Data all over the place!

How SQL and Apache Calcite bring sanity to streaming and spatial data

Julian Hyde | **Lyft** | 2018/06/27

@julianhyde

SQL
Query planning
Query federation
OLAP
Streaming
Hadoop

ASF member
Original author of Apache Calcite
PMC Apache Arrow, Calcite, Drill, Eagle, Kylin
Architect at Looker

Overview

Apache Calcite - What is it, exactly?

Streaming SQL / standardizing / how Calcite helps

Spatial / accelerated by materialized views / spatial + streaming

1. Calcite

Apache Calcite

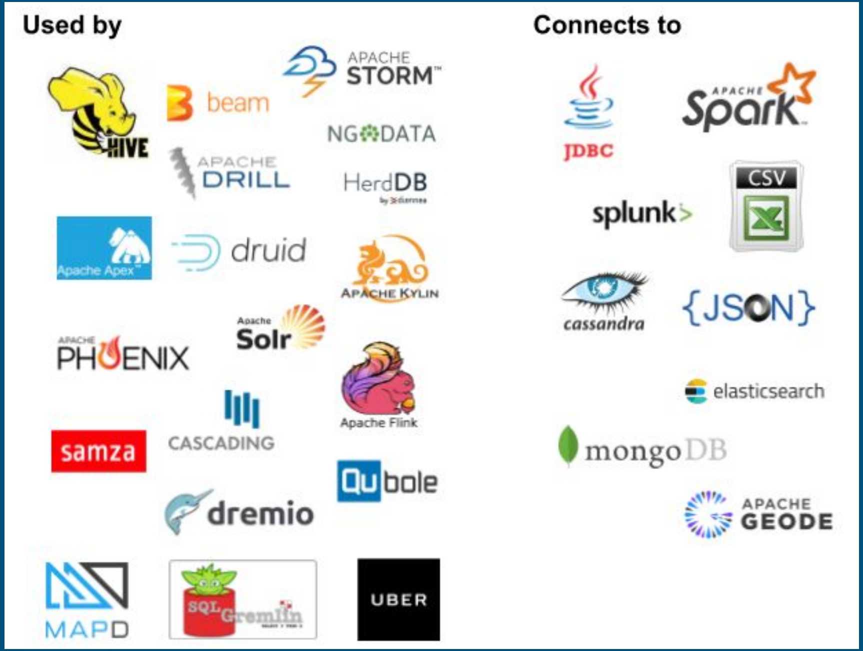
Apache top-level project

Query planning framework used in many projects and products

Also works standalone: embedded federated query engine with SQL / JDBC front end

Apache community development model

http://calcite.apache.org
http://github.com/apache/calcite



Apache Calcite - Project goals

Make it easier to write a simple DBMS

Advance the state of the art for complex DBMS by pooling resources

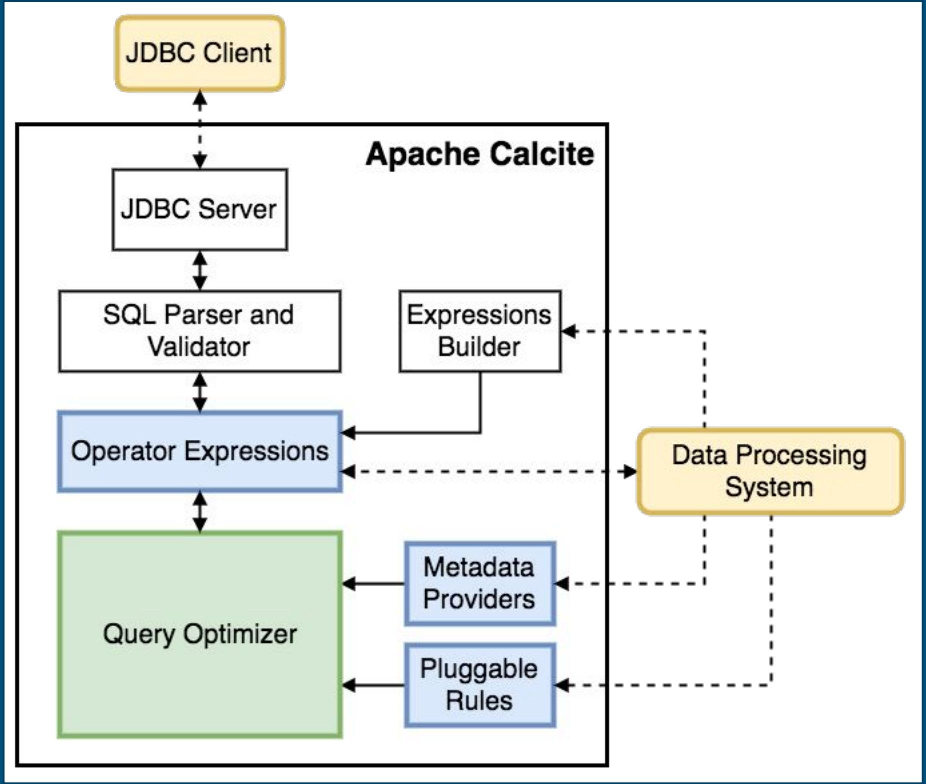
Bring database approaches to new areas (e.g. streaming)

Allow create a DBMS by composing pieces (federation, etc.)

Customize by plugging into framework, evolving framework when necessary

Apache license & governance

Architecture



Core – Operator expressions (relational algebra) and planner (based on Volcano/Cascades[2])

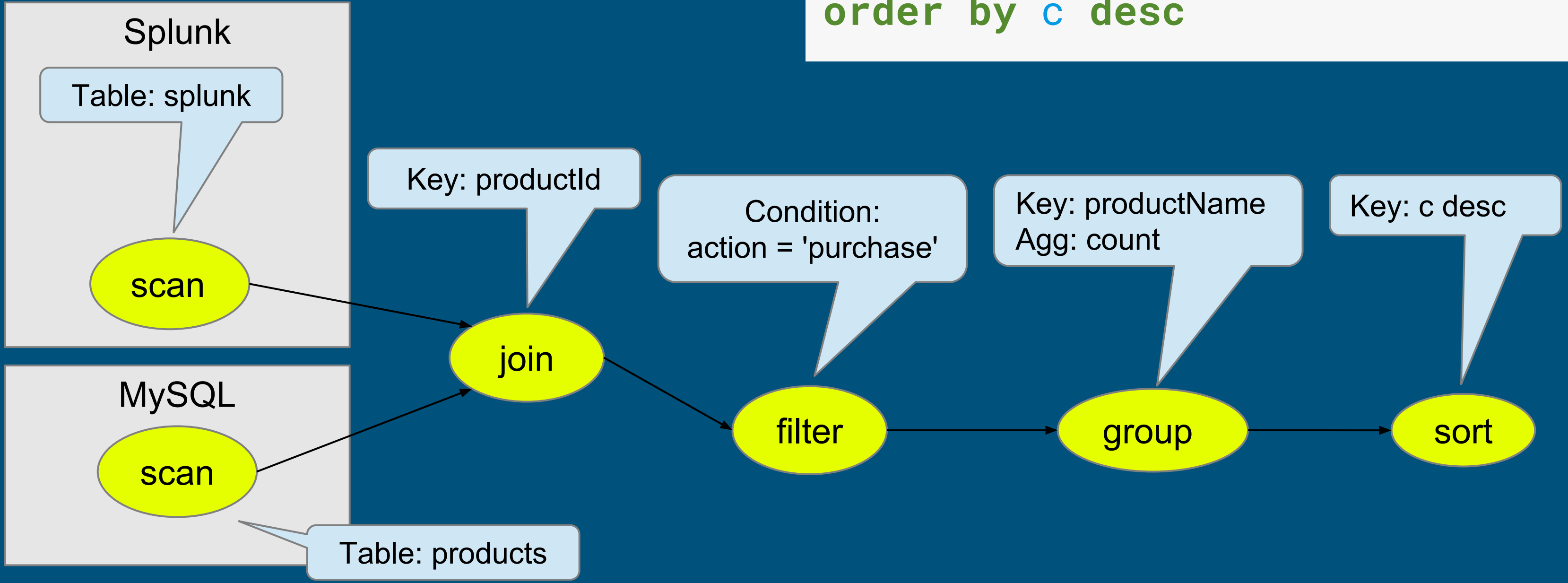
External – Data storage, algorithms and catalog

Optional – SQL parser, JDBC & ODBC drivers

Extensible – Planner rewrite rules, statistics, cost model, algebra, UDFs

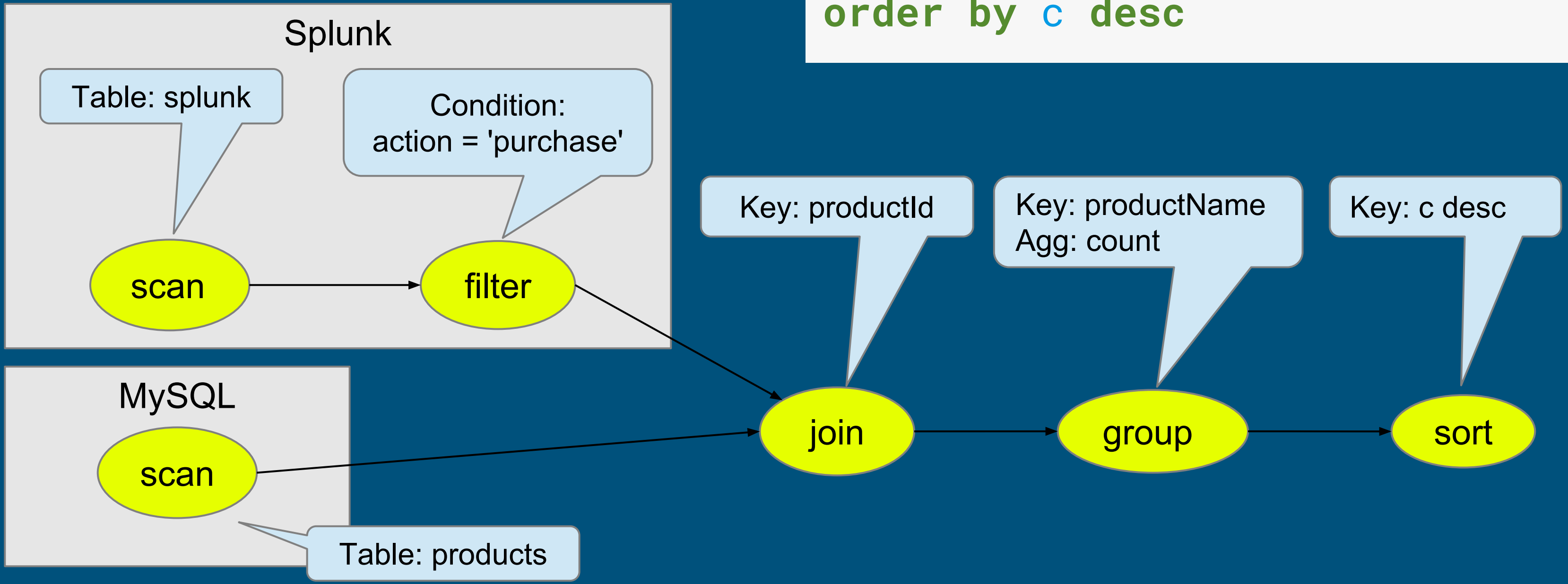
Planning queries

```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
```



Calcite framework

Relational algebra

RelNode (operator)
- TableScan
- Filter
- Project
- Union
- Aggregate
- …

RelDataType (type)
RexNode (expression)
RelTrait (physical property)
- RelConvention (calling-convention)
- RelCollation (sortedness)
- RelDistribution (partitioning)

RelBuilder

SQL parser

SqlNode
SqlParser
SqlValidator

Metadata

Schema
Table
Function
- TableFunction
- TableMacro

Lattice

JDBC driver

Transformation rules

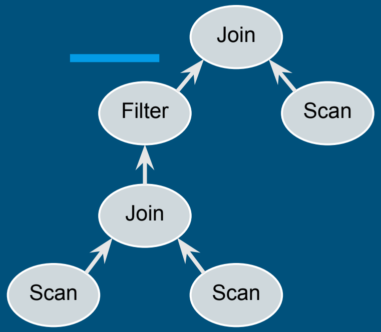
RelOptRule
- FilterMergeRule
- AggregateUnionTransposeRule
- 100+ more

Global transformations
- Unification (materialized view)
- Column trimming
- De-correlation

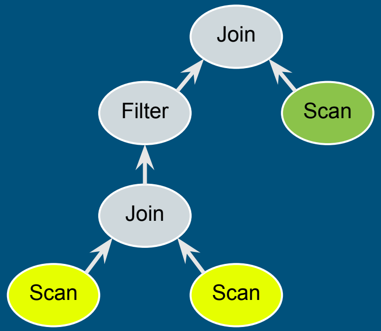
Cost, statistics

RelOptCost
RelOptCostFactory
RelMetadataProvider
- RelMdColumnUniquensss
- RelMdDistinctRowCount
- RelMdSelectivity

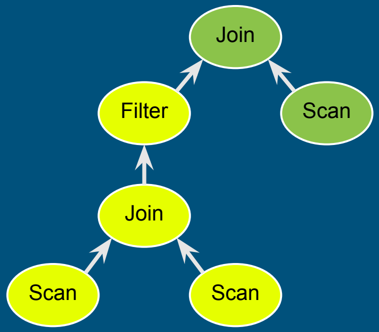
Conventions



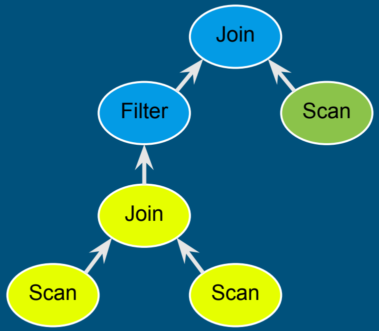
1. Plans start as logical nodes.



2. Assign each Scan its table's native convention.



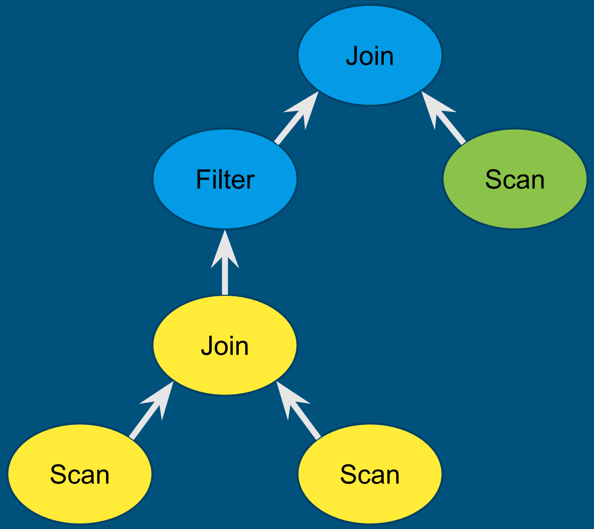
3. Fire rules to propagate conventions to other nodes.



4. The best plan may use an engine not tied to any native format.

To implement, generate a program that calls out to query1 and query2.

Adapters



Conventions provide a uniform representation for hybrid queries

Like ordering and distribution, conventions are a **physical property** of nodes

Adapter =
    schema factory (lists tables)
  + convention
  + rules to convert nodes to a convention

Example adapters: file, web, MongoDB, Druid, Drill, Spark, Flink

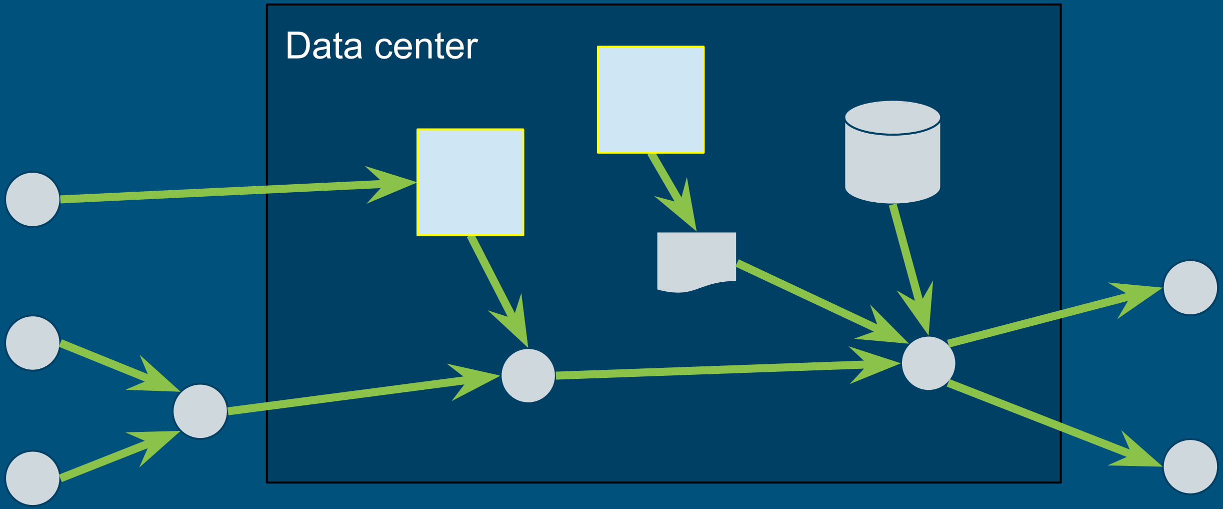
2. Streaming

Why SQL?

Data in motion, data at rest - it's all just data

Stream / database duality

SQL is the best language ever created for data (because it's declarative)



Calcite and streaming SQL

Building a streaming SQL standard via consensus

- Please! No more “SQL-like” languages!

Toolkit for building a streaming SQL engine

- Key technologies are open source (many are Apache projects)
- Use Calcite’s framework to build a streaming SQL parser/planner for your engine

Developing example queries, data sets, TCK

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

Hybrid query combines a stream with its own history

- `Orders` is used as both as stream and as "stream history" virtual table
- "Average order size over last year" should be maintained by the system, i.e. a materialized view

"Orders" used as a stream

"Orders" used as a "stream history" virtual table

```
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)
```

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
- `match_recognize` – Complex event processing (CEP)

“Standard streaming SQL”

Consistent semantics over tables (if database contains both streams and tables)

Consistent semantics over streams. 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.*

Consistent look and feel (e.g. same data types, keywords)

Some hard problems remain... mainly concerning time…

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
```

Event time

Example streams and query so far have used a `rowtime` column

- Is `rowtime` a system column, or can streams choose their own sort key?
- Can a stream have more than one sort key?
- Is `rowtime`'s type always TIMESTAMP?
- Is a stream totally sorted on `rowtime`? Alternatives:
    - K-sorted: Every row is within 100 rows of sorted
    - T-sorted: Every row is within 5 minutes of sorted
    - Bounded latency: Every row is within 10 minutes of wallclock time
    - Watermark: An upper-bound on rows' delay, allows early emit

Event time in queries

We’ve sorted out event time in streams… what about derived streams (queries / views)?

Given a streaming query:

- What are its sort keys?
- What is the maximum delay of rows?
- Does the system refuse to run queries with unbounded delay?

SQL syntax to promote a query column to a sort key

```
select ~~count(*)~~
from ~~Orders~~
```

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

```
select shipTime
  as rowtime
from Orders
order by shipTime
```

?

Models of relations

A ***table*** is a multi-set of records; contents vary over time, with some kind transactional consistency

A ***temporal table*** is like a table, but you can also query its contents at previous times; usually mapped to a base table with (start, end) columns

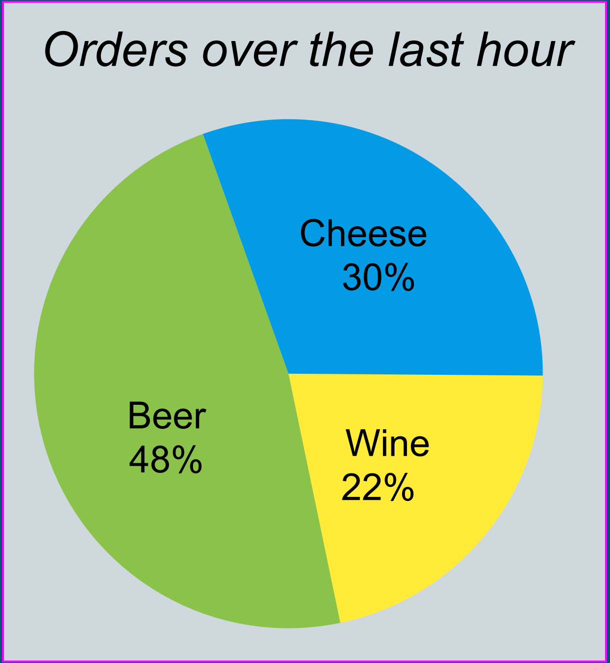
So, what is a stream?

1. A ***stream*** is an append-only table with an event-time column
2. Or, if you speak monads[3], a ***stream*** is a function: $Stream\ x \longrightarrow Time \longrightarrow Bag\ x$
3. Or, a ***stream*** is the time-derivative of a table

Are a table, its temporal table, and its stream one catalog object or three?

The “pie chart” problem

- **Task** – Write dashboard of orders over last hour
- **Problem** – The `Orders` stream only contains the current few records
- **Solution** – Materialize short-term history
- **Question** – Does the developer maintain that mapping, or does the DBMS do it automatically?



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

Transactions & isolation

In a regular database, INSERT, UPDATE and DELETE are transactional operations; SELECT is not

In a streaming database, SELECT might be transactional too

- Consider a pub-sub system: the system acknowledges when you send a message, you must acknowledge when you receive a message

In a regular database query, snapshot isolation freezes the database in time; in a streaming query, we don't want that

Joins

Stream to stream join

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

Stream to table join

```
select stream *
from Orders as o
join Products as p
  on p.productId = o.productId
```

Stream to temporal table join

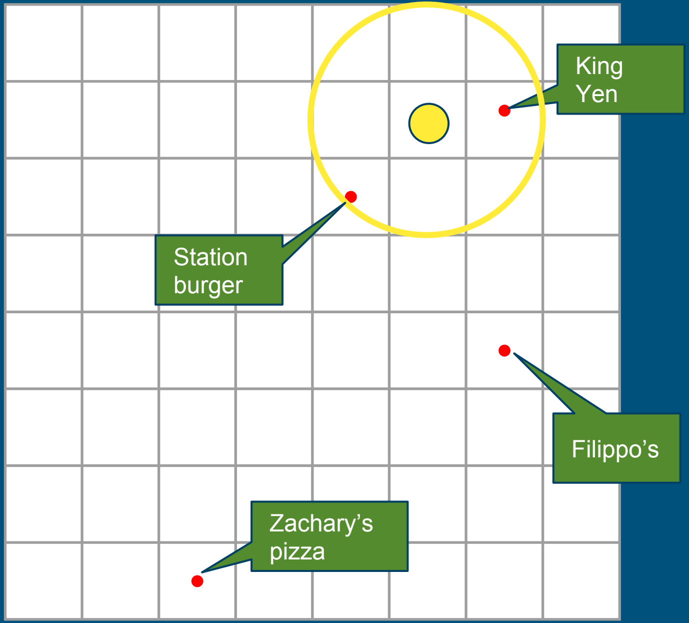
```
select stream *
from Orders as o
join TemporalProducts as of o.rowtime
    as p
  on p.productId = o.productId
```

"as of" is based on SQL:2011 temporal syntax

3. Spatial

Spatial query

Find all restaurants within 1.5 distance units of my location (6, 7)



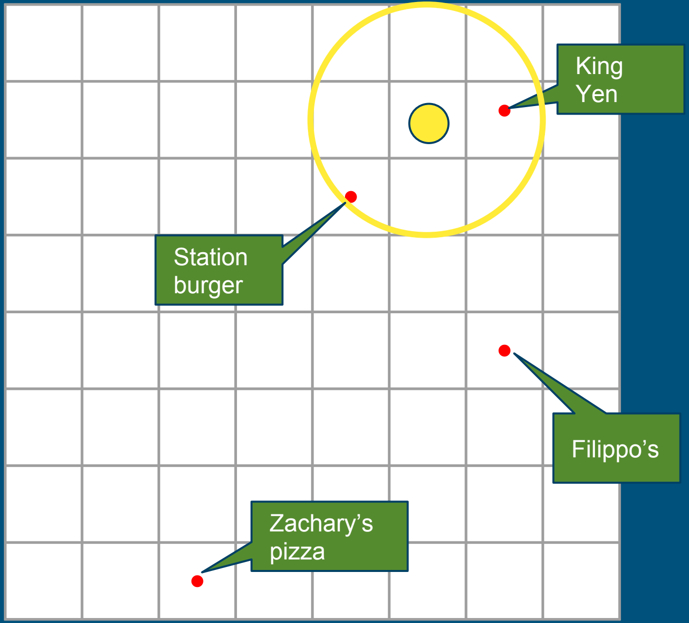
restaurant	x	y
Zachary’s pizza	3	1
King Yen	7	7
Filippo’s	7	4
Station burger	5	6

Spatial query

Find all restaurants within 1.5 distance units of my location (6, 7)

Using OpenGIS SQL extensions:

```
SELECT *
FROM Restaurants AS r
WHERE ST_Distance(
  ST_MakePoint(r.x, r.y),
  ST_MakePoint(6, 7)) < 1.5
```



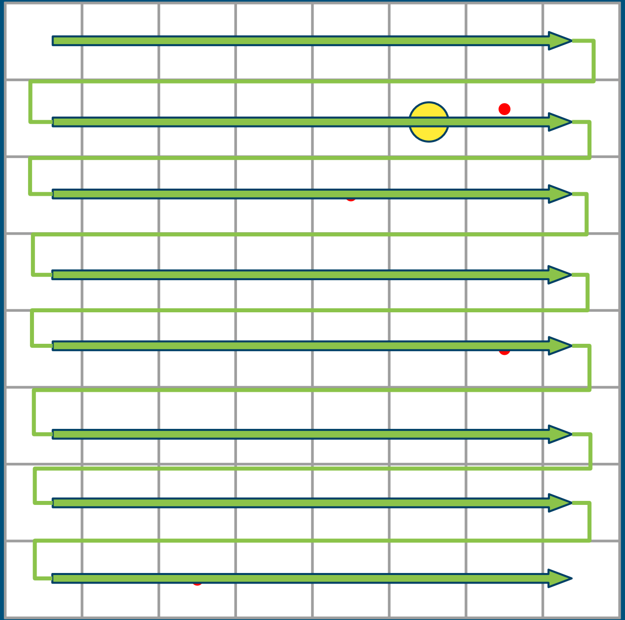
restaurant	**x**	**y**
Zachary's pizza	3	1
King Yen	7	7
Filippo's	7	4
Station burger	5	6

Efficient spatial query?

Spatial queries are hard to execute efficiently.

Conventional DBs use two tricks that don't work:

- Hash (no ranges / no locality)
- Sort (only one-dimensional ranges)

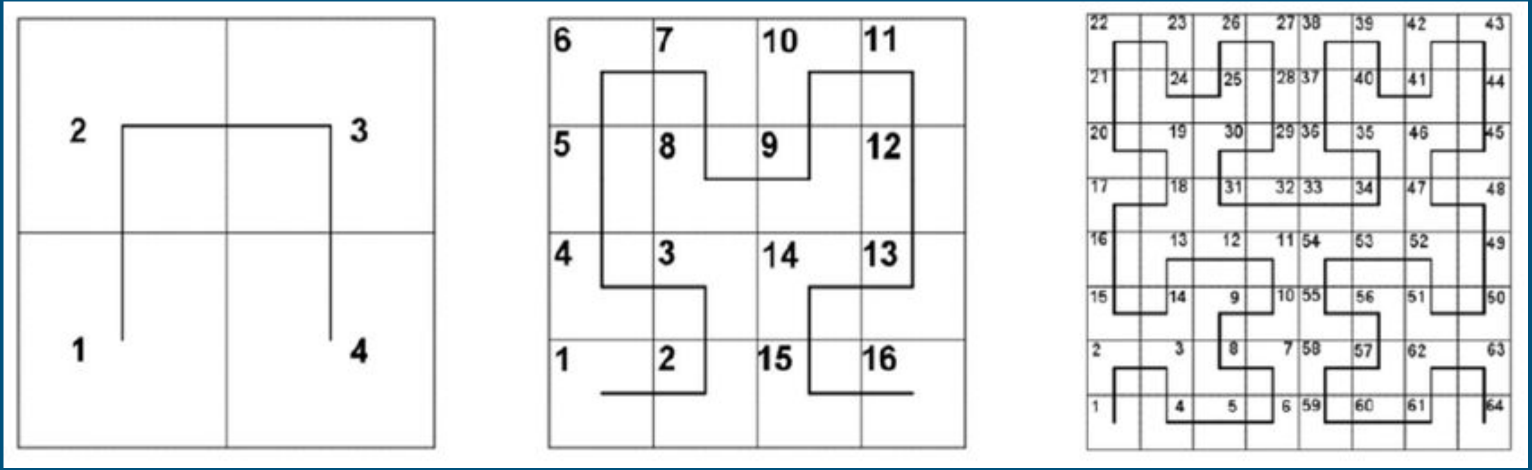

A scan over a two-dimensional index only has locality in one dimension

Two techniques that do work[4]:

- Data-organized structures (e.g. r-tree) – self-balancing
- Space-organized structures (e.g. tiles) – allow algebraic rewrite

Algebraic rewrites work on general-purpose data structures, and they compose.

Hilbert space-filling curve



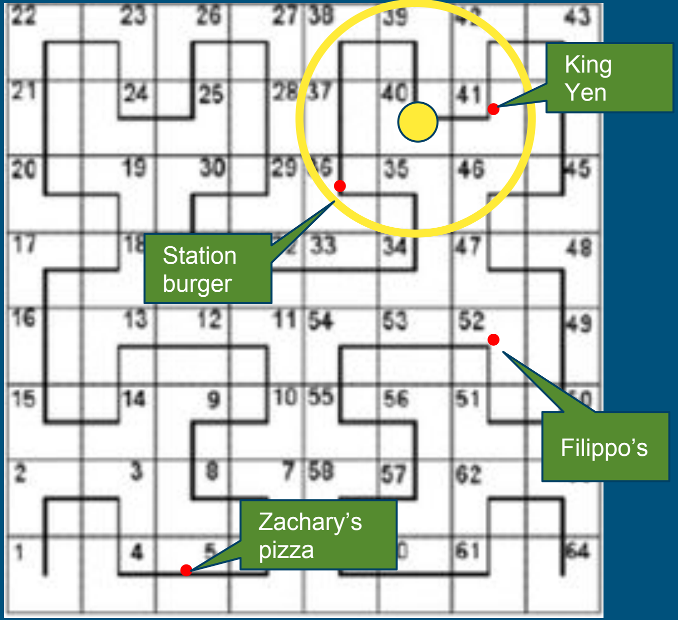
- A space-filling curve invented by mathematician David Hilbert
- Every (x, y) point has a unique position on the curve
- Points near to each other typically have Hilbert indexes close together

Using Hilbert index

Add restriction based on **h**, a restaurant's distance along the Hilbert curve

Must keep original restriction due to false positives

```
SELECT *
FROM Restaurants AS r
WHERE (r.h BETWEEN 35 AND 42
    OR r.h BETWEEN 46 AND 46)
AND ST_Distance(
  ST_MakePoint(r.x, r.y),
  ST_MakePoint(6, 7)) < 1.5
```



restaurant	x	y	h
Zachary's pizza	3	1	5
King Yen	7	7	41
Filippo's	7	4	52
Station burger	5	6	36

Telling the optimizer

1. Declare h as a generated column
2. Sort table by h

Planner can now convert spatial range queries into a range scan

Does not require specialized spatial index such as r-tree

Very efficient on a sorted table such as HBase

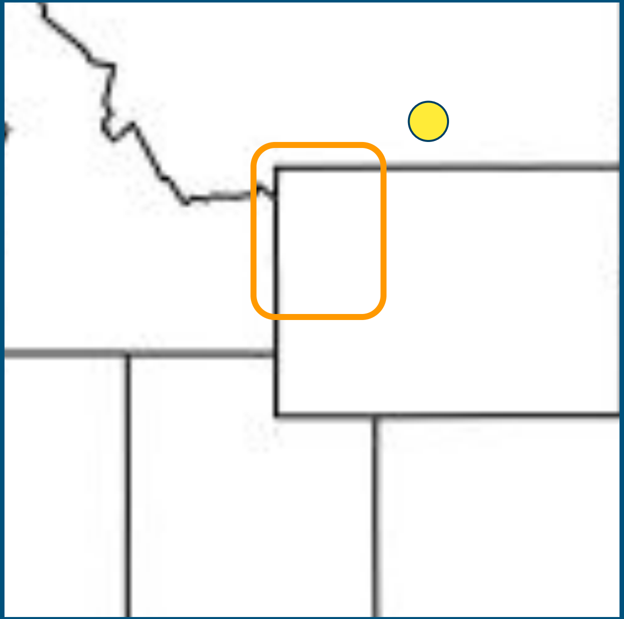
```
CREATE TABLE Restaurants (
  restaurant VARCHAR(20),
  x DOUBLE,
  y DOUBLE,
  h DOUBLE GENERATED ALWAYS AS
     ST_Hilbert(x, y) STORED)
SORT KEY (h);
```

restaurant	x	y	h
Zachary's pizza	3	1	5
Station burger	5	6	36
King Yen	7	7	41
Filippo's	7	4	52

More spatial queries

What state am I in? (point-to-polygon)

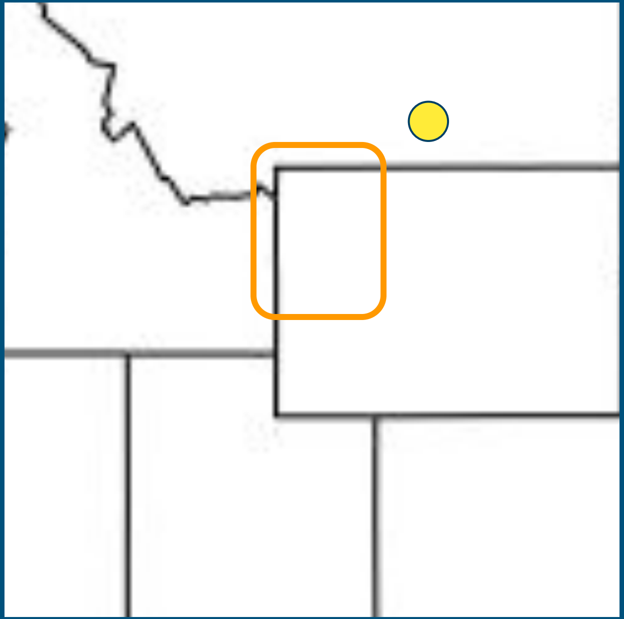
Which states does Yellowstone NP intersect? (polygon-to-polygon)



More spatial queries

What state am I in? (point-to-polygon)

Which states does Yellowstone NP intersect? (polygon-to-polygon)



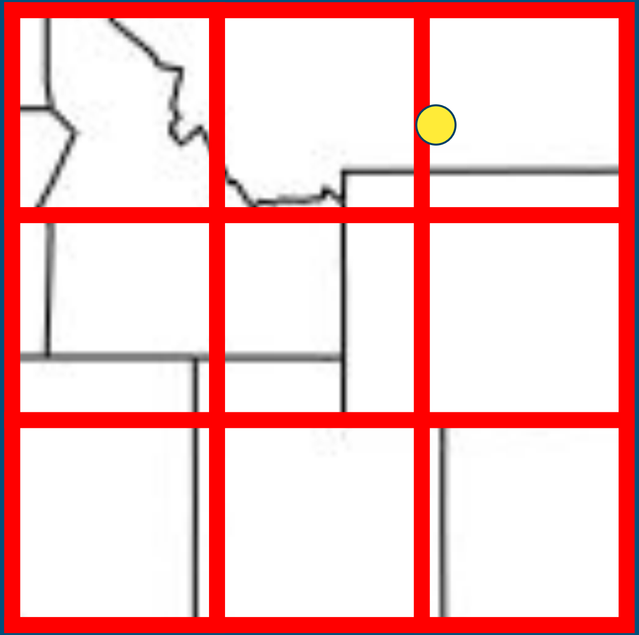
```
SELECT *
FROM States AS s
WHERE ST_Intersects(s.geometry,
  ST_MakePoint(6, 7))
```

```
SELECT *
FROM States AS s
WHERE ST_Intersects(s.geometry,
  ST_GeomFromText('LINESTRING(...)'))
```

Point-to-polygon query

What state am I in? (point-to-polygon)

1. Divide the plane into tiles, and pre-compute the state-tile intersections
2. Use this ‘tile index’ to narrow list of states



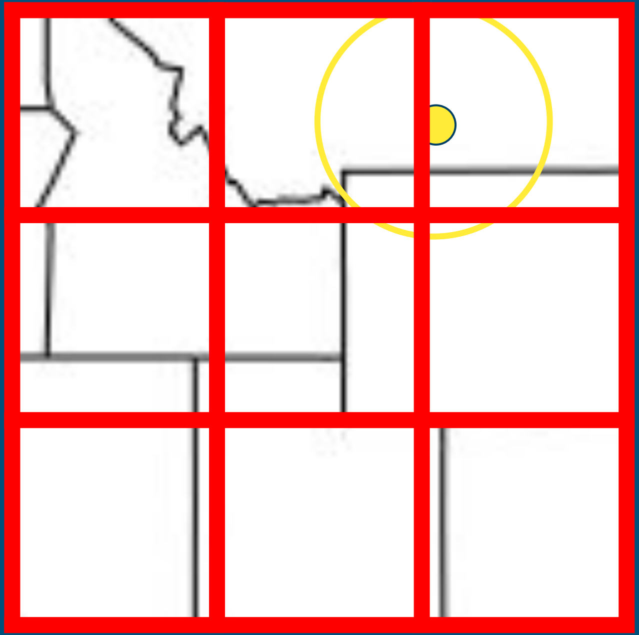
```
SELECT s.*
FROM States AS s
WHERE s.stateId IN (SELECT stateId
  FROM StateTiles AS t
  WHERE t.tileId = 8)
AND ST_Intersects(s.geometry, ST_MakePoint(6, 7))
```

Building the tile index

Use the ST_MakeGrid function to decompose each state into a series of tiles

Store the results in a table, sorted by tile id

A materialized view is a table that remembers how it was computed, so the planner can rewrite queries to use it



```
CREATE MATERIALIZED VIEW StateTiles AS
SELECT s.stateId, t.geometry, t.tileId, t.id_col, t.id_row
FROM States AS s,
 LATERAL TABLE(ST_MakeGrid(s.stateId, 4, 4)) AS t
```

Streaming + spatial

Every minute, emit the number of journeys that have intersected each city. (Some journeys intersect multiple cities.)

(Efficient implementation is left as an exercise to the reader. Probably involves splitting journeys into tiles, partitioning by tile hash-code, intersecting with cities in those tiles, then rolling up cities.)

```
SELECT STREAM c.name, COUNT(*)
FROM Journeys AS j
CROSS JOIN Cities AS c
  ON ST_Intersects(c.geometry, j.geometry)
GROUP BY c.name, FLOOR(j.rowtime TO HOUR)
```

4. Conclusion

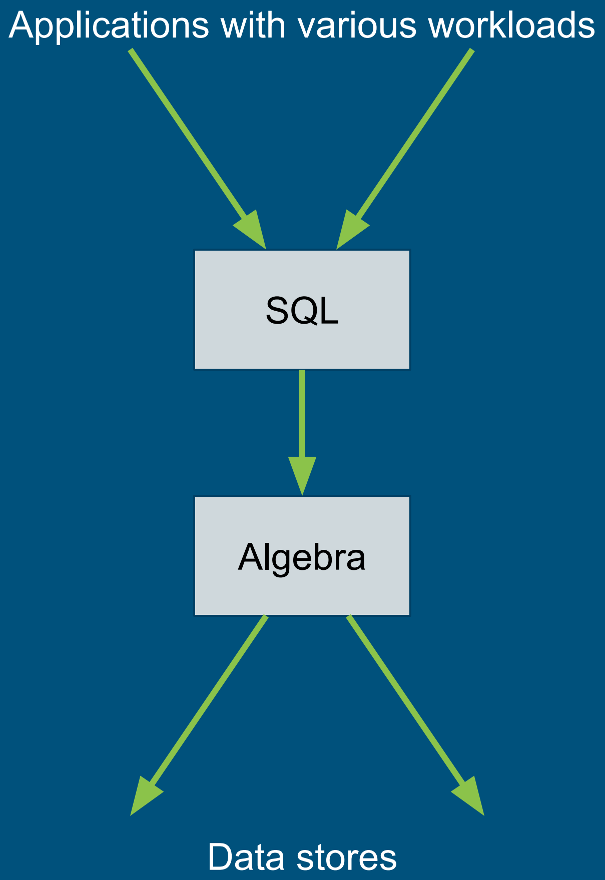
Why SQL?

Common language across paradigms (OLTP, analytics, streaming, spatial, graph, search…) and workloads (batch, transactional, continuous)

SQL is standard (and growing)

Relational algebra allows a planner to transform queries

Materialized views (indexes, caches) allow you to optimize your data



Thank you! Questions?

References

1. Edmon Begoli, Jesús Camacho Rodríguez, Julian Hyde, Michael Mior, Daniel Lemire. 2018. Apache Calcite: A Foundational Framework for Optimized Query Processing Over Heterogeneous Data Sources. SIGMOD 2018.
2. Goetz Graefe and William J. McKenna. 1993. The Volcano Optimizer Generator: Extensibility and Efficient Search. ICDE.
3. Sebastian Herbst , Andreas M. Wahl, Johannes Tenschert, Klaus Meyer-Wegener. 2018. Sequences, yet Functions: The Dual Nature of Data-Stream Processing. arXiv preprint.
4. Edwin H. Jacox, Hanan Samet. 2007. Spatial join techniques. TODS.

https://calcite.apache.org
@ApacheCalcite
@julianhyde

THE APACHE® SOFTWARE FOUNDATION

Apache Calcite

looker

Extra slides

Talk proposal

Bio
Julian Hyde is an expert in query optimization and in-memory analytics. He founded Apache Calcite, a framework for query optimization and data virtualization. He also founded Mondrian, the popular open source OLAP engine, and co-founded SQLstream, an early streaming SQL platform. He is an architect at Looker.

Title
Data all over the place! How Apache Calcite brings SQL and sanity to streaming and spatial data

Abstract
The revolution has happened. We are living the age of the deconstructed database. The modern enterprises are powered by data, and that data lives in many formats and locations, in-flight and at rest, but somewhat surprisingly, the lingua franca for remains SQL. In this talk, Julian describes Apache Calcite, a toolkit for relational algebra that powers many systems including Apache Beam, Flink and Hive. He discusses some areas of development in Calcite: streaming SQL, materialized views, enabling spatial query on vanilla databases, and what a mash-up of all three might look like.

Other applications of data profiling

Query optimization:

- Planners are poor at estimating selectivity of conditions after N-way join (especially on real data)
- New join-order benchmark: “Movies made by French directors tend to have French actors”
- Predict number of reducers in MapReduce & Spark

“Grokking” a data set

Identifying problems in normalization, partitioning, quality

Applications in machine learning?

Further improvements

- Build sketches in parallel
- Run algorithm in a distributed framework (Spark or MapReduce)
- Compute histograms
  - For example, Median age for male/female customers
- Seek out functional dependencies
  - Once you know FDs, a lot of cardinalities are no longer “surprising”
  - FDs occur in denormalized tables, e.g. star schemas
- Smarter criteria for stopping algorithm
- Skew/heavy hitters. Are some values much more frequent than others?
- Conditional cardinalities and functional dependencies
  - Does one partition of the data behave differently from others? (e.g. year=2005, state=LA)

Relational algebra (plus streaming)

Core operators:

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

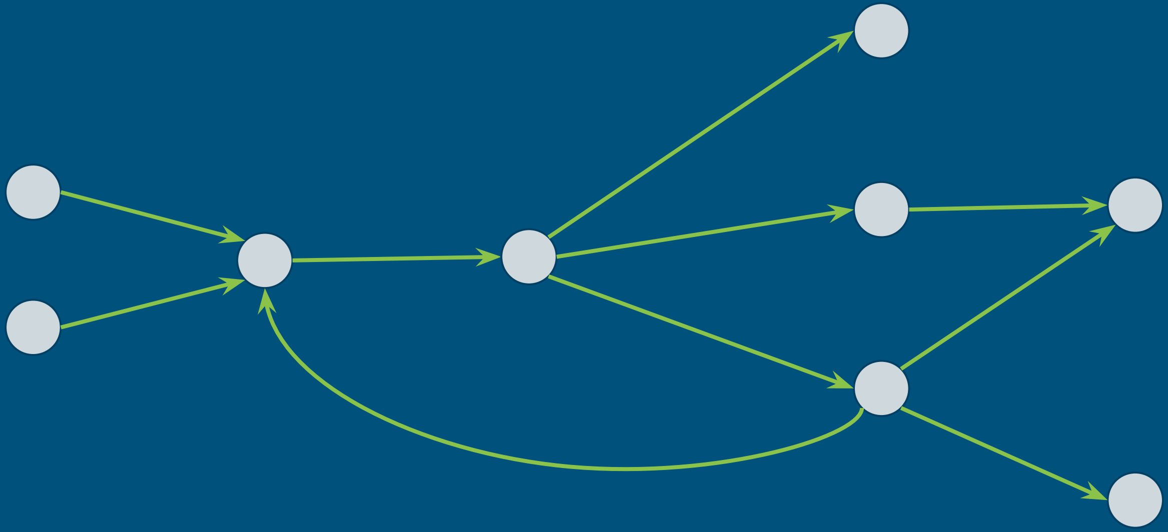
Streaming operators:

- ➢ 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.

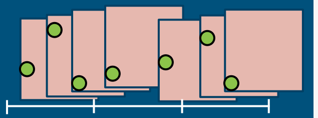
The transformations for delta are mostly simple:

- ➢ 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:

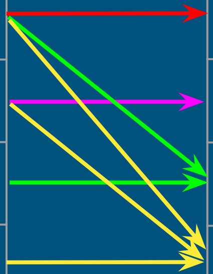
- ➢ Delta(Join(r0, r1, predicate)) → Union(Join(r0, Delta(r1)), Join(Delta(r0), r1)
- ➢ Delta(Scan(aTable)) → Empty

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

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
```

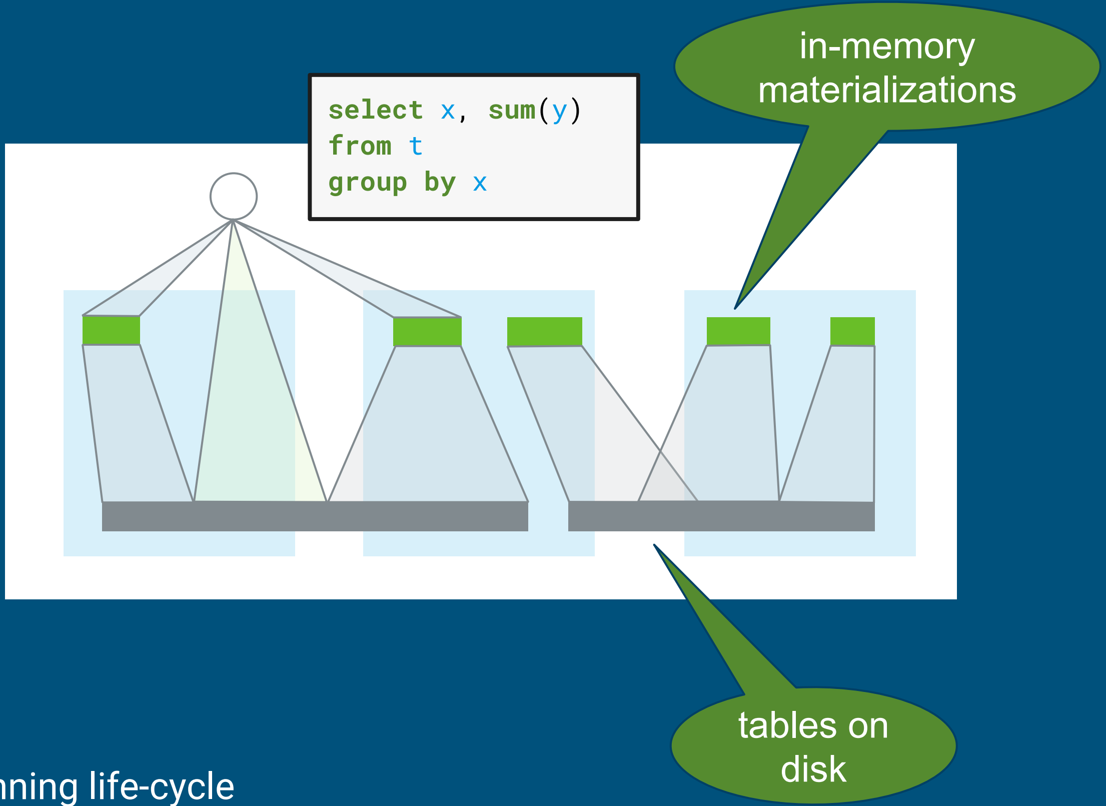
Calcite design

Design goals:

- Not-just-SQL front end
- Federation
- Extensibility
- Caching / hybrid storage
- Materialized views

Design points:

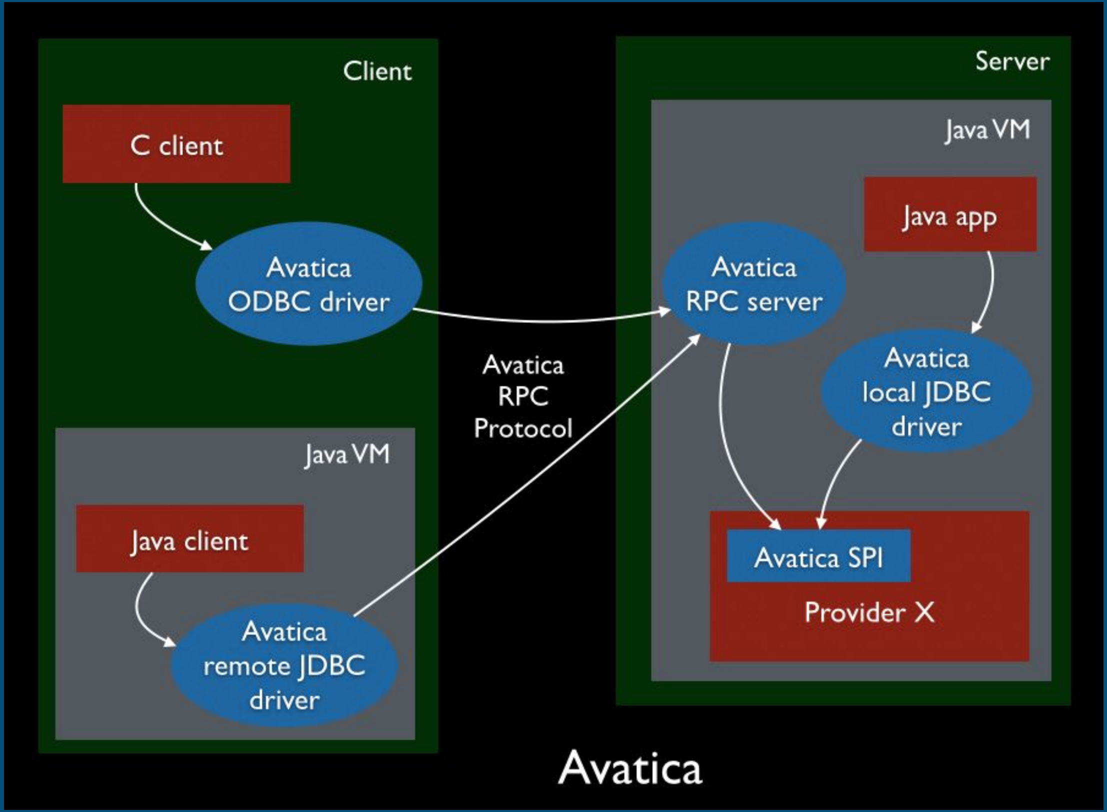
- Relational algebra foundation
- Composable transformation rules
- Support different physical engines
- Pluggable cost model, statistics, planning life-cycle



```
select x, sum(y)
from t
group by x
```

Avatica

- Database connectivity stack
- Self-contained sub-project of Calcite
- Fast, open, stable
- Protobuf or JSON over HTTP
- Powers Phoenix Query Server

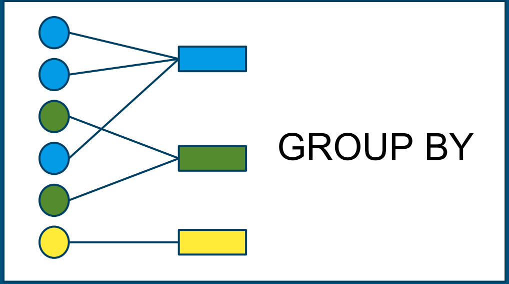


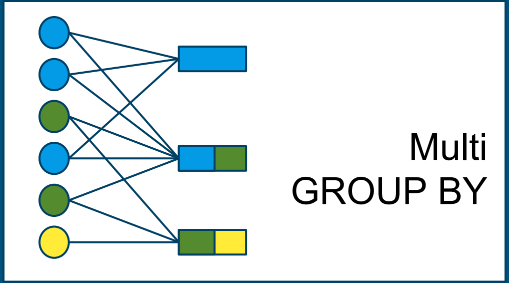
Aggregation and windows on streams

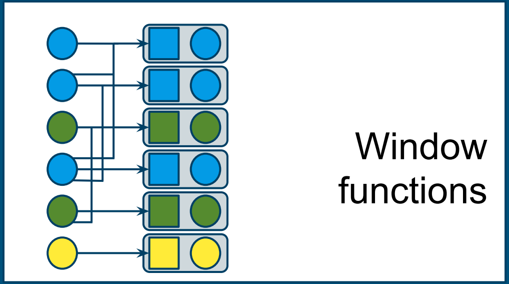
GROUP BY aggregates multiple rows into sub-totals

- ➢ 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)

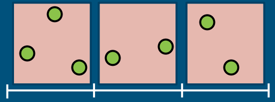






Tumbling, hopping & session windows in SQL

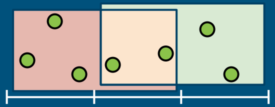
Tumbling window



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

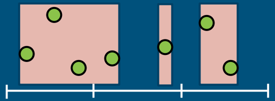
```
select stream … from Orders
group by tumble(rowtime, interval ‘1’ hour)
```

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)
```