Streaming SQL

Julian Hyde

Apache Samza meetup
Mountain View, CA
2016/2/17

@julianhyde

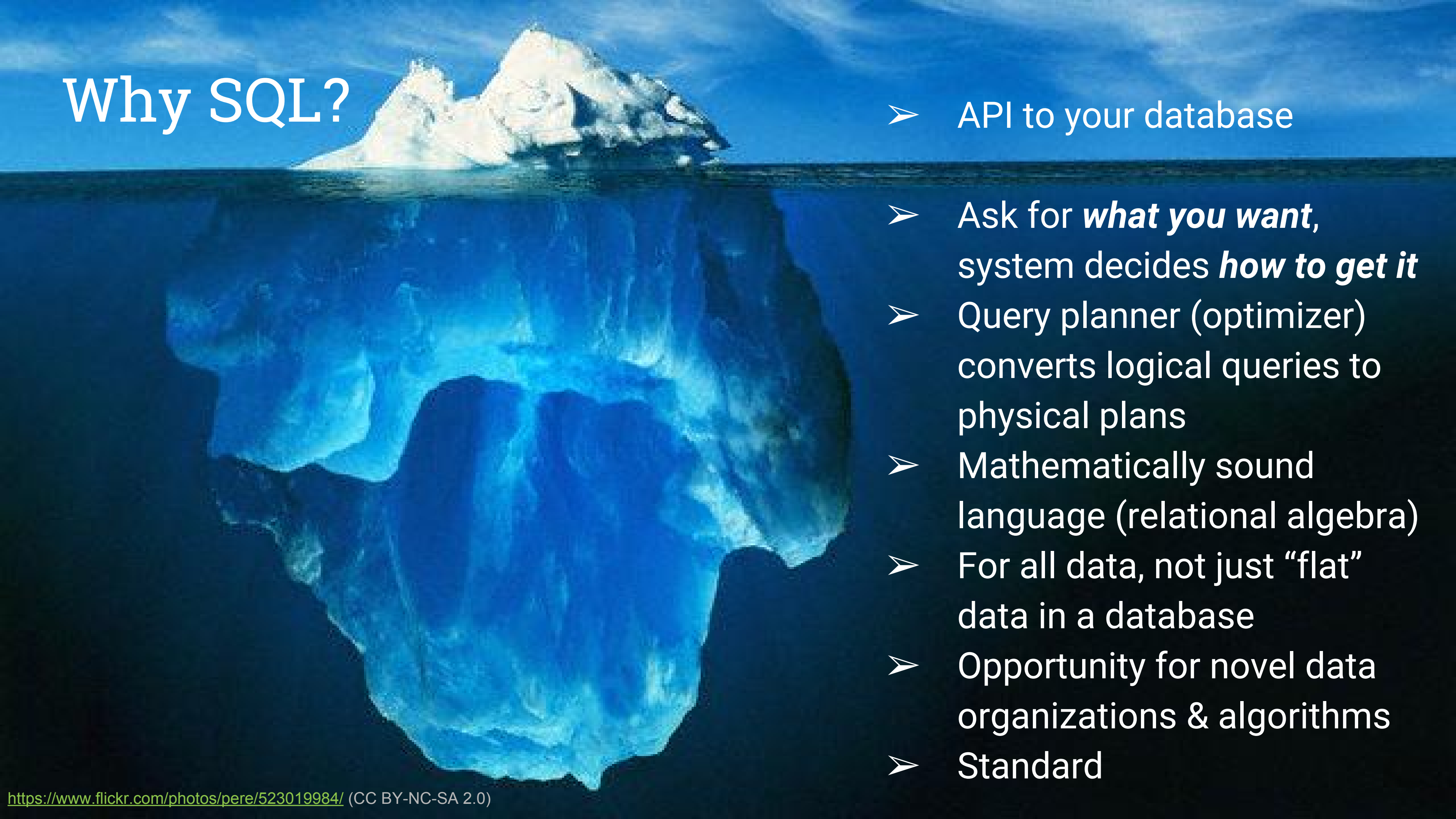
SQL
Query planning
Query federation
OLAP
Streaming
Hadoop

Hortonworks®

Apache Calcite

mondrian™ pentaho analysis services™

Thanks to Milinda Pathirage for his work on samza-sql and the design of streaming SQL



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

How much is your data worth?

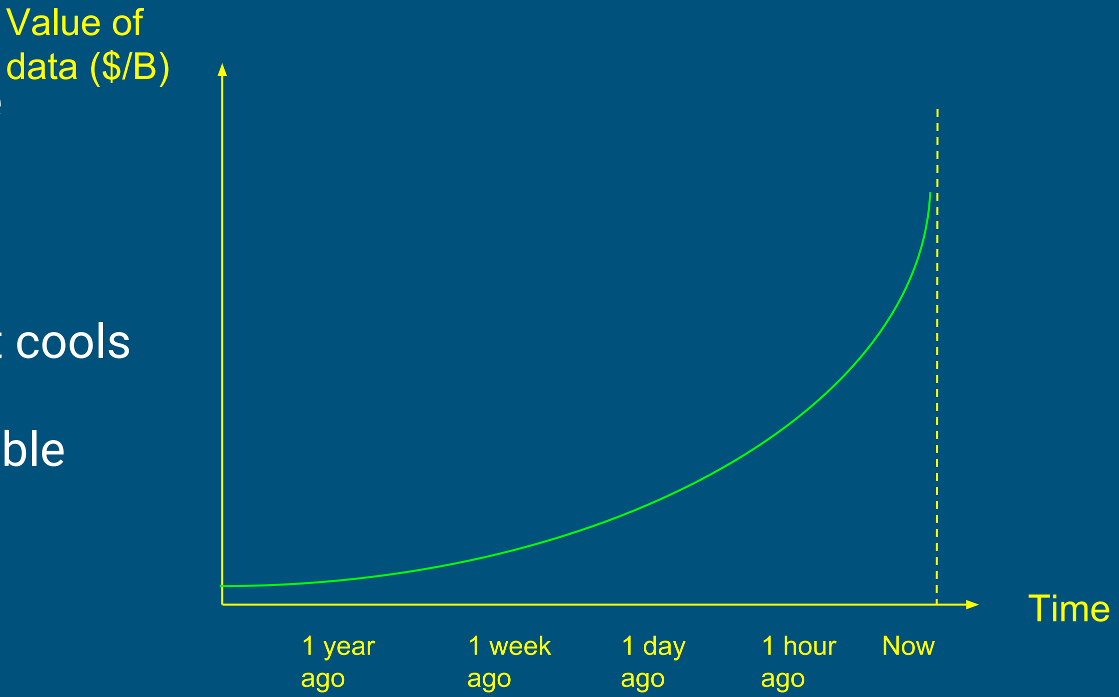
Recent data is more valuable

- ...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



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

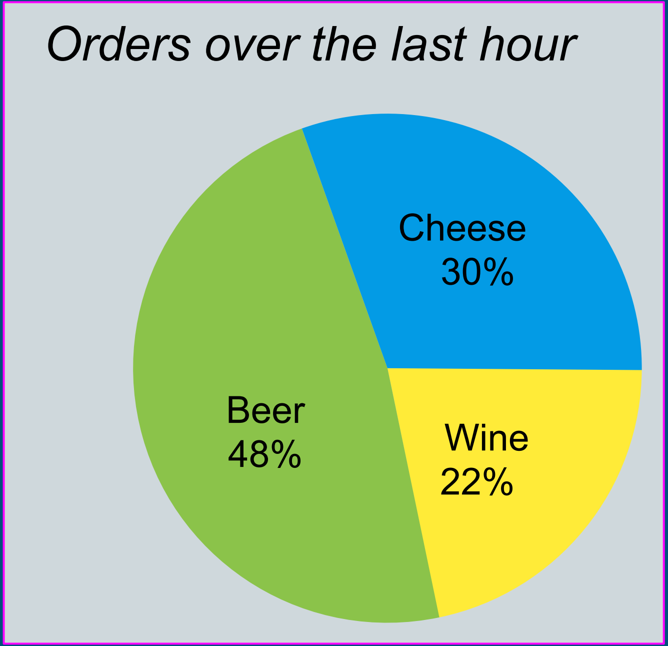
Combining past and future

```
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

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



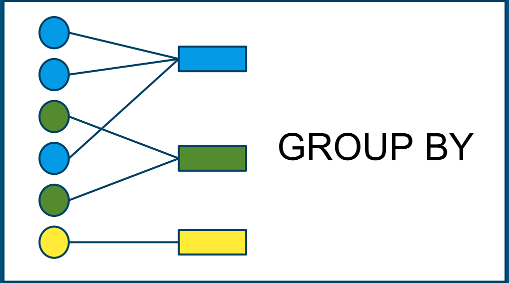
```
select productId, count(*)
from Orders
where rowtime > current_timestamp - interval ‘1’ hour
group by productId
```

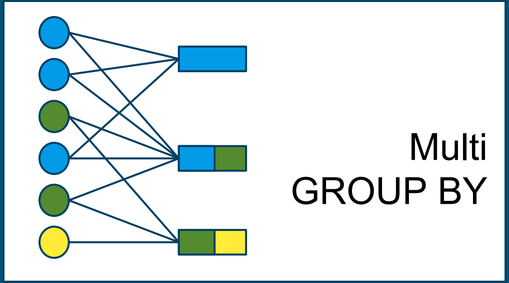
Aggregation and windows on streams

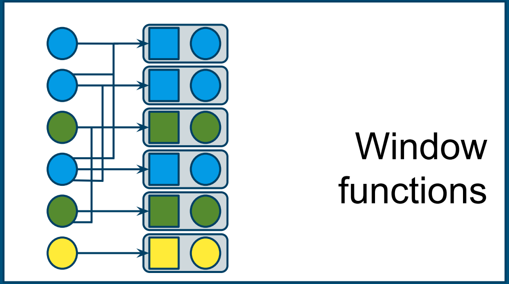
GROUP BY aggregates multiple rows into sub-totals

- ➢ 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 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

When are rows emitted?

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 punctuations)

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
```

Window types

- ➢ Tumbling window: “Every T seconds, emit the total for T seconds”

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

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

- ➢ Hopping window: “Every T seconds, emit the total for T2 seconds”

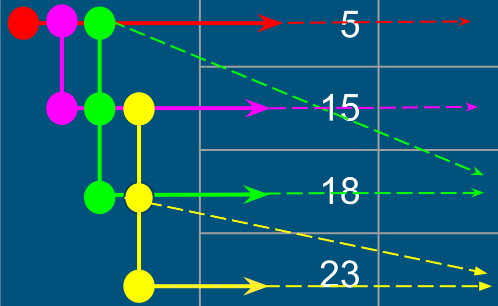
```
select stream … from Orders
group by hop(rowtime, interval ‘1’ hour, interval ‘2’ hour)
```

- ➢ Sliding window: “Every record, emit the total for the surrounding T seconds” or “Every record, emit the total for the surrounding T records” (see next slide…)

Window functions

```
select stream sum(units) over w as units1h,
  sum(units) over w (partition by productId) as units1hp,
  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



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

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.

```
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

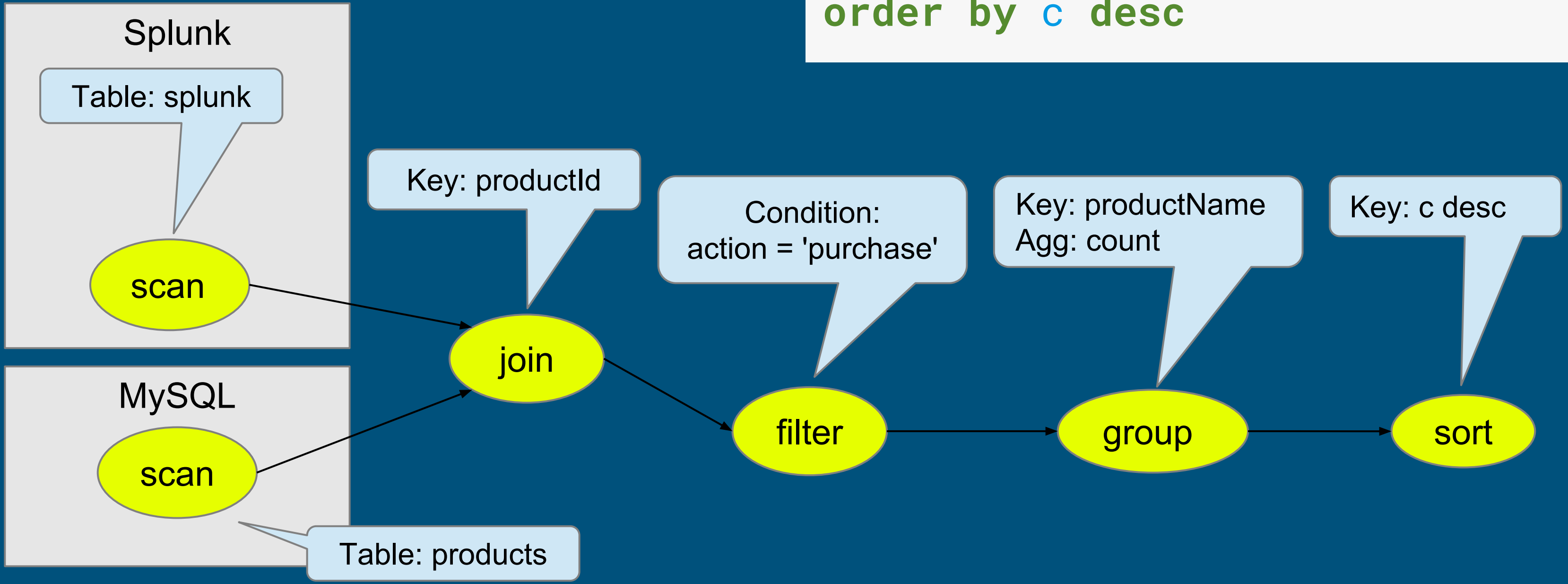
We can join streams if the join condition forces them into "lock step", within a window (in this case, 1 hour).

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
```

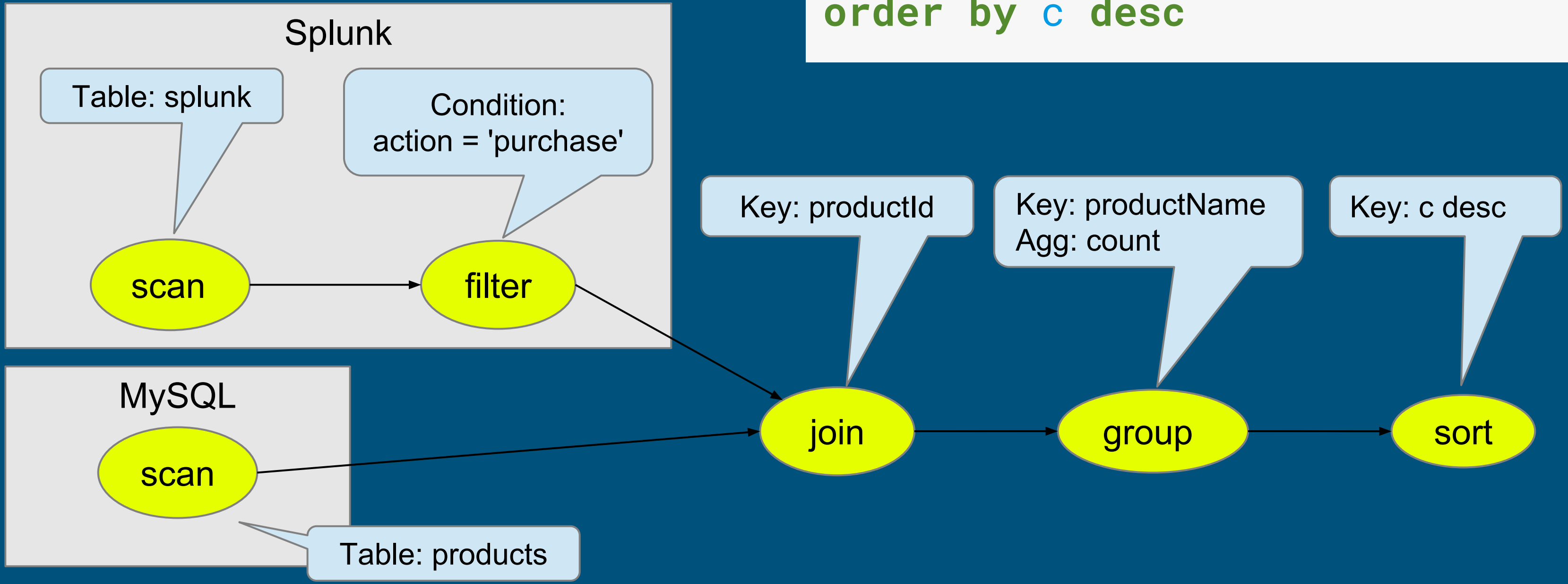
Planning queries

```
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

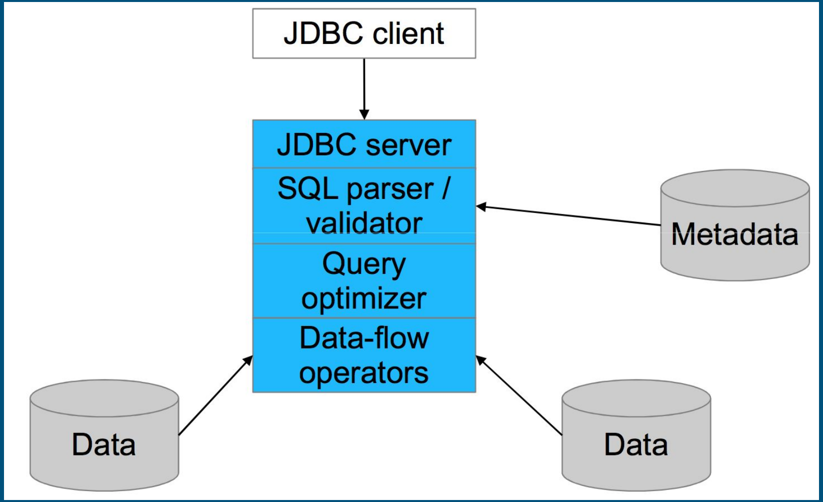
- ➢ Library
- ➢ Optional SQL parser, JDBC server
- ➢ Community-authored rules, adapters

Embedded	**Adapters**	**Streaming**
Apache Drill	Apache Cassandra*	Apache Flink*
Apache Hive	Apache Spark	Apache Samza
Apache Kylin	CSV	Apache Storm
Apache Phoenix*	In-memory	
Cascading	JDBC	
Lingual	JSON	
	MongoDB	
	Splunk	
	Web tables	

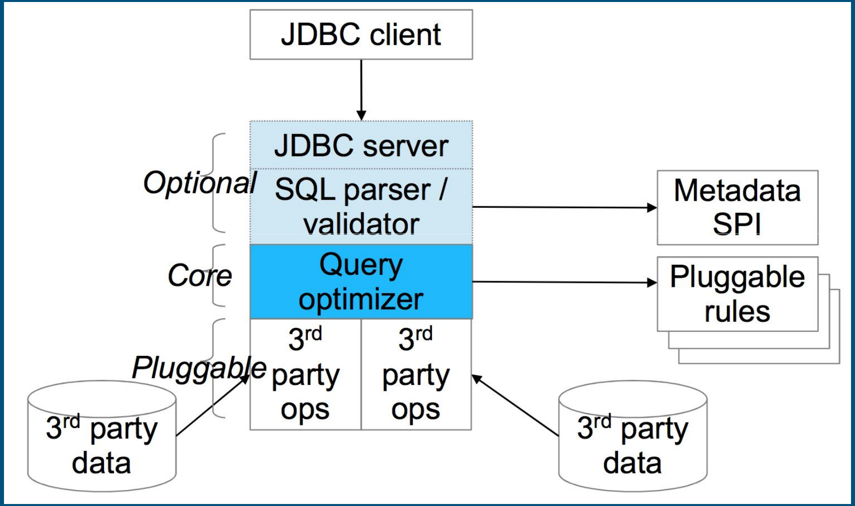
** Under development*

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

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

ORDER BY

Sorting a streaming query is valid as long as the system can make progress.

```
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
```

Union

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

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

Give existing tools and traditional users to access streaming data

Combine streaming and historic data

Streaming SQL is a superset of standard SQL

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

“Data in Flight”, Communications of the ACM, January 2010 [article]

[SAMZA-390] High-level language for SAMZA