Streaming SQL

Julian Hyde

9th XLDB Conference
SLAC, Menlo Park, 2016/05/25
@julianhyde

SQL
Query planning
Query federation
OLAP
Streaming
Hadoop

Apache member
VP Apache Calcite
PMC Apache Arrow, Drill, Kylin

Thanks:
- Milinda Pathirage & Yi Pan (Apache Samza)
- Haohui Mai (Apache Storm)
- Fabian Hueske & Stephan Ewen (Apache Flink)
Streaming data sources

Sources:
- Devices / sensors
- Web servers
- (Micro-)services
- Databases (CDC)
- Synthetic streams
- Logging / tracing

Transports:
- Kafka
- Nifi
How much is your data worth?

Recent data is more valuable
➢ ...if you act on it in time

Data moves from expensive memory to cheaper disk as it cools

Old + new data is more valuable still
➢ ...if we have a means to combine them
Why query streams?

Stream - Database Duality:

- “Your database is just a cache of my stream”
- “Your stream is just change-capture of my database”

“Data is the new oil”

- Treating events/messages as data allows you to extract and refine them

Declarative approach to streaming applications
Why SQL?

- API to your database
- Ask for **what you want**, system decides **how to get it**
- Query planner (optimizer) converts logical queries to physical plans
- Mathematically sound language (relational algebra)
- For all data, not just "flat" data in a database
- Opportunity for novel data organizations & algorithms
- Standard

https://www.flickr.com/photos/pere/523019984/ (CC BY-NC-SA 2.0)
Data workloads

- Batch
- Transaction processing
- Single-record lookup
- Search
- Interactive / OLAP
- Exploration / profiling
- Continuous execution generating alerts (CEP)
- Continuous load

A variety of workloads, requiring specialized engines, but to the user it’s all “just data”.
Building a streaming SQL standard via consensus

Please! No more “SQL-like” languages!

Key technologies are open source (many are Apache projects)

Calcite is providing leadership: developing example queries, TCK

(Optional) Use Calcite’s framework to build a streaming SQL parser/planner for your engine

Several projects are working with us: Samza, Storm, Flink. (Also non-streaming SQL in Cassandra, Drill, Druid, Elasticsearch, Flink, Hive, Kylin, Phoenix.)
Simple queries

### Traditional (non-streaming)

```
select * 
from Products 
where unitPrice < 20
```

- `Products` is a table
- Retrieves records from $-\infty$ to now

### Streaming

```
select stream * 
from Orders 
where units > 1000
```

- `Orders` is a stream
- Retrieves records from now to $+\infty$
- Query never terminates
Stream-table duality

- Yes, you can use a stream as a table
- And you can use a table as a stream
- Actually, Orders is both
- Use the stream keyword
- Where to actually find the data? That’s up to the system

```sql
select * 
from Orders
where units > 1000
```

```sql
select stream * 
from Orders
where units > 1000
```
Combining past and future

```sql
select stream *
from Orders as o
where units > (  
    select avg(units)
    from Orders as h
    where h.productId = o.productId
    and h.rowtime > o.rowtime - interval '1' year)
```

➢ Orders is used as both stream and table
➢ System determines where to find the records
➢ Query is invalid if records are not available
The replay principle:

*A streaming query produces the same result as the corresponding non-streaming query would if given the same data in a table.*

Output must not rely on implicit information (arrival order, arrival time, processing time, or watermarks/punctuations)

(Some triggering schemes allow records to be emitted early and re-stated if incorrect.)
Making progress

It’s not enough to get the right result. We need to give the right result at the right time.

Ways to make progress without compromising safety:

➢ Monotonic columns (e.g. rowtime) and expressions (e.g. floor (rowtime to hour))
➢ Punctuations (aka watermarks)
➢ Or a combination of both

```sql
select stream productId, count(*) as c
from Orders
group by productId;
```

ERROR: Streaming aggregation requires at least one monotonic expression in GROUP BY clause
Policies for emitting results

Monotonic column

Event time

10:00 10:15 10:30 11:00 11:15
Arrival time

1 2 3 5 4 7

Drop out-of-sequence records

Emit 10:00-11:00 window when first record after 11:00 arrives

Watermark

Event time

10:00 10:15 10:30 11:00 11:15
Arrival time

1 2 4 5 7 8

Emit 10:00-11:00 window when 11:00 watermark arrives

New watermark. Re-state 10:00-11:00 window
Aggregation and windows on streams

**GROUP BY** aggregates multiple rows into sub-totals

- In regular **GROUP BY** each row contributes to exactly one sub-total
- In multi-**GROUP BY** (e.g. HOP, GROUPING SETS) a row can contribute to more than one sub-total

**Window functions** (OVER) leave the number of rows unchanged, but compute extra expressions for each row (based on
GROUP BY

```sql
select stream productId, 
    floor(rowtime to hour) as rowtime, 
    sum(units) as u, 
    count(*) as c 
from Orders 
group by productId, 
    floor(rowtime to hour)
```
### Window types

<table>
<thead>
<tr>
<th>Window Type</th>
<th>Description</th>
<th>Diagram</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tumbling window</td>
<td>“Every T seconds, emit the total for T seconds”</td>
<td><img src="image1" alt="Diagram" /></td>
</tr>
<tr>
<td>Hopping window</td>
<td>“Every T seconds, emit the total for T² seconds”</td>
<td><img src="image2" alt="Diagram" /></td>
</tr>
<tr>
<td>Session window</td>
<td>“Emit groups of records that are separated by gaps of no more than T seconds”</td>
<td><img src="image3" alt="Diagram" /></td>
</tr>
<tr>
<td>Sliding window</td>
<td>“Every record, emit the total for the surrounding T seconds”</td>
<td><img src="image4" alt="Diagram" /></td>
</tr>
<tr>
<td></td>
<td>“Every record, emit the total for the surrounding R records”</td>
<td><img src="image5" alt="Diagram" /></td>
</tr>
</tbody>
</table>
Tumbling, hopping & session windows in SQL

**Tumbling window**

```sql
select stream ... from Orders
group by floor(rowtime to hour)
```

```sql
select stream ... from Orders
group by tumble(rowtime, interval '1' hour)
```

**Hopping window**

```sql
select stream ... from Orders
group by hop(rowtime, interval '1' hour, interval '2' hour)
```

**Session window**

```sql
select stream ... from Orders
group by session(rowtime, interval '1' hour)
```
select stream
  sum(units) over w (partition by productId) as units1hp,
  sum(units) over w as units1h,
  rowtime, productId, units
from Orders
window w as (order by rowtime range interval '1' hour preceding)
The “pie chart” problem

➢ Task: Write a web page summarizing orders over the last hour
➢ Problem: The Orders stream only contains the current few records
➢ Solution: Materialize short-term history

```
select productId, count(*)
from Orders
where rowtime > current_timestamp - interval '1' hour
group by productId
```
Join stream to a table

Inputs are the `Orders` stream and the `Products` table, output is a stream.

Acts as a “lookup”.

Execute by caching the table in a hashmap (if table is not too large) and stream order will be preserved.

```
select stream *
from Orders as o
join Products as p
  on o.productId = p.productId
```
Join stream to a *changing* table

Execution is more difficult if the Products table is being changed while the query executes.

To do things properly (e.g. to get the same results when we re-play the data), we’d need temporal database semantics.

(Sometimes doing things properly is too expensive.)

```sql
select stream * 
from Orders as o
join Products as p 
on o.productId = p.productId
and o.rowtime
    between p.startEffectiveDate
and p.endEffectiveDate
```
Join stream to a stream

We can join streams if the join condition forces them into “lock step”, within a window (in this case, 1 hour).

Which stream to put input a hash table? It depends on relative rates, outer joins, and how we’d like the output sorted.

```
select stream *
from Orders as o
join Shipments as s
on o.productID = p.productID
and s.rowtime
    between o.rowtime
    and o.rowtime + interval '1' hour
```
Planning queries

```
select p.productName, count(*) as c
from splunk.splunk as s
    join mysql.products as p
    on s.productId = p.productId
where s.action = 'purchase'
group by p.productName
order by c desc
```
select p.productName, count(*) as c
from splunk.splunk as s
join mysql.products as p
  on s.productId = p.productId
where s.action = 'purchase'
group by p.productName
order by c desc
Apache Calcite

Apache top-level project since October, 2015

Query planning framework
➢ Relational algebra, rewrite rules
➢ Cost model & statistics
➢ Federation via adapters
➢ Extensible

Packaging
➢ Library
➢ Optional SQL parser, JDBC server
➢ Community-authored rules, adapters

<table>
<thead>
<tr>
<th>Embedded</th>
<th>Adapters</th>
<th>Streaming</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache Drill</td>
<td>Apache</td>
<td>Apache Flink*</td>
</tr>
<tr>
<td>Apache Hive</td>
<td>Cassandra</td>
<td>Apache Samza</td>
</tr>
<tr>
<td>Apache Kylin</td>
<td>Apache Spark</td>
<td>Apache Storm</td>
</tr>
<tr>
<td>Apache Phoenix*</td>
<td>CSV</td>
<td></td>
</tr>
<tr>
<td>Cascading</td>
<td>Druid*</td>
<td></td>
</tr>
<tr>
<td>Lingual</td>
<td>Elasticsearch*</td>
<td></td>
</tr>
</tbody>
</table>

* Under development

Embedded

Adapters

Streaming

Apache Drill
Apache Hive
Apache Kylin
Apache Phoenix*
Cascading
Lingual

Apache
Cassandra
Apache Spark
CSV
Druid*
Elasticsearch*
In-memory
JDBC
JSON
MongoDB
Splunk
Web tables
Architecture

Conventional database

Calcite
Relational algebra (plus streaming)

Core operators:
- Scan
- Filter
- Project
- Join
- Sort
- Aggregate
- Union
- Values

Streaming operators:
- Delta (converts relation to stream)
- Chi (converts stream to relation)

In SQL, the STREAM keyword signifies Delta.
Streaming algebra

- Filter
- Route
- Partition
- Round-robin
- Queue
- Aggregate
- Merge
- Store
- Replay
- Sort
- Lookup
Optimizing streaming queries

The usual relational transformations still apply: push filters and projects towards sources, eliminate empty inputs, etc.

The transformations for delta are mostly simple:

- \( \Delta(\text{Filter}(r, \text{predicate})) \rightarrow \text{Filter}(\Delta(r), \text{predicate}) \)
- \( \Delta(\text{Project}(r, e_0, ...)) \rightarrow \text{Project}(\Delta(r), e_0, ...) \)
- \( \Delta(\text{Union}(r_0, r_1), \text{ALL}) \rightarrow \text{Union}(\Delta(r_0), \Delta(r_1)) \)

But not always:

- \( \Delta(\text{Join}(r_0, r_1, \text{predicate})) \rightarrow \text{Union}(\text{Join}(r_0, \Delta(r_1)), \text{Join}(\Delta(r_0), r_1)) \)
- \( \Delta(\text{Scan}(a\text{Table})) \rightarrow \text{Empty} \)
Sorting a streaming query is valid as long as the system can make progress.

Need a monotonic or watermark-enabled expression in the ORDER BY clause.

```
select stream productId,
       floor(rowtime to hour) as rowtime,
       sum(units) as u,
       count(*) as c
from Orders
group by productId,
       floor(rowtime to hour)
order by rowtime, c desc
```
As in a typical database, we rewrite $x \ union \ y$ to:

```
select distinct * from (x union all y)
```

We can implement $x \ union \ all \ y$ by simply combining the inputs in arrival order but output is no longer monotonic. Monotonicity is too useful to squander!

To preserve monotonicity, we merge on the sort key (e.g. `rowtime`).
DML

➢ View & standing INSERT give same results
➢ Useful for chained transforms
➢ But internals are different

```
insert into LargeOrders
select stream * from Orders
where units > 1000
```

```
create view LargeOrders as
select stream * from Orders
where units > 1000
```

Use DML to maintain a “window” (materialized stream history).

```
upsert into OrdersSummary
select stream productId, count(*) over lastHour as c
from Orders
window lastHour as (partition by productId
order by rowtime
range interval '1' hour preceding)
```
Summary: Streaming SQL features

Standard SQL over streams and relations

Streaming queries on relations, and relational queries on streams

Joins between stream-stream and stream-relation

Queries are valid if the system can get the data, with a reasonable latency

- Monotonic columns and punctuation are ways to achieve this

Views, materialized views and standing queries
Summary: The benefits of streaming SQL

Relational algebra covers needs of data-in-flight and data-at-rest applications.

High-level language lets the system optimize quality of service (QoS) and data location.

Give DB tools and traditional users to access streaming data; give message-oriented tools access to historic data.

Combine real-time and historic data, and produce actionable results.

Discussion continues at Apache Calcite, with contributions from Samza, Flink, Storm and others. Please join in!
Thank you!

@julianhyde

@ApacheCalcite

http://calcite.apache.org

http://calcite.apache.org/docs/stream.html

References