Relational Queries on Data Streams in Apache Flink

Problem scope

SQL is a popular query language to describe data processing and computation logic. The rich semantics from SQL can meet the needs of most use cases. The Apache Flink community has started an effort to add two relational APIs, SQL and the LINQ-style Table API, to Apache Flink. Since Apache Flink supports both batch and stream processing and because there are many use cases where both kinds of processing are necessary, we aim for a unified SQL layer for batch and streaming data sources. While relational processing of batch data sources is well defined and understood, there is no widely accepted definition of the semantics for relational processing of streaming data. This document aims to define the semantics of Flink's SQL and Table API on data streams.

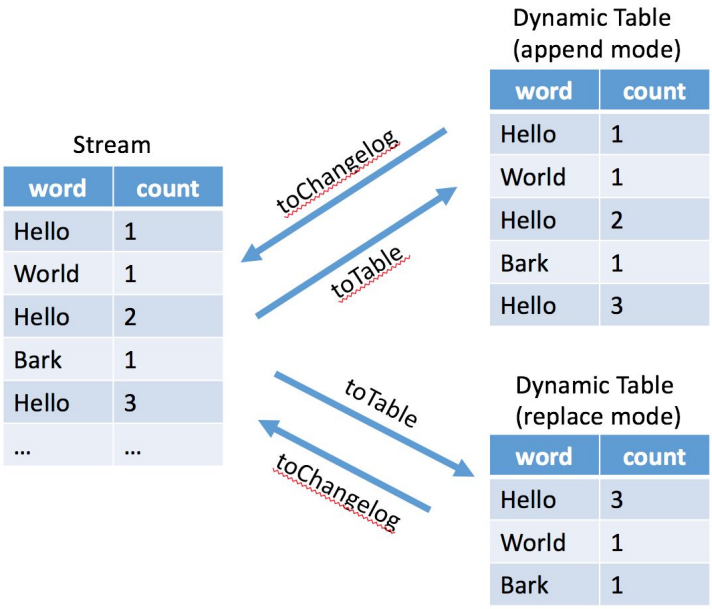
Unifying queries on batch and streaming sources

Relational database systems feature Materialized View which can for example be defined as follows

```
CREATE MATERIALIZED VIEW Result AS
      SELECT *, UDF(a, b, c)
      FROM Item JOIN Seller
      ON Item.uid=Seller.id
      WHERE Item.catid IN (.....)
```

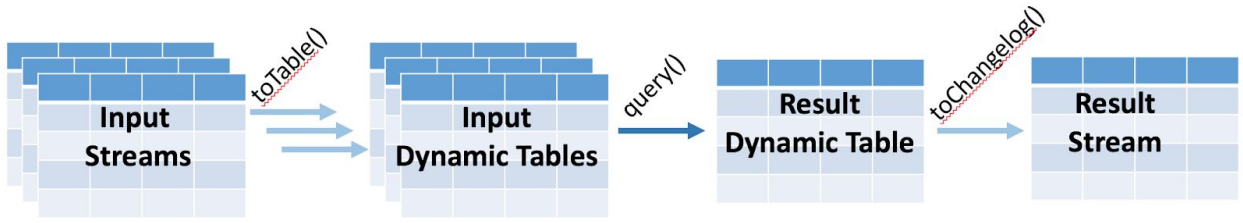
When a user creates a materialized view (see above for an example), the database system runs the query to initially populate the view. Later, when the input table get modified, it's the responsibility of the database system to maintain the consistency between the materialized view and its input tables. If we view both the updates to the input tables and the corresponding update to the result table as streams, the view maintenance can be viewed as a stream processing application. The interesting part is that the logic for this stream processing application is compiled from the view definition query. This offers an interesting way to unify batch and stream processing using relational query interfaces such as SQL and the Table API.

The basic concept is the stream and table duality:



A stream can be converted into a table with identical schema. There are two modes to convert a stream into a table: 1) *Append Mode*: All records of a stream are appended to the table. Once a record has been added to the table it is never updated or deleted. 2) *Replace Mode*: This mode requires to define a table with a primary key attribute. Records of the stream are either inserted into the table if no record with the same key attribute exists, otherwise the existing record is replaced by the new record (see example figure above). In both cases, the derived tables are *dynamic tables*, i.e., tables whose content changes over time.
The operation inverse to converting a stream into a table is to derive a stream from a table by emitting the changes applied to the table as a changelog stream. The changelog step does also have two modes: 1) *Retraction Mode*: In retraction mode, update changes on dynamic table result in two emitted changes, a delete change for the updated record and an insert change for the updating record. 2) *Update mode*: Update mode requires the definition of a key attribute on the dynamic table. All change events emitted to the changelog have a key and a value attribute. The key attribute is always the key attribute of the dynamic table. For insert and update change the value is the new record. For delete changes the value is empty to indicate that the key has been deleted.
With these two operations (toTable() and toChangelog()), we can see that a stream and its derived dynamic tables (both in append and replace mode) contain the same information. This is the stream and table duality.

We leverage the table-stream duality to formalize stream processing logic using batch queries over the dynamic tables. First, for each input stream, we conceptually construct a dynamic table. Then we express the desired result dynamic table from these input dynamic tables. Finally, we derive the changelog from the result history table.



The Query step is relatively straightforward, it can be the standard relational query. Queries can be either defined as SQL statements or as Table API queries. Initially, both API will be equally expressive. Over time, the Table API might evolve and support operations that cannot be expressed with SQL.

Since a stream can be unbounded (infinite), it's possible that the dynamic table is unbounded (infinite). When we query infinite tables, we will need to make sure that the query is well defined. In some cases, this requires us to specify some constraints in the query itself. We will cover more details later.

Use cases for relational queries on streams

- ETL / Data Import: Ingest streaming data, transform (normalize, aggregate) it, and write it to another system (Files, Kafka, DBMS). These are continuous queries that usually emit final results to a log-style system where emitted data cannot be updated (results cannot be refined).
- Online View Maintenance: Ingest streaming data and compute aggregates for online systems (dashboards, recommenders, etc.) or data analysis tools (Tableau). Results are written into k-v stores (Cassandra, queryable Flink state), indexes (Elasticsearch), or DBMSs (MySQL, PostgreSQL, ...). These queries are continuous queries that can emit early results which can be updated and refined.
- Ad-hoc continuous stream queries: Ad-hoc queries against a data stream to analyze and explore streaming data in a real-time fashion. Query results might be displayed in a notebook (e.g., Apache Zeppelin). Nice features might be fixing a stream to a certain position from which multiple queries are executed (kind of like starting new queries from an empty save point).

Semantics of relational queries on streams

The following sections formalize the previous discussion and define details.

Table and query semantics

1. A dynamic table *T* is a table that changes over time.

2. Conceptually, a dynamic table T can be retrieved for a point in time *t* as *T(t)*.
3. A change on a dynamic table can be represented as *(insert, <newRow>)* for insert changes, *(delete, <oldRow>)* for delete changes, and as a *(delete, <oldRow>)*, *(insert, <newRow>)* pair for update changes.
4. Changes on a dynamic table with primary key attribute *k* can also be represented as *(k, <newRow>)* for insert and update changes and as *(k, _)* for delete changes.
5. The set of all changes to a dynamic table T between two different points in time t and u is defined as: *c(T, t, u)*.
6. A dynamic table T can be queried with a regular relational query *q*. *q* is a dynamic query and returns a dynamic table Q which can be retrieved for a specific point in time *Q(t) = q(T(t))*. If a query q references two (or more) dynamic tables T and S, all tables are synced to the same time, *Q(t) = q(T(t), S(t))*.

Dynamic tables, streams, and duality

1. The changelog *CL(T)* of a dynamic table *T* is the stream of the changes between periodic retrievals of *T* in an interval of x: *CL(T) = c(T, 0, t) ++ c(T, t, t+x) ++ c(T, t+x, t+2x) ++ ...*
2. A stream consists of rows that have a schema and a timestamp attribute *rowtime* which is (quasi-)monotonically increasing. *rowtime* is a read-only system attribute.
3. A dynamic append table *A(t)* is a dynamic table to which rows are only appended. Existing rows are never removed or modified. A dynamic append table *A(t)* grows infinitely for $t \rightarrow \infty$.
4. A stream can be converted into a dynamic append table *A(t)*. At each time *t*, the table is defined as all stream rows with *rowtime* <= *t*. The conversion fulfills the stream-table duality: S -> A, CL(A) -> S', S == S'
5. A stream can be converted into a dynamic update table *U(t)*. At each time *t*, the table contains the row with *max(rowtime)* <= *t* for a given key. The conversion fulfills the stream-table duality: S -> U, CL(U) -> S', S == S'
6. Dynamic append table and dynamic update table are both regular dynamic tables.
7. A dynamic query q is evaluated by computing its changelog CL(Q), i.e., by periodically evaluating q for increasing points in time.

Supported queries

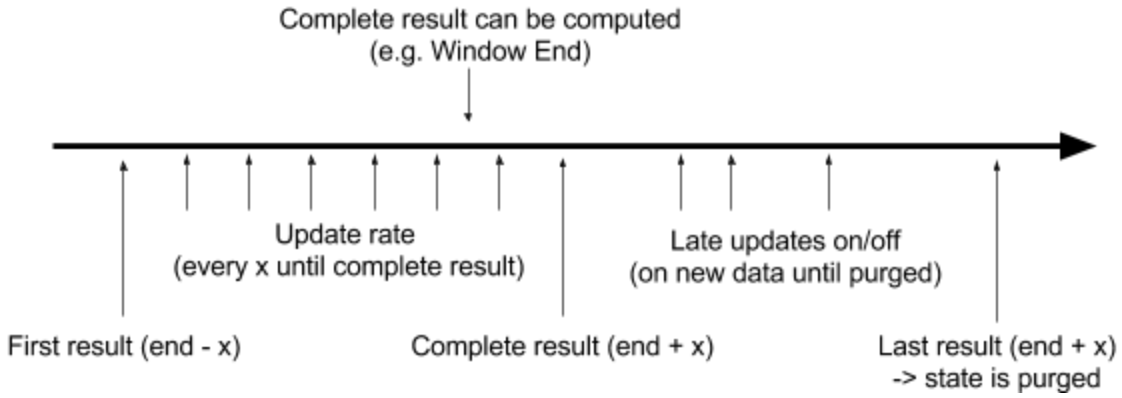
1. Conceptually, a dynamic query q is evaluated by periodically computing q on its dynamic input tables for increasing timestamps, i.e., Q(t) = q(T(t)), for t = 1, 2, …, n.
2. In practice, Flink translates a dynamic query into a continuous streaming application that evaluates the query for an increasing logical time (usually defined by watermarks, but a processing time mode might be supported as well). Hence, Flink does not support to evaluate queries (and input tables) for arbitrary points in time but only for the current logical time ('now').

3. Conceptually, the results of q at different points in time can be converted into a changelog stream CL(Q) = *c(Q, t, t+x) ++ c(Q, t+x, t+2x) ++ c(Q, t+2x, t+3x) ++ ...*
4. Because it is not feasible to compute a query for every point of time from scratch, queries must be incrementally computable from the changes on their base tables in order to be supported by Flink. There are three types of queries that can be incrementally computed.
5. Queries that continuously update previous results, i.e., queries that produce tables with insert, update, and delete changes. These queries can be computed as Q(t+1) = q'(Q(t), c(T, t, t+1)) where Q(t) is the previous result of q, c(T, t, t+1) are the changes of T from t to t+1, and q' is an incremental version of query q. An example for this type of queries is a query that computes a running aggregate where the aggregation function is commutative and associative (i.e., combinable). The query can be evaluated by incrementally refining the aggregation result, e.g., adding to or subtracting from a sum.
6. Queries that produce infinite append-only dynamic tables and which can be completely computed from the tail of an (infinite) dynamic input table, i.e., Q(t+1) = q''(c(T, t-x, t+1)) ∪ Q(t), where Q(t) is the result of q at time t, q'' is an incremental version of the original query q that does not require the result of q at time t, and c(T, t-x, t+1) is a tail of input table *T* of length *x+1*. *x* depends on the query semantics, for example a window aggregation query of the last hour needs to keep at least the input of one hour as state. Other supported query types are:
    - SELECT WHERE queries which operate on each row individually
    - GROUP BY clauses on the rowtime attribute (i.e, time based window aggregates)
    - OVER windows (row-windows) with ORDER BY rowtime
    - ORDER BY rowtime
7. Queries that require point accesses to a complete input table can only be supported if the input tables are sufficiently small, i.e., Q(t+1) = q'''(T(t), c(T, t, t+1)) where T(t) is the previous input, c(T, t, t+1) are the changes on the input, and q''' is an incremental version of q which accesses T(t) only pointwise to compute Q(t+1). For example the result of a query that joins two size-bound dynamic tables can be computed using a symmetric hash join strategy which requires each input table to be stored in an updateable hashtable. In this case, the input table is pointwise accessed to compute new, updated, or retracted join result records.
8. Meta information tables can help the optimizer to reason about the size of intermediate state or result of queries. For example the information that the domain of an attribute is fixed and not growing / evolving can help to conclude that the intermediate state of a query which groups on that attribute is fixed.

Result computation and refinement timing

Certain relational operators such as windowed aggregates have to wait for data to arrive before a final result can be computed. For example, a window that closes at 10:30 needs to wait at least until 10:30 before it can compute a final result. Due to out-of-order events and late arriving data, it might be necessary to wait even longer in order to avoid the computation of an

incomplete result. On the other hand, certain applications would like to have access to early results which are continuously refined. Hence, there are different requirements when a result should be computed, refined, or made final.
Result refinement does not affect the semantics of a query and should therefore not be part of the query. Instead it is a property of the query evaluation.
The following figure depicts different configuration parameters to control early result and refinement computation of query results.



- The “First Result Offset” defines the time at which the first early result is computed. The time is relative to the time at which a full result can be computed the first time, i.e., when the logical time (processing or event) reaches a certain time. A first result offset of -10 minutes will produce the first result at 10:20 for a window which ends at 10:30. By default the first result offset is equal to 0, i.e, the first result of a window is computed at its end time.
- The “Complete Result Offset” defines the time at which the complete result is computed, i.e., the result where all relevant data is expected to have arrived). The time is relative to the time at which the full result can be computed the first time, i.e., a complete result offset of 5 minutes will produce the complete result of a window which ends at 10:30 at 10:35. The complete result offset parameter can mitigate the effect of late arriving data and is therefore only valid for event time operations. By default the complete result offset is equal to 0, i.e., the complete result of a window is computed at its end time.
- The “Update Rate” defines the interval (time or count) in which results are updated until the complete result is computed. For example, an update rate of 5 minutes for a 30 minute tumbling window that started at 10:00 with an early result at -15 minutes and a complete result at 2 minutes will produce result updates at 10:20, 10:25, and 10:30 (10:15 is the first, 10:32 the complete result). The update rate parameter does also define the rate at which results are updated which will never complete but continuously adapted such as running aggregates. The default update rate is count(1), which means that results are update for every new arriving record.
- The “Late Updates Switch” configures whether or not late updates are computed for each late arriving record after the complete result has been emitted until the state of the computation has been purged.

- The “Last Result Offset” defines the time at which the last result *can* be computed. This is the point in time at which the internal state is purged.

Emission of dynamic tables

Conceptually, a dynamic table is changing over time. As long as it is not emitted these changes are completely internal and not visible to a user. However, these changes materialize when a dynamic table is emitted. How often the changes of a dynamic table are emitted depends on the configuration of the result computation (see section above). For example setting the “Update rate” to 5 minutes for a dynamic table that represents a running aggregate means that the updates of five minutes are batched and emitted. The result computation and refinement options can also be used to control whether and how many result refinements are computed. Since all changes only materialize when a dynamic table is emitted, we treat the result computation and refinement options as parameters for the emission of dynamic tables. In addition users can request a “result state flag” that indicates whether a record is an early, complete, or final result.

There are two ways to emit a dynamic table.

1. Emitting the changelog of the table as a DataStream.
2. Writing the table to an external table which is defined in the TableEnvironment.

Both options are discussed in the following.

Emitting a table as a changelog DataStream

A dynamic table can be converted into a DataStream by emitting all changes as a changelog stream. Changes can be emitted in two ways, as retraction or update changes.

1. Retraction Changelog DataStream: A dynamic table can be converted into a retraction changelog DataStream[(Boolean, Row)]. The boolean field is true for insertion changes and false for deletion changes. Update changes are modeled a deleting the old record and inserting the new row, i.e., an update change results in two (Boolean, Row) events. Emitting a table as a retraction changelog stream could look like this:

   ```
   // emitting a table
   table.toChangelog(outputOpts)

   // emission via the table environment
   tEnv.toChangelog(table, outputOpts)
   ```

2. Update Changelog DataStream: A dynamic table with primary key (or unique attributes) can be converted into an update changelog DataStream[(Row, Option[Row])]. The first Row attribute holds the key attributes of the inserted, updated, or deleted row. The

second Option attribute holds the inserted or updating row or is None in case of a deletion.
Emitting a table as an update changelog stream could look like this:

```
// emitting a table
table.toChangelog(outputOpts, 'keyAttr)

// emission via the table environment
tEnv.toChangelog(table, outputOpts, 'keyAttr)
```

Writing a table to an external table

A dynamic table can be written into an external table, i.e., a table that resides in an external storage system such as HDFS, Apache Kafka, Apache Cassandra, Apache HBase, or PostgreSQL. The table has to be registered as an output table in the TableEnvironment. The registration includes a name, a schema (including key attributes), and a TableSink object that is taking care of writing the table to the external system in the correct format (e.g., Parquet file in HDFS, Avro-encoded Kafka topic, or Cassandra table). Registering an external table could look as follows:

```
tEnv.registerOutputTable(
  "myTable",                    // table name
  tableSink,                    // TableSink object
  'uid.key, 'uname, 'lastvisit) // schema with (optional) key definition
```

A table can be written to an external table as follows:

```
// write table to an output table
table.writeToOutputTable("myTable", outputOpts)

// writing via the table environment
tEnv.writeToOutputTable(table, "myTable", outputOpts)
```

The optimizer will check that the schema of the dynamic and output tables match.
A SQL query can also be directly emitted as follows:

```
// write result of a query to an output table
table.sqlToOutputTable(
s"""
INSERT INTO myTable
SELECT *
FROM _
WHERE ...
""", outputOpts)

// write query result via table environment
```

```
tEnv.sqlToOutputTable(sql, outputOpts)
```

There are two types of output table: 1) append output table and 2) update output tables. The distinction is made based on whether the schema of the output table has a key attribute or not and based on the type of the TableSink object, e.g., a key attribute may only be defined for an UpdateTableSink which supports key updates. Both output table types are discussed in detail below:

1. External Append Output Table: All output tables defined without a key attribute are append output tables. Records written to an append output table cannot be updated or deleted. Therefore, only insert changes can be emitted to an append output table. Dynamic tables that have update and delete changes cannot be emitted. Consequently, append output tables are only supported for dynamic tables which are produced in an append only fashion. Moreover, result refinements must be disabled (only a single final result for each aggregate may be produced). These checks must be done before a query is started and emitted to an append output. Example of systems backing an append output table are:
    - Log-based systems such as Kafka and Kinesis
    - (Rolling) files in (distributed) filesystems (HDFS, S3)
3. Update Table Sink: All output tables defined with a key attribute are update output tables. A dynamic table with primary key (or unique attributes) can be written to an update output table. Records written to an update output table can be updated and deleted. Examples for systems that back update output tables are:
    - Key-value stores such as Cassandra, HBase, Elasticsearch
    - A compacted Kafka topic
    - Materialization into Flink's key-value state to make it accessible for queryable state. This requires a bounded size, i.e., table size may not depend on time and key space must be bounded.

Bounding memory requirements

Relational queries incrementally process continuously arriving data. As highlighted in the "Supported Queries" section, incremental processing of certain queries requires to keep some data (parts of the input or intermediate results) as state in memory. In order to ensure that queries do not fail at some point in time, it is important that the space requirements of a query are bounded, i.e., do not infinitely grow over time. There are two main reasons for growing space requirements: 1) growing intermediate state of computations which are not bounded by a time predicate such as increasing key spaces of running aggregates and 2) growing intermediate state of computations which are bounded by time but that need to incorporate late arriving data, such as window aggregates. While the second case is taken care of by defining a "Last Result Offset" as proposed in the "Result Computation" section, the first case needs to be detect by the optimizer. Queries whose computation requires an intermediate result which is not

bounded by a time predicate should be rejected. Instead the optimizer should give a hint how to fix the query and request an appropriate time predicate. For example the intermediate state of the following query on the a dynamic append table “pageviews”:

```
SELECT user, page, COUNT(page) AS pCnt
FROM pageviews
GROUP BY user, page
```

will very likely grow over time as the number of users and pages grow. The space requirements can be bounded by adding a time predicate as

```
SELECT user, page, COUNT(page) AS pCnt
FROM pageviews
WHERE rowtime BETWEEN now() - INTERVAL '1' HOUR AND now() // only last hour
GROUP BY user, page
```

Since not all attributes have growing domains (such as user and page), tables can be annotated with schema metadata such as constant domain size. Given this information, the optimizer can infer that intermediate state is not growing over time and accept queries without time predicates:

```
val sensorT: Table = sensors
  .toTable('id, 'loc, 'stime, 'temp)
  .attributeDomain('loc, Domain.constant) // domain of 'loc is not growing
env.registerTable("sensors", sensorT)

SELECT loc, AVG(temp) AS avgTemp
FROM sensors
GROUP BY loc
```

Examples

Simple group window aggregate query emitted to Cassandra

```
val tEnv: TableEnvironment = ???

// DataStream: sensorId, location, time, temp
val sensors: DataStream[(Long, String, Long, Double)]

// convert stream into append table
val sensorT: Table = sensors.toTable('id, 'loc, 'stime, 'temp) // table schema

// Cassandra table sink
//   Implements a updatable table sink interface to indicate that
//   emitted rows can be updated.
val casTableSink: StreamingTableSink = new CassandraTableSink(
  "avgTemp",      // table to write to
  props)          // Cassandra connection properties

// register update output table
tEnv.registerOutputTable(
  "avgRoomTemp", casTableSink, 'id.key, 'atime.key,  'temp) // key

// configure update mode
val outputOpts = OutputOptions()
  .firstResult(-15.minutes)    // first result 15 mins early
  .completeResult(+5.minutes)  // complete result 5 mins late
  .updateRate(3.minutes)       // result is updated every 3 mins
  .lateUpdates(true)           // late result updates enabled
  .lastResult(+15.minutes)     // last result 15 mins late -> state is purged

// compute hourly average temperature of each sensor
sensorT.sqlToOutputTable(
S"""
  INSERT INTO avgRoomTemp
  SELECT id, TUMBLE_START(stime, "1" HOUR INTERVAL) AS atime, AVG(temp)
  FROM _     // "_" references to the table on which sql() is called.
  WHERE loc LIKE 'room%'
  GROUP BY id, TUMBLE(stime, "1" HOUR INTERVAL)
""", outputOpts)
```

Simple row window aggregate query emitted to append-only file

```
val tEnv: TableEnvironment = ???

// DataStream: sensorId, location, time, temp
val sensors: DataStream[(Long, String, Long, Double)]

// register sensor stream as table
tEnv.registerDataStream("sensorT", sensors, 'id, 'loc, 'stime, 'temp)

// compute smoothed temperature of each sensor
val smoothedTemp: Table = tEnv.sql(
s"""
  SELECT id, loc, stime,
    AVG(temp) OVER (ORDER BY rowtime RANGE INTERVAL '5' MINUTE PRECEDING)
      AS smoothTemp
  FROM sensorT
""")

// Rolling file table sink
val fileTable: StreamingTableSink = new RollingFileTableSink(
  "/users/johndoe/avgTemp", // directory to write to
  "yyyy-MM-dd--HH")         // filename date-time pattern

// register append output table
tEnv.registerOutputTable(
  "extSmoothedTemp", fileTable, 'id, 'loc, 'stime, 'smoothTemp) // no key!

// configure update mode
val outputOpts = OutputOptions()
  .completeResult(+10.minutes)  // complete result 10 mins late
                                // early results and late updates disabled
                                //   by default.

// emit result to Cassandra sink
tEnv.writeToOutputTable(smoothedTemp, "extSmoothedTemp", outputOpts)
```

Query on keyed table emitted as changelog DataStream

```
val tEnv: TableEnvironment = ???

// DataStream: sensorId, location, time, temp
val sensors: DataStream[(Long, String, Long, Double)]

// convert stream into update table keyed on sensorId
val sensorT: Table = sensors
  .toKeyedTable('id.key, 'loc, 'stime, 'temp) // 'id is key
  .attributeDomain('loc, Domain.constant) // domain of 'loc is not growing

// compute smoothed temperature of each sensor
val avgLocTemp: Table = sensorT.sql(
s"""
  SELECT loc, AVG(temp) as avgTemp
  FROM _
  WHERE rowtime BETWEEN now() - INTERVAL '1' HOUR AND now() // only last hour
  GROUP BY loc
""")

//
val outputOpts = OutputOptions()
  .updateRate(5.minutes) // results are updated every 5 minutes

// emit table as update changelog stream.
//   A changelog is a DataStream[Tuple2[key, Option[row]].
//   Insert and update changes are have the Option set to the row value.
//   Delete changes have a None Option.
val avgLocTempChgs: DataStream[(Tuple1[String], Option[(String, Double)])] =
  avgLocTemp.toChangelogStream(outputOpts, 'loc.key)
```