Streaming SQL

Julian Hyde

FlinkForward
Berlin
2016/09/12

@julianhyde

SQL
Query planning
Query federation
OLAP
Streaming
Hadoop

HORTONWORKS

Apache Calcite

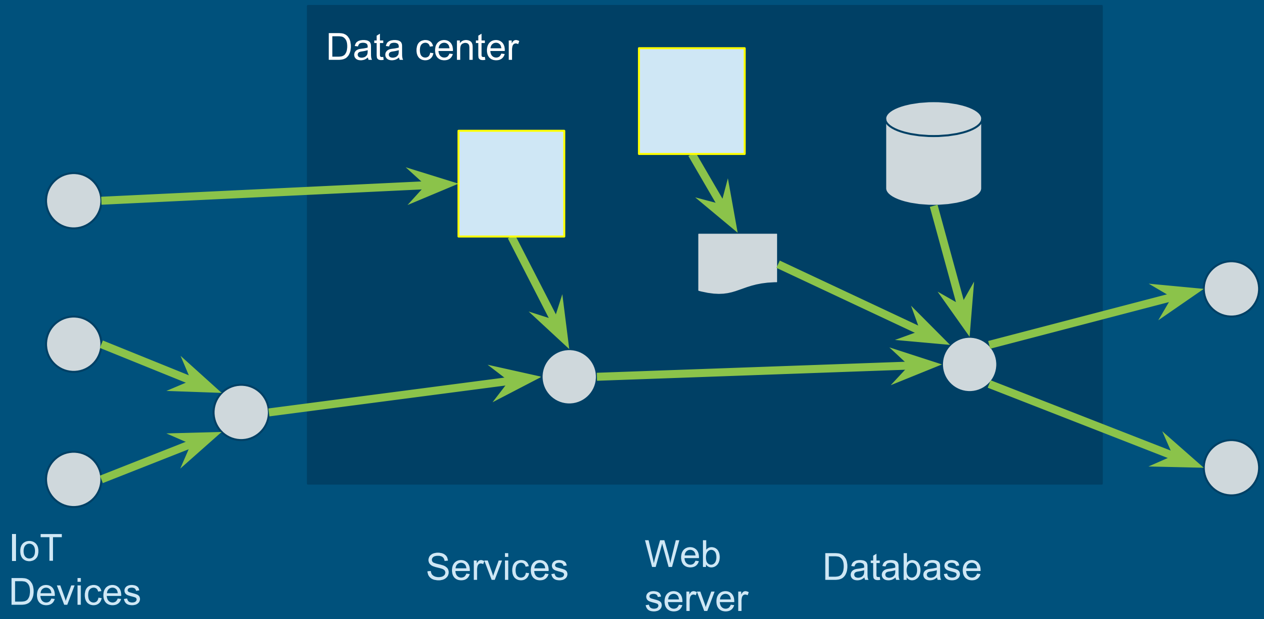
mondrian
pentaho analysis services

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?

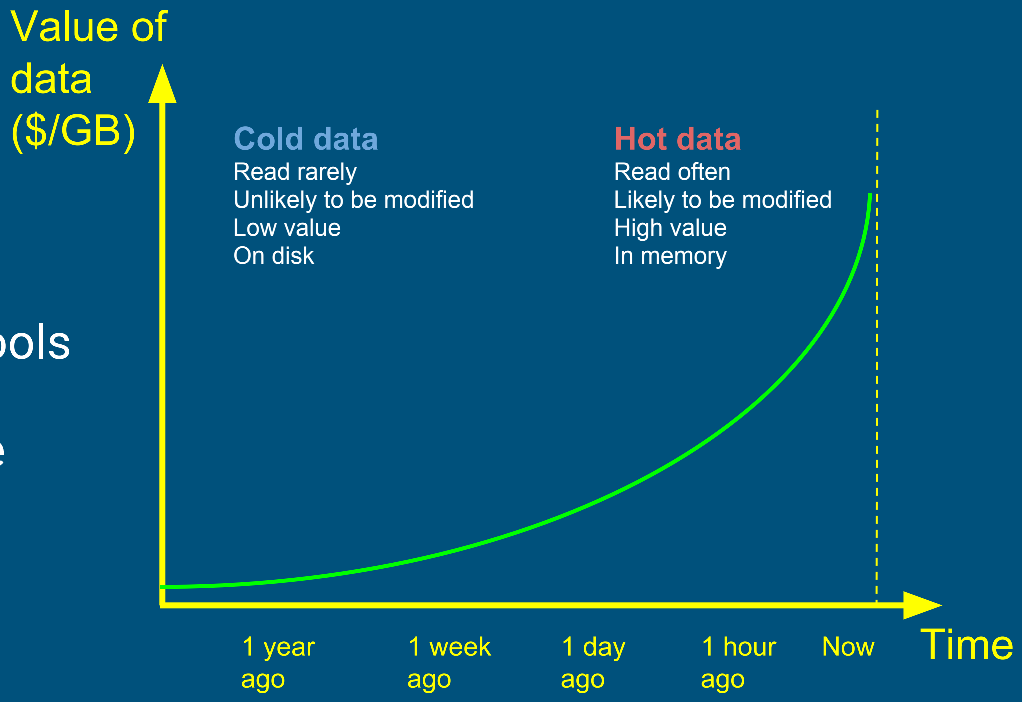
Recent data is more valuable

➢ ...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:

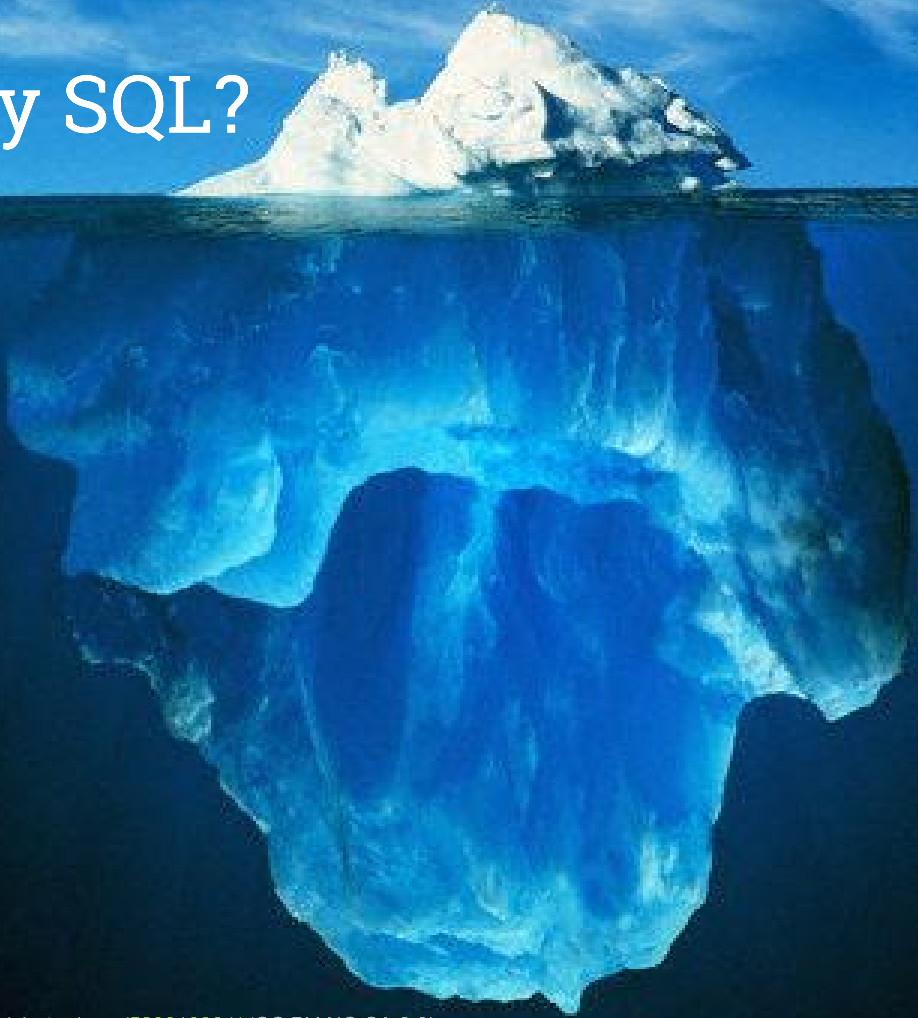
- “Your database is just a cache of my stream”
- “Your stream is just change-capture of my database”

“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

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: Flink, Samza, Storm, Apex. (Also non-streaming SQL in Cassandra, Drill, Druid, Elasticsearch, Flink, Hive, Kylin, Phoenix.)

Simple queries

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

- Traditional (non-streaming)
- `Products` is a table
- Retrieves records from -∞ to now

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

- Streaming
- `Orders` is a stream
- Retrieves records from now to +∞
- Query never terminates

Stream-table duality

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

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

- ➢ 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

Combining past and future

```
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

Semantics of streaming queries

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.

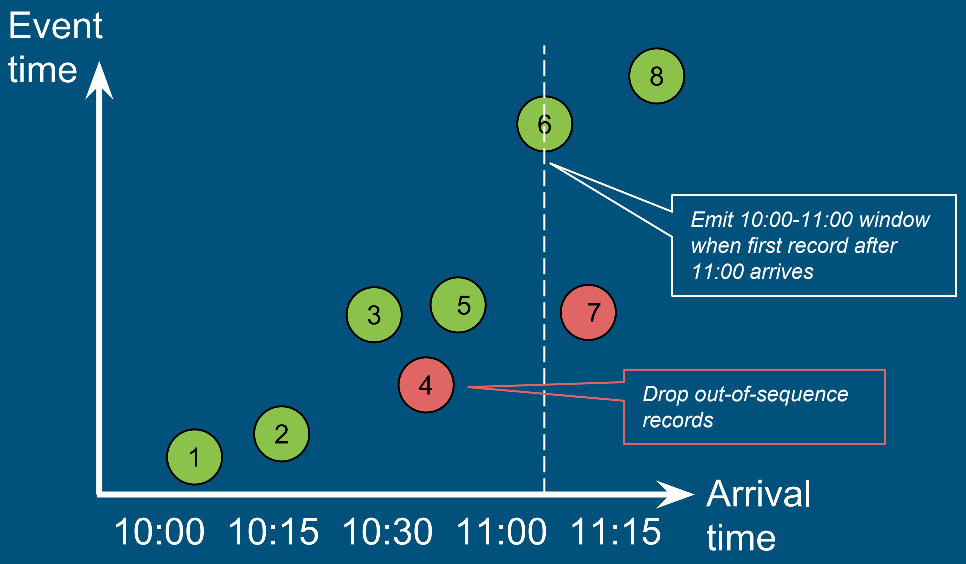
Ways to make progress without compromising safety:

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

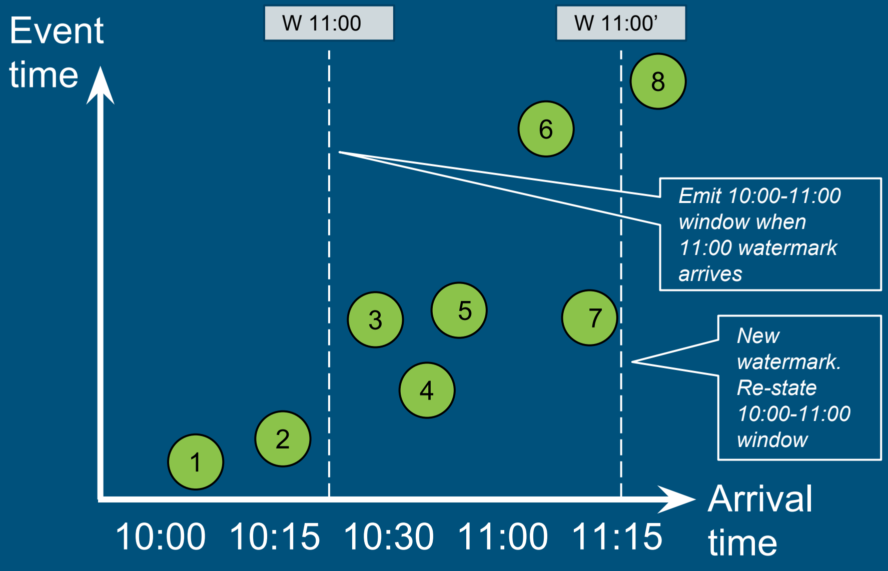
```
select stream productId,
  count(*) as c
from Orders
group by productId;

ERROR: Streaming aggregation
requires at least one
monotonic expression in
GROUP BY clause
```

Policies for emitting results



Monotonic column



Watermark

Controlling when data is emitted

Early emission is the defining characteristic of a streaming query.

The emit clause is a SQL extension inspired by Apache Beam's "trigger" notion. (Still experimental… and evolving.)

A relational (non-streaming) query is just a query with the most conservative possible emission strategy.

```
select stream productId,
  count(*) as c
from Orders
group by productId,
  floor(rowtime to hour)
emit at watermark,
  early interval '2' minute,
  late limit 1;
```

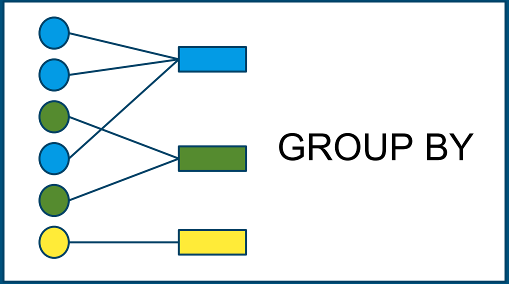
```
select *
from Orders
emit when complete;
```

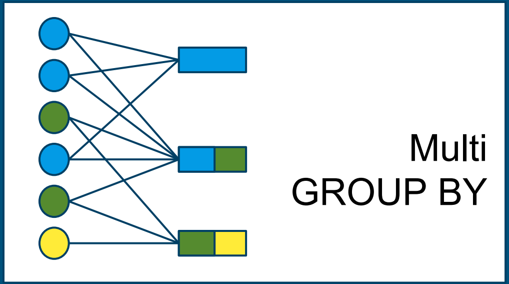
Aggregation and windows on streams

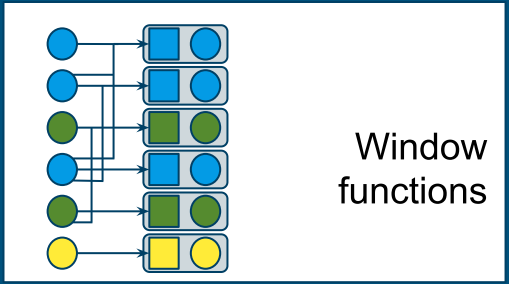
GROUP BY aggregates multiple rows into sub-totals

- ➢ 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 neighboring rows)







GROUP BY

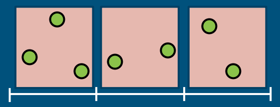
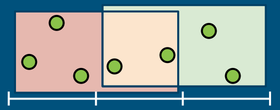
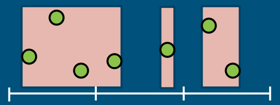
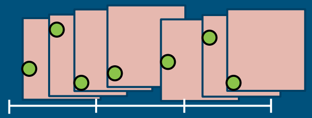
```
select stream productId,
  floor(rowtime to hour) as rowtime,
  sum(units) as u,
  count(*) as c
from Orders
group by productId,
  floor(rowtime to hour)
```

rowtime	productId	units
09:12	100	5
09:25	130	10
09:59	100	3
10:00	100	19
11:05	130	20

rowtime	productId	u	c
09:00	100	8	2
09:00	130	10	1
10:00	100	19	1

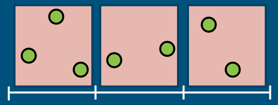
not emitted yet; waiting
for a row >= 12:00

Window types

Tumbling window	“Every T seconds, emit the total for T seconds”	
Hopping window	“Every T seconds, emit the total for T2 seconds”	
Session window	“Emit groups of records that are separated by gaps of no more than T seconds”	
Sliding window	“Every record, emit the total for the surrounding T seconds” “Every record, emit the total for the surrounding R records”	

Tumbling, hopping & session windows in SQL

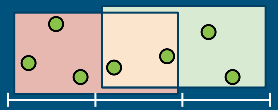
Tumbling window



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

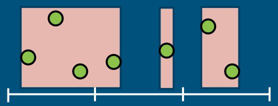
```
select stream … from Orders
group by tumble(rowtime, interval ‘1’ hour)
```

Hopping window



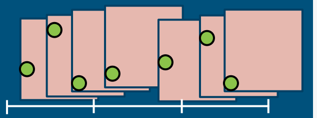
```
select stream … from Orders
group by hop(rowtime, interval ‘1’ hour,
    interval ‘2’ hour)
```

Session window



```
select stream … from Orders
group by session(rowtime, interval ‘1’ hour)
```

Sliding windows in SQL



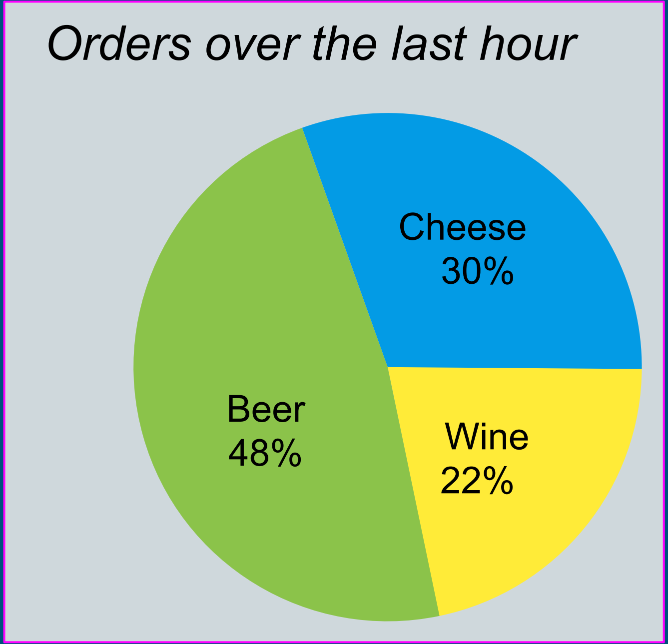
```
select stream
  sum(units) over w (partition by productId) as units1hp,
  sum(units) over w as units1h,
  rowtime, productId, units
from Orders
window w as (order by rowtime range interval '1' hour preceding)
```

rowtime	productId	units
09:12	100	5
09:25	130	10
09:59	100	3
10:17	100	10

units1hp	units1h	rowtime	productId	units
5	5	09:12	100	5
10	15	09:25	130	10
8	18	09:59	100	3
23	13	10:17	100	10

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 hash-map (if table is not too large) and stream order will be preserved.

What if `Products` table is being modified while query executes?

```
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.)

```
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

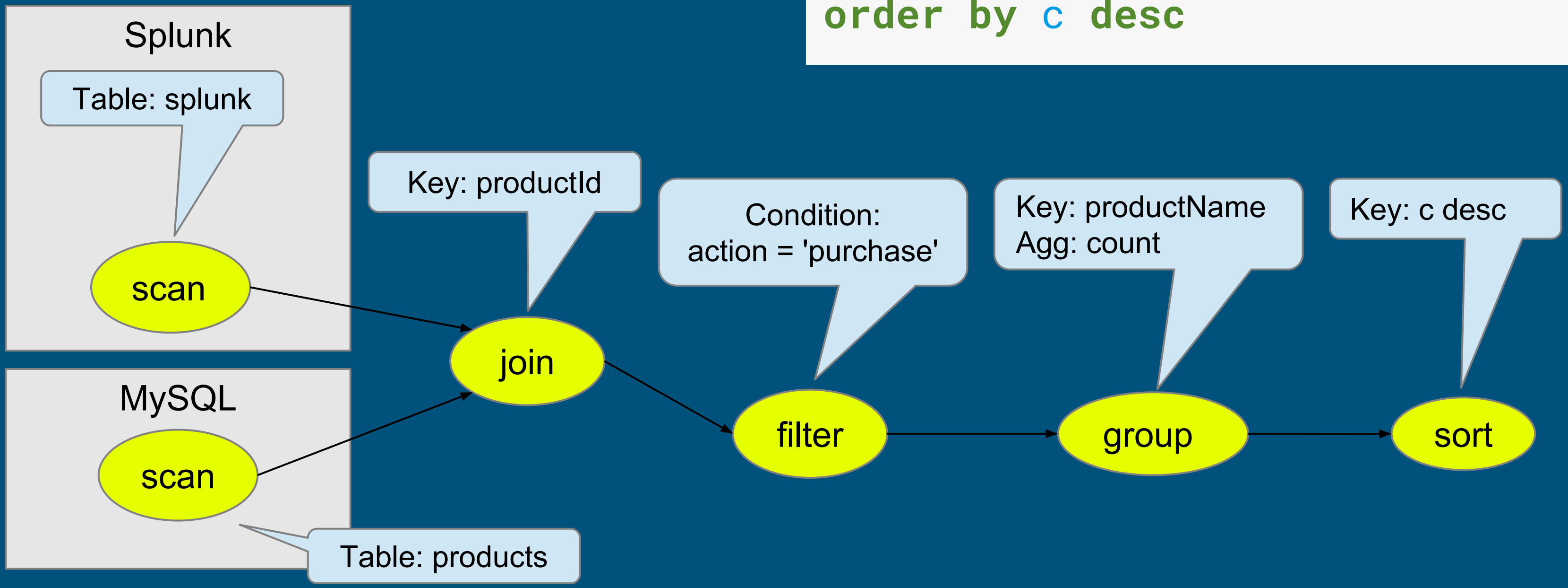
We can join streams if the join condition forces them into “lock step”, within a window (in this case, 1 hour).

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
```

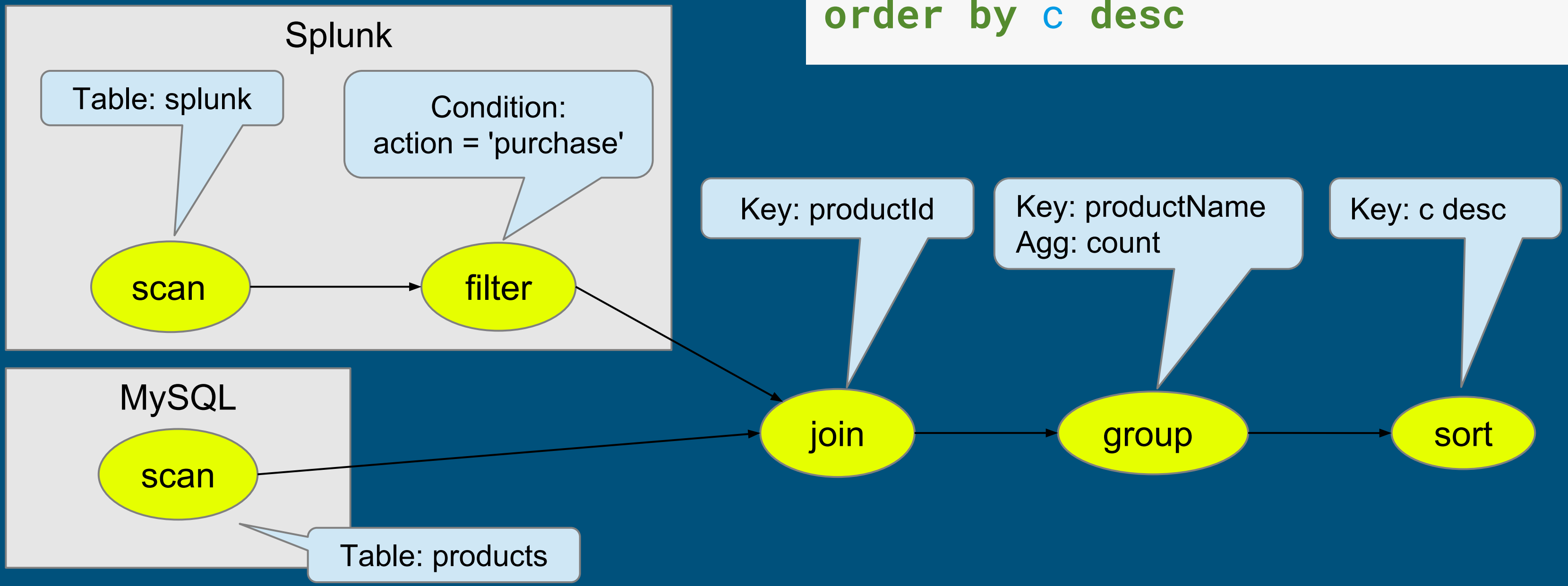
Planning queries

```sql
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
```



Optimized query

```
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

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	Apache Flink* Apache Samza Apache Storm

** Under development*

Join the community!

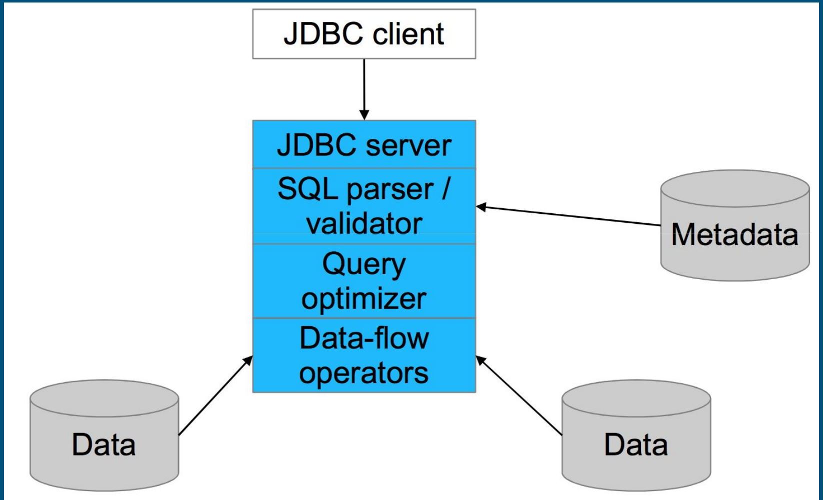
Calcite and Flink are projects of the Apache Software Foundation

The Apache Way: meritocracy, openness, consensus, community

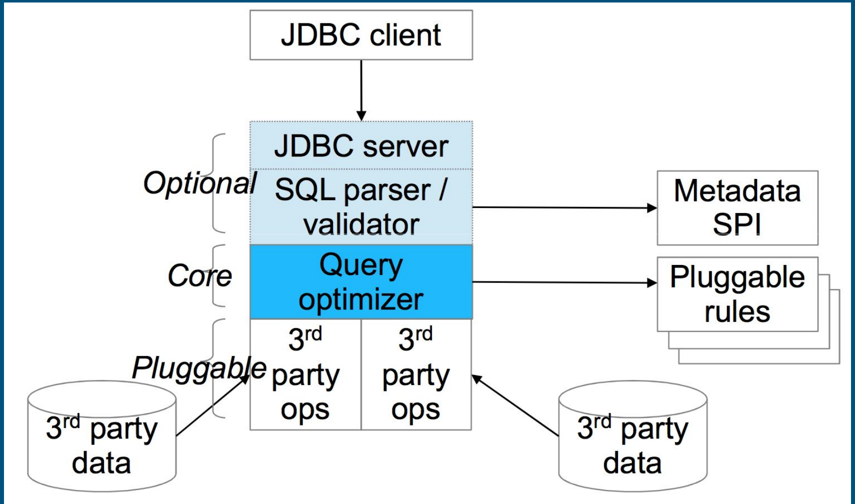
We welcome new contributors!

Architecture

Conventional database



Calcite



Relational algebra (plus streaming)

Core operators:

- ➢ Scan
- ➢ Filter
- ➢ Project
- ➢ Join
- ➢ Sort
- ➢ Aggregate
- ➢ Union
- ➢ Values

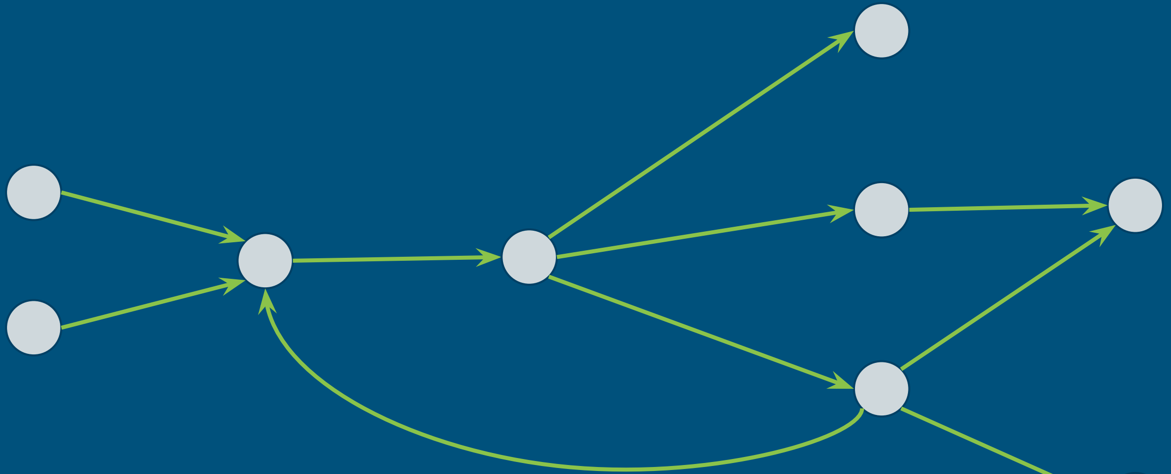
Streaming operators:

- ➢ 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(Filter(r, predicate)) → Filter(Delta(r), predicate)
- ➢ Delta(Project(r, e0, ...)) → Project(Delta(r), e0, …)
- ➢ Delta(Union(r0, r1), ALL) → Union(Delta(r0), Delta(r1))

But not always:

- ➢ Delta(Join(r0, r1, predicate)) → Union(Join(r0, Delta(r1)), Join(Delta(r0), r1)
- ➢ Delta(Scan(aTable)) → Empty

Other operations

Other relational operations make sense on streams (usually only if there is an implicit time bound).

Examples:

- `order by` - E.g. Each hour emit the top 10 selling products
- `union` - E.g. Merge streams of orders and shipments
- `insert`, `update`, `delete` - E.g. Continuously insert into an external table
- `exists`, `in` sub-queries - E.g. Show me shipments of products for which there has been no order in the last hour
- `view` - Expanded when query is parsed; zero runtime cost

Summary

Features of streaming SQL:

- Standard SQL over streams and relations
- Relational queries on streams, and vice versa
- Materialized views and standing queries

Benefits:

- Brings streaming data to DB tools and traditional users
- Brings historic data to message-oriented applications
- Lets the system optimize quality of service (QoS) and data location

Thank you!

@julianhyde

@ApacheCalcite

http://calcite.apache.org

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

References

- Hyde, Julian. "Data in flight." Communications of the ACM 53.1 (2010): 48-52. [pdf]
- Akidau, Tyler, et al. "The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing." Proceedings of the VLDB Endowment 8.12 (2015): 1792-1803. [pdf]
- Arasu, Arvind, Shivnath Babu, and Jennifer Widom. "The CQL continuous query language: semantic foundations and query execution." The VLDB Journal—The International Journal on Very Large Data Bases 15.2 (2006): 121-142. [pdf]