Permanent Lookup Table

- Status: Design
- Author: Jiyong Huang
- Discussion: https://github.com/lf-edge/ekuiper/issues/1120

Motivation

Currently, we only support to join the snapshot of stream called table in memory. That is only suitable for small amount of data. When doing enrichment, it is desirable to join a permanent data source like database. The goal of this feature is to support join data from a permanent place and do the lookup on demand without loading all the data into the memory.

Concepts

- Stream: a sequence of events. When doing stream processing, the processing happens for each event.
- Table: a snapshot of the accumulation of events.
- Temporary table: the table in memory and will only exist in the session.
- Permanent table: a table that is bound to a physical data source like db. It will exist and be visible by all sessions.
- Scan source: aka. stream, the current source approach. A changelog (finite or infinite) for which all changes are consumed continuously until the changelog is exhausted.
- Lookup source: aka. table, a continuously changing or very large external table whose content is usually never read entirely but queried for individual values when necessary.
- Updatable sink: By default, the sink only support `insert`/`append`. For change data capture (CDC) scenarios, the sink can write out bounded or unbounded streams with insert, update, and delete rows.

Updatable table for MQTT: do we support such source or let it read from a memory sink? Or create a `permanent` memory table which can be accessed by all rules

Another problem: for scan source, how to deal with table/stream from the same topic. We can hardly guarantee the events arrived at the same time.

Proposals

1. Permanent lookup source for table: support to bind physical table as `table` and generate lookup command(e.g. a SQL on db) on demand.
2. Updatable sink: supports `insert`, `delete` and `update` for available sink like SQL sink and memory sink.

Permanent Lookup Table

Provide a way to bind the `table` to a physical data source like db table. This is useful for enriching stream data.

Apparently, only a few of sources is suitable as lookup table which requires the source itself is queryable. The potential supported sources are:

- Memory source: if a memory source is used as table type, we need to accumulate the data as a table in memory. This source can be a converter between scan source and lookup source. Thus, it is untypical.
- File source: it is possible to serve as a lookup table. Need an efficient way to query.

Suitable shipped source plugins:

- SQL source: This is the most typical lookup source. We can use SQL directly to query. The implementation of permanent lookup table will use this as the target source.

Usage

We can reuse the table statements to create tables, but now we have two table types: permanent and temporary. The usage is almost the same as previous table.

1. Create a stream.
   ```
   CREATE STREAM demoStream() WITH (DATASOURCE="demo", FORMAT="json", TYPE="mqtt")
   ```
2. Create permanent table. In the source configuration yaml, the database connection must be configured. This can map to a **real** table in the db.
   ```
   CREATE TABLE myTable() WITH (DATASOURCE=\"myTable\", TYPE=\"sql\", KIND=\"lookup\")
   ```
3. Join table with a stream
   ```sql
   SELECT * FROM demoStream INNER JOIN myTable on demoStream.id = myTable.id
   ```

New API

Discussion: Do we provide option to use scan for table if the source supports lookup?

1. In table definition, support a property `KIND` to specify if it is permanent or not.
2. Add lookup source interface.
   ```go
   // Lookup receive lookup values to construct the query and return query results
    Lookup(ctx StreamContext, lookupValues []interface{}) ([]interface{}, error)
   ```

Validations to add:

- Only `LookupSource` instance can be used as a permanent table.
- Check multiple joins.

Unlike temporary table, permanent table will keep running until it is dropped implicitly. Temporary table will only run if the rule is running.

Flow

Discussion: Lookup cache TTL setting?

1. When creating a rule which refers to a permanent table, parse the rule logic to generate the lookup query template. In this example, in SQL sink, the query template is `SELECT * FROM table WHERE id=?`
2. When running, driven by the stream events. Each event has a lookup value. In this example, the lookup value is `id` field of the stream. Firstly, query lookup cache by this value. The lookup cache is a map of the db lookup return result indexed by the id. If hit and not pass TTL, just return the result. Lookup cache is implemented in the table runtime.
3. If not hit, query the DB by SQL template with the new event value and return. This is implemented in the source side.
4. Save the new return value to cache.

Stream to permanent table

Some source type itself do not support lookup, but it is still valuable to have a `temporary` snapshot of data for joining as we have already provided. The current approach has two limitations:

- The table is append-only.
- The table cannot be shared by multiple rules.

Propose to use **memory permanent table** as a conversion middleware for stream/table conversion. Once the memory sink supports updatable, these two limitations are addressed.

Discussion: Do we pursuit for direct permanent support for scan sources?

Plugin support

Just add the mechanism to identify `LookupSource`.

Query the table

Discussion: Make this as low priority?

We provide the API to query the table which converts into a call to the lookup method.

- SQL table: just emit the SQL to the physical DB.
- File/memory table: parse SQL and run the query.

For temporary table, there is no handle to access the table, it is hard to implement yet.

Updatable Sink

Provide a general mechanism to update the sink "table". Similar to lookup source, only a few sinks are "updatable" naturally. The sink must support insert, update and delete.

- Neuron ?
- REST: use verbs to represent update, delete
- Memory: if the sink is bound to an in-memory table
- File: may be inefficient
- SQL: convert to various SQL statement
- Influxdb
- TDEngine
- Redis
- Image: manipulated as a file

Usage

> *Flink implementation Each event has a RowType: INSERT, UPDATE, DELETE*

Add a change type as a verb to the sink result. It is also the switch to indicate if the sink is updatable.

```
{
   "actions": [
      {
         "sql": {
            "database": "mydb",
            "table": "mytable",
            "changeTypeCol": "verb"
         }
      }
   ]
}
```

In this example, the data send to sink should have a field named `verb` whose value is in ["INSERT","UPDATE","DELETE"]. An example data is as below:

```
{
   "verb": "UPDATE",
   "id": 5,
   "name": "aaa"
}
```

Discussion: Verb as a field or a meta? During the data transformation, the meta may not exist.