SPIP: Support Streaming SQL interface in Spark

Authors: Jacky Lee, Genmao Yu, Sujith Chacko

1. Background and Motivation

At present, KSQL (Kafka SQL), Flink SQL (which is actually based on Calcite), SQL Stream, Storm SQL all provide an easy-to-use yet powerful interactive SQL interface, with which users can easily develop a stream processing, without the need to write code in a programming language such as Java or Scala. In Spark, we can also provide a streaming SQL syntax based on Structured Streaming.

There are some benefits to supporting SQL interface for Structured Streaming:

- Users, who are unfamiliar with spark streaming, can easily use SQL interface to develop a stream processing especially when migrating static query to stream query.
- Unified management for stream table and static table. We can store stream table definition in unified metastore, like Hive metastore. Users can easily read/write data from/to stream table. Besides, it is very helpful to do static-steam combination query.

2. Goals

- Introduce an extended SQL interface for stream processing.
- Stream query management SQL interface.

3. Design

3.1 Table and Query

Before introducing design, we should clarify two concepts:

1. **Table**: In the design of DataSource V2, we extended the capacity of TABLE. It can define both static data source and stream data source. *PS:* in the following document, **we use `stream table` in design description, but not introduce a new table concept.**

2. **Query**: The table supports both batch and streaming. When define a table in spark, we may be able to query it in batch or stream way. We should find a way to define how to read a Table, especially how to define a stream query on table.

In this document, we try to extend current SQL interface to support expressing stream query. Before we do a stream query, we should be able to define a stream table with table DDL. And then we can do a query on stream table, like select or insert, i.e. stream query related table DML. As we can do multiple queries on the same stream table, and all these query jobs **SHOULD** be long-running, so we **MUST** be able to manage these jobs, including “get status”, “restart job” (failover) and so on, i.e. *Stream Query Management* (let’s put it this way). We will introduce these concepts in detail in following sections.

3.2 TABLE DDL PROPOSQL

In this section, we will introduce how to create stream table and start a stream query. We introduced a new keyword “**STREAM**” which is the entity of stream query. This “STREAM” entity contains properties, like “batch interval”, “output mode” and “checkpoint path”, which can only be set on application/job level.

3.2.1 CREATE TABLE

This DDL is used to define a stream table. We can create a stream table like same way spark supports data source table creation. Besides, there is no obvious difference between static table definition and stream table definition.

Proposed DDL	**Equivalent in DataFrame**
create table kafka_int_test using kafka options (subscribe="test_topic", kafka.bootstrap.servers="localhost:9092");	*SessionCatalog.createTable(tableDefinition)*

3.2.2 CREATE SCAN

As mentioned above, `Table` in spark is just a definition of data source. For some specific data source, like Kafka, we can do both batch and stream query on it. Besides, there are some runtime configurations should be set when submit query, like ‘maxOffsetsPerTrigger’ in kafka data source. So there SHOULD be a concept to express these settings. In this section, we introduce the `SCAN` syntax. We can use `SCAN` to identify which type of query, and set some optional configurations.

Proposed DDL	**Equivalent in DataFrame**
create scan kafka_stream_scan using stream on kafka_in_test options("maxOffsetsPerTrigger" ="10000");	*spark.readStream.format("kafka")* *.option("options in table definition", "option value")* *.option("*maxOffsetsPerTrigger *","10000")* *.load()*

Proposed DDL	**Equivalent in DataFrame**

create scan kafka_batch_scan using batch on kafka_in_test options("startingoffsets" ="10000", "endingoffsets"="10020");	*spark.read.format("kafka")* *.option("options in table definition", "option value")* *.option("*startingoffsets *","10000")* *.option("*endingoffsets *","10020")* *.load()*

We register these table scan as a temporary view in spark..

3.2.3 CRAETE STREAM

This DDL is used to start a stream query job. User can specify particular stream related properties in ***OPTIONS***. When this is DDL executed, spark will start a structured streaming job in a separate thread to do the streaming ingestion, A stream job thread will be added/stored in a in memory thread safe container, this will enable spark engine to display all the stream jobs and to stop any stream job with specific DDL's is been executed by the user.

Proposed DDL	**Equivalent in DataFrame**
create stream job_1 options("outputMode" ="Append", "checkpointLocation"="/path/to/checkpoint", "trigger"="ProcessTime", "interval"="1 seconds") insert into kafka_out_test select * from kafka_stream_scan;	*df.writeStream.format("kafka")* *.option("kafka.bootstrap.servers","localhost:9092")* *.option("topic","kafka_out_topic")* *.option("checkpointLocation", "/path/to/checkpoint")* *.outputMode("Append")* *.trigger(Trigger.ProcessingTime("1 seconds"))* *.start()*

This query will write result to "kafka_out_test" table. `job_1` is the name of STREAM as well as streaming query. Here we use the `kafka_stream_scan`, which read kafka data in streaming way.

3.2.4 Window support

Spark Structured Streaming has introduced a built-in *window* function to express windowed aggregations. This *window* function accepts two duration arguments, "windowDuration" and "slideDuration". The "windowDuration" means the width of window, and "slideDuration" means the sliding interval of the window. In this section, we will introduce three types of window as alternatives based on existing streaming system, like Flink and Beam.

Group Window Function	**Description**
TUMBLING(time_column, interval)	(Fixed windows) Tumbling windows are a series of fixed-sized, non-overlapping and contiguous time intervals.

HOPPING(time_column, interval, interval)	(Sliding windows) Hopping windows model scheduled overlapping windows.
SESSION(time_column, interval)	(**unsupported**) Session-based windowing, and see details in https://issues.apache.org/jira/browse/SPARK-22565

Proposed DDL	**Equivalent in DataFrame**
create stream job_1 options("outputMode" ="Append", "checkpointLocation"="/path/to/checkpoint", "trigger"="ProcessTime", "interval"="1 seconds") insert into kafka_out_test select avg(inv_quantity_on_hand) qoh from kafka_stream_scan group by tumbling (inv_data_time, interval 1 minute);	*df.groupBy(* *window($"inv_data_time", "1 minutes", "1 minutes")* *.agg(type.avg(_*.inv_quantity_on_hand*))* *.writeStream.format("kafka")* *.option("kafka.bootstrap.servers","localhost:9092")* *.option("topic","kafka_out_topic")* *.option("checkpointLocation", "/path/to/checkpoint")* *.outputMode("Append")* *.trigger(Trigger.ProcessingTime("1 seconds"))* *.start()*

Proposed DDL	**Equivalent in DataFrame**
create stream job_1 options("outputMode" ="Append", "checkpointLocation"="/path/to/checkpoint", "trigger"="ProcessTime", "interval"="1 seconds") insert into kafka_out_test select avg(inv_quantity_on_hand) qoh from kafka_stream_scan group by hopping (inv_data_time, interval 1 minute, interval 30 second);	*df.groupBy(* *window($"inv_data_time", "1 minutes", "30 seconds")* *.agg(type.avg(_*.inv_quantity_on_hand*))* *.writeStream.format("kafka")* *.option("kafka.bootstrap.servers","localhost:9092")* *.option("topic","kafka_out_topic")* *.option("checkpointLocation", "/path/to/checkpoint")* *.outputMode("Append")* *.trigger(Trigger.ProcessingTime("1 seconds"))* *.start()*

PS: We can discuss whether keep using *window* function or introduce new *keyword.*

An optional alternative is the design in https://arxiv.org/abs/1905.12133v1. Add (as a starting point) built-in table-valued functions Tumble and Hop which take a relation and event time column descriptor as input and return a relation with additional

event-time interval columns as output, and establish a convention for the eventtime interval column names. GROUP BY is truly a grouping of rows according to a column's value. In Calcite, Beam, and Flink, GROUP BY HOP(...) violates relational semantics by causing multiple input rows. A more uniform notation for all window functions. The near-trivial Tumble has the same general form as the input-expanding Hop, and using a table-valued functions allows adding a wide variety of more complex functionality (such as calendar windows or sessionization) with a similar look-and-feel.

3.2.5 Watermark support

As discussed, `watermark` should be a per-query setting. It is inapposite to set it in stream table definition. Instead, we can set watermark in table scan definition.

Configuraion	**Description**
watermark.column	*the name of the column that contains the event time of the row.*
watermark.delayThreshold	*the minimum delay to wait to data to arrive late, relative to the latest record that has been processed in the form of an interval (e.g. "1 minute" or "5 hours").*

Proposed DDL	**Equivalent in DataFrame**
create scan kafka_stream_scan on kafka_in_test using stream options("watermark.column" = "data_time", "watermark.delayThreshold" = "2 minutes" "maxOffsetsPerTrigger" = "10000");	*df.withWatermark("data_time", "2 minutes")*

3.2.6 Aggregation support

Proposed DDL	**Equivalent in DataFrame**

create stream job_1 options("outputMode"="Append", "checkpointLocation"="/path/to/checkpoint", "trigger"="ProcessTime", "interval"="1 seconds") insert into kafka_out_test select device_type, avg(signal) avg_signal from kafka_stream_scan group by device_type;	*df.groupBy($"*device_type *")* *.agg(type.avg(_.signal))* *.writeStream.format("kafka")* *.option("kafka.bootstrap.servers","localhost:9092")* *.option("topic","kafka_out_topic")* *.option("checkpointLocation", "/path/to/checkpoint")* *.outputMode("Append")* *.trigger(Trigger.ProcessingTime("1 seconds"))* *.start()*

3.2.7 Join support

Structured Streaming supports joining a streaming Dataset/DataFrame with a static Dataset/DataFrame as well as another streaming Dataset/DataFrame.

- Stream-stream joins

Proposed DDL	**Equivalent in DataFrame**
create stream job_2 options ("checkpointLocation" ="/path/to/checkpoint", “outputMode” = “complete”) insert into kafka_out_test select t1.value from (select cast(value as string), timestamp as time1 from kafka_stream_scan_1) as t1 inner join (select cast(value as string), timestamp as time2 from kafka_stream_scan_2) as t2 on time1>=time2 and time1 <= time2 + interval 10 seconds where t1.value == t2.value	*val df1 = spark.readStream...load()* *val df2 = spark.readStream...load()* *val df = df1.join(df2,expr("""* *time1 >= time2 AND* *time1 <= time2 + interval 10 seconds"""))* *df.writeStream.format("kafka")* *.option("kafka.bootstrap.servers","localhost:9092")* *.option("topic","kafka_out_topic")* *.option("checkpointLocation", "/path/to/checkpoint")* *.outputMode("*Complete*")* *.start()*

- Stream-static joins

Proposed DDL	**Equivalent in DataFrame**

create stream job_2 *options (* *"checkpointLocation" ="/path/to/checkpoint",* “outputMode” = “complete”*)* *insert into kafka_out_test* *select t1.value from* *(select cast(value as string) from kafka_stream_scan) as t1* *inner join static_table* *on t1.value = static_table.value*	*val streamDF = spark.readStream...load()* *val staticDF = spark.read....load()* *val df = streamDF.join(staticDF)* *df.writeStream.format("kafka")* *.option("kafka.bootstrap.servers","localhost:9092")* *.option("topic","kafka_out_topic")* *.option("checkpointLocation", "/path/to/checkpoint")* *.outputMode("Complete")* *.start()*

3.3 Stream Query Management

As mentioned above, we may do multiple queries on stream tables, and all these stream queries **SHOULD** be long running. When we do SQL query in Spark SQL CLI in interactive way, the stream SQL query **MUST NOT** print any log into console, as it will block subsequent SQL query. So, we can choose to submit these queries to SQL server, like Thrift Server or Livy. Then, there should be a *Stream Query Management* to manage these stream queries, providing:

1. **LIST** stream query
2. **SHOW** stream query status
3. **DESC** a stream query
4. **STOP** a stream query
5. **(RE) START** a stream query

Proposed DDL	**Equivalent in DataFrame**
LIST STREAM	*None*

Proposed DDL	**Equivalent in DataFrame**
SHOW STREAM jobName	*val query = df.writeStream.format(“kafka”).start* query.id() query.runId() query.status() query.recentProgress()

Proposed DDL	**Equivalent in DataFrame**
DESC STREAM jobName	None

Proposed DDL	**Equivalent in DataFrame**

STOP STREAM jobName	*val query = df.writeStream.format("kafka").start* *query.stop()*

Proposed DDL	**Equivalent in DataFrame**
START STREAM jobName	*query.start()*

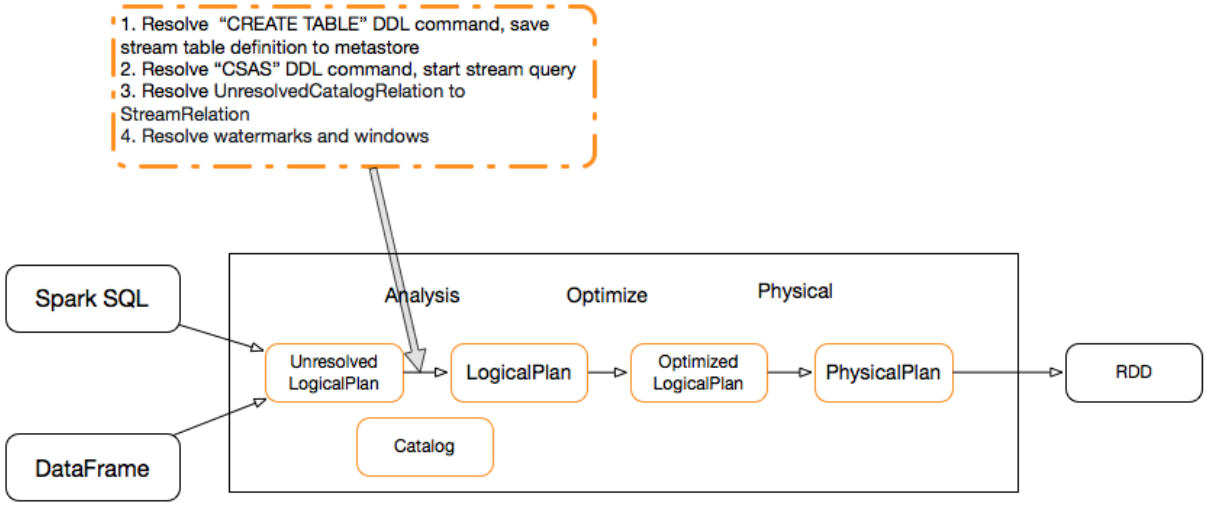
3.4 Unified management for stream and static table

Materialized the streaming table definition in some external catalog, like *HiveMetastore* or others. Then the stream table can be reused in other session without creating them again.

4. Optional Implementation Sketch

Main works will happen at logical plan analysis phase. In this phase we will achieve:

1. Resolve "CREATE TABLE" DDL command to create stream table. By parsing *USING* keyword, we can analyze the kind of stream data source. Save the stream table definition into metastore.
2. Resolve "CSAS" (Create Stream As Select) DDL command to start a stream query. We need to resolve *UnresolvedCatalogRelation* to either *StreamRelation* or Static Relation. In this phase, we will create an either *DataStreamReader* to read streaming data (unbound reading from stream data source) or *DataFrameReader* to read non-streaming data (range reading from stream data source).
3. Resolve watermarks and windows. We need to transfer the *Filter* logical plan to *EventTimeWatermark* logical plan. We will also need to do some validation check about watermark, i.e. *proximity declaration principle* and *one-on-one principle*.



5. Rejected alternatives

In the Streaming SQL design process, we have gone through several types of design:

1. (KEEP discussing) Use flag to identify the type of table (static or stream) and query (batch query or stream query). In the design of DataSource V2, the Table should only contain the information about data source, there is no need to identify a table is static or stream. Besides, there is no need to add flag for query, because we can achieve this by checking the query properties. For example, we may be doing a stream query if given an unbound offset of Kafka. Similarly, we may be doing a batch query if given a range offset of Kafka.

2. Directly add streaming related syntax, when the semantic analysis can be converted into an actual plan. The disadvantage is the system not only changes to much, but also become less friendly to users.

3. Only modify the semantic layer. The problem is that we can't determine if the user uses offline or real-time queries, because the offline queries for Kafka source are supported in Spark.

4. Modify the syntax layer, identify Streaming SQL, the Table is defined as a special *UnresolvedRelation*, the semantic layer directly to its analysis. The problem is that it is impossible to determine whether Table should be changed to *StreamingRelation*, especially when a query like Stream join Batch is executed.

5. Identify Streaming Query with the 'stream' keyword. It supports pure-SQL very efficiently, but does not support Table API well. Users cannot implement complex Streaming through Table/SQL API + DataSet API.

6. Related Works

6.1 Apache Beam

Apache Beam is an open source, unified model for defining both batch and streaming data-parallel processing pipelines.

- **External Table**：Beam SQL's *CREATE EXTERNAL TABLE* statement registers a virtual table that maps to an external storage system. For some storage systems, CREATE EXTERNAL TABLE does not create a physical table until a write occurs. After the physical table exists, you can access the table with the SELECT, JOIN, and INSERT INTO statements.
- **SqlTransform**: SQL query is translated to a *PTransform*, an encapsulated segment of a Beam pipeline. You can freely mix SQL *PTransforms* and other *PTransforms* in your pipeline. *SqlTransform.query(queryString)* method is the only API to create a *PTransform* from a string representation of the SQL query.
- **Window and Watermark**：
  - Use Beam's windowing semantics in two ways: (a) you can configure windowing on your input PCollections before passing them to a BeamSql transform. (b) you can use

windowing extensions in your windowing query, which will override the windowing of your input PCollections
    - Triggers allow processing of late data by triggering after the event time watermark passes the end of the window. Triggering can only be used by setting it on your input PCollections; there are **no SQL extensions** for specifying triggering.
- **Join limit:** only support INNER, LEFT OUTER, RIGHT OUTER.
- **Behaviour in shell**: specify the **LIMIT x** clause at the end of the SELECT statement to limit the output to x number of records. Otherwise, the pipeline can potentially run forever if one of the tables represents an unbounded source.

related docs:
- https://beam.apache.org/documentation/dsls/sql/windowing-and-triggering/
- https://beam.apache.org/documentation/dsls/sql/create-external-table/

6.2 Apache Flink
Current Flink SQL API support only DML (e.g. SELECT and INSERT statements), and a new Flink Table DDL design is under discussing, link. IIUC, the job specific parameters, like `checkpoint`, should be set in job env. There is not a concept in Flink table DDL to define a stream job.

7. FAQ

7.1 Multiple stream tables support?
Yes, we support one query on multiple stream tables.

7.2 Multiple streaming query support?
(KEEP discussing) Is it necessary to support multi streaming? Support multi streaming means it will be difficult to check the log if one of these Streaming has errors. It also causes different streams to compete for resources, making the computing environment more complex.

7.3 Interactive programing support?
Yes, we support running streaming query interactively. As mentioned above, we run stream query in a “detached” way. All stream queries are running in threads, and it **SHOULD** not print any log into console to disturb interactive programming. We can use *Stream Query Management* DDL to manage our stream queries.