- For example: SELECT * FROM paimon_tb FOR SYSTEM_TIME AS OF 'TIMESTAMP '2023-04-27 00:00:00' - INTERVAL '1' DAY
```
Query the data for the current moment

-- By default, if no time is specified, the system will query data from the latest timestamp.

SELECT [column_name(s)] FROM [table_name]

-- For example: SELECT * FROM paimon_tb;

-- Specify a current-time expression

SELECT [column_name(s)] FROM [table_name] FOR SYSTEM_TIME AS OF [current_time expression];

-- For example: SELECT * FROM paimon_tb FOR SYSTEM_TIME AS OF CURRENT_TIMESTAMP;

Supported Syntax and Types of Timestamp Expression

1. Supported Timestamp type

As shown in the above syntax, when querying with time travel, we support specifying a constant expression of time. Therefore, the type corresponding to this expression may be TIMESTAMP or TIMESTAMP_LTZ.

But the interface reserved for the Connector layer is a long value timestamp(The newly added method for the Catalog is as follows) which means that we need to convert both TIMESTAMP and TIMESTAMP_LTZ expressions into long values in the end.

By default, it is natural to convert data of type TIMESTAMP_LTZ to Long value. However, if it is a TIMESTAMP type, we will implicitly convert it to TIMESTAMP_LTZ and then convert it to a Long value.

2. Supported Expression Syntax

<TIMESTAMP EXPRESSION> must be a valid and executable expression.

For example, we will support TIMESTAMP '2023-05-05 00:00:00', but not support TIMESTAMP_LTZ '2023-05-05 00:00:00'. If you need to specify the TIMESTAMP_LTZ type, you need to use TO_TIMESTAMP_LTZ function.

Public interfaces

Catalog

@PublicEvolving
public interface Catalog {

    /**
     * Returns a [CatalogTable] or [CatalogView] identified by the given [ObjectPath]. The framework will resolve the metadata objects when necessary.
     * @param tablePath Path of the table or view
     * @param timestamp Timestamp of the table snapshot, which is milliseconds since 1970-01-01 00:00:00 UTC
     * @return The requested table or view
     * @throws TableNotExistException if the target does not exist
     * @throws CatalogException in case of any runtime exception
     */
    default CatalogBaseTable getTable(ObjectPath tablePath, long timestamp)
        throws TableNotExistException, CatalogException {
            throw new UnsupportedOperationException(  
                String.format("Table %s does not support time travel.", tablePath));
        }   

}
## Proposed Changes

### Flink SQL related Changes

Here is the poc of the FLIP: [https://github.com/apache/flink/compare/master...hackergin:flink:FLIP-308](https://github.com/apache/flink/compare/master...hackergin:flink:FLIP-308)

I will describe the necessary modifications from several stages of SQL compilation and parsing.

1. **Convert SQL String to SqlNode and Validate**

At this stage, the schema of the table will be verified, so when parsing the SqlSnapshot node, the corresponding `peroid` (maybe a time constant or an expression) can be converted into a timestamp, and a LongSchemaVersion is constructed to generate a new The calcite schema for schema validate.

2. **SqlNode To RelNode.**

At this stage, the SQL schema will also be verified, so when converting the SqlSnapshot node, the timestamp of the obtained SqlSnapshot can be parsed, and the LongSchemaVersion will be constructed to generate a new Calcite Schema for table validation.

3. **Optimizer**

At present, Flink SQL only supports temporol join, so related detection items are added to avoid the use of unsupported syntax. The current implementation of time travel should remove these restrictions.

4. **Convert RelNode to Transformation (Convert CommonExecTableSourceScan To Transformation)**

If a connector needs to support TimeTravel, the corresponding timestamp can be obtained through the `getSnapshot` interface of CatalogTable when constructing a DynamicTableSource.

```java
@Override
public DynamicTableSource createDynamicTableSource(Context context) {
    Optional<Long> snapshot = context.getCatalogTable().getSnapshot();
    ...
}
```

### Integrate with other systems

This section will describe how other systems can integrate to support TimeTravel.

Taking the support of Paimon for Timetravel as an example, when implementing the `getTable(ObjectPath tablepath, long timestamp)` method, we will specify the corresponding consumption timestamp.
/** Catalog for paimon. */
public class FlinkCatalog extends AbstractCatalog {

  ...

  @Override
  public CatalogTable getTable(ObjectPath tablePath)
      throws TableNotExistException, CatalogException {
    Table table;
    try {
      table = catalog.getTable(toIdentifier(tablePath));
    } catch (Catalog.TableNotExistException e) {
      throw new TableNotExistException(getName(), tablePath);
    }

    if (table instanceof FileStoreTable) {
      return toCatalogTable(table);
    } else {
      return new SystemCatalogTable(table);
    }
  }

  @Override
  public CatalogTable getTable(ObjectPath tablePath, long timestamp)
      throws TableNotExistException, CatalogException {
    // Due to Paimon's Catalog not providing an interface for historical Schema,
    // we can only obtain the latest Schema at present.
    CatalogTable catalogTable = this.getTable(tablePath);

    // Attach the specified consumption time parameter.
    Options option =
        new Options()
            .set(CoreOptions.SCAN_TIMESTAMP_MILLIS, timestamp);

    return catalogTable.copy(option.toMap());
  }

  ....
}

Compatibility, Deprecation, and Migration Plan

This is a newly added feature, so there will be no compatibility issues

Test Plan

UT&IT

Whether to support other syntax implementations

Due to the current limitations of Calcite syntax, we can only support specifying a single time and cannot specify a time range.
SqlSnapshot Snapshot(SqlNode tableRef) {
    final Span s;
    final SqlNode e;
}

{ s = span(); } <FOR> <SYSTEM_TIME> <AS> <OF>
// Syntax for temporal table in
// standard SQL 2011 IWD 9075-2:2017(E) 7.6 <table reference>
// supports grammar as following:
// 1. datetime literal
// 2. datetime value function, i.e. CURRENT_TIMESTAMP
// 3. datetime term in 1 or 2 +(or -) interval term

// We extend to support column reference, use Expression
// to simplify the parsing code.
    e = Expression(ExprContext.ACCEPT_NON_QUERY) {
        return new SqlSnapshot(s.end(this), tableRef, e);
    }
}

Currently, we can't support the following syntax.

```
SELECT ENo,EName,Sys_Start,Sys_End
FROM Emp FOR SYSTEM_TIME FROM
TIMESTAMP '2011-01-02 00:00:00'TO
TIMESTAMP '2011-12-31 00:00:00'
```

OR

```
SELECT ENo,EName,Sys_Start,Sys_End
FROM Emp FOR SYSTEM_TIME BETWEEN
TIMESTAMP '2011-01-02 00:00:00'AND
TIMESTAMP '2011-12-31 00:00:00'
```

Rejected Alternatives

TimeTravel is not only an ability of Source connector, but also needs support from Catalog. Travel not only requires obtaining data at the corresponding time point, but also requires the corresponding Schema at that time point.

As a Source ability, SupportTimeTravel will be applied during SQL optimization, which means that it can't meet the requirement for the TableSchema TimeTravel.

Add SupportsTimeTravel source ability

Here is the brief implement plan.

1. Add the Source Ability interface to support TimeTravel systems in implementing TimeTravel capabilities.
2. Add corresponding table optimization rules so that during LogicalPlan optimization, the Snapshot interval can be pushed up to the ScanTableSource.
3. To avoid conflicts with Temporal Join, we should apply TimeTravel transformation rules before apply the Temporal Join transformation rules.
/**
 * Enables to push down the timestamp of snapshot into a ScanTableSource.
 * <p>When a source connector implements the SupportsTimeTravel interface, users can leverage its time-travel functionality to analyze historical data, perform backfilling operations, and more. This can be particularly useful for debugging and data recovery purposes.</p>
 */

@PublicEvolving
public interface SupportsTimeTravel {

    /**
     * @param timestamp The timestamp of the snapshot to apply.
     */
    void applySnapshot(long timestamp);
}