Lazy beats Smart & Fast

Julian Hyde | **DataEngConf SF**
2018/04/17

@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

looker

A “simple” query

Data

- 2010 U.S. census
- 100 million records
- 1KB per record
- 100 GB total

System

- 4x SATA 3 disks
- Total read throughput 1 GB/s

Query

```
SELECT SUM(householdSize)
FROM CensusHouseholds;
```

Goal

- Compute the answer to the query in under 5 seconds

Solutions

Sequential scan	Query takes 100 s (100 GB at 1 GB/s)
Parallelize	Spread the data over 40 disks in 10 machines Query takes 10 s
Cache	Keep the data in memory 2nd query: 10 ms 3rd query: 10 s
Materialize	Summarize the data on disk All queries: 100 ms
Materialize + cache + adapt	As above, building summaries on demand

Lazy > Smart + Fast

(Lazy + adaptive is even better)

Overview

How do you tune a data system? How can (or should) a data system tune itself?

What problems have we solved to bring these things to Apache Calcite?

Part 1: Strategies for organizing data

Part 2: How to make systems self-organizing?

Relational algebra

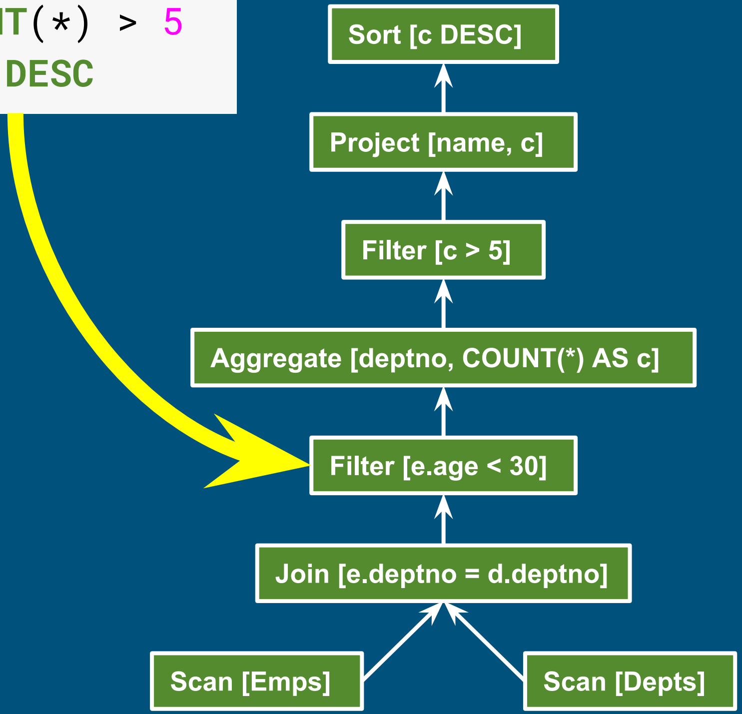
```sql
SELECT d.name, COUNT(*) AS c
FROM Emps AS e
JOIN Depts AS d USING (deptno)
WHERE e.age < 40
GROUP BY d.deptno
HAVING COUNT(*) > 5
ORDER BY c DESC
```

Based on set theory, plus operators: Project, Filter, Aggregate, Union, Join, Sort

Requires: declarative language (SQL), query planner

Original goal: data independence

Enables: query optimization, new algorithms and data structures



Apache Calcite

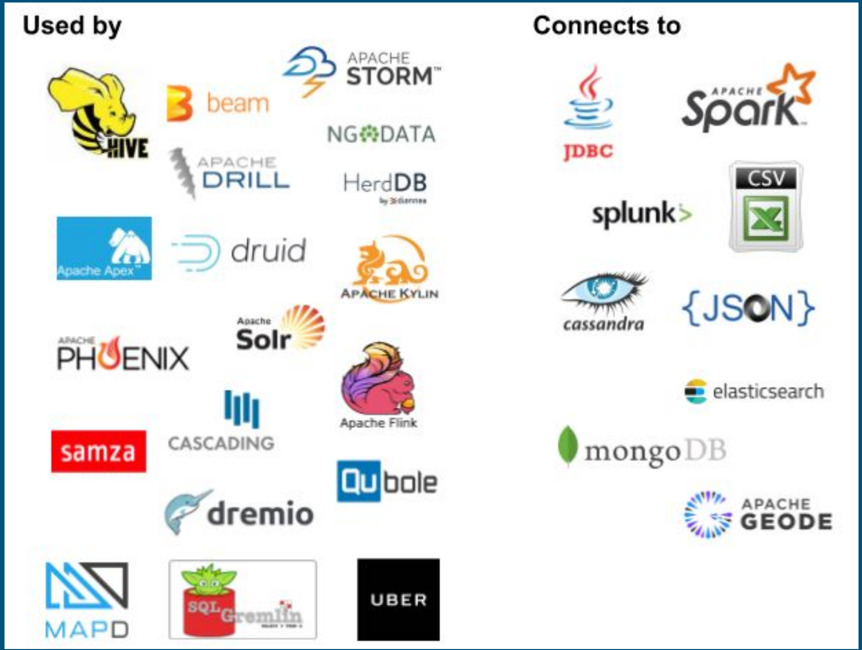
Apache top-level project

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



1. Organizing data

Ways of organizing data

Format (CSV, JSON, binary)

Layout: row- vs. column-oriented (e.g. Parquet, ORC), cache friendly (e.g. Arrow)

Storage medium (disk, flash, RAM, NVRAM, ...)

Non-lossy copy: sorted / partitioned

Lossy copies of data: project, filter, aggregate, join

Combinations of the above

Logical optimizations >> physical optimizations

Index

A sorted, projected materialized view

Accelerates queries that use ranges, correlated lookups, sorting, aggregate, distinct

```sql
CREATE TABLE Emp (empno INT,
 name VARCHAR(20), deptno INT);

CREATE INDEX I_Emp_Deptno
ON Emp (deptno, name);

SELECT DISTINCT deptno FROM Emp
WHERE deptno BETWEEN 20 AND 40
ORDER BY deptno;
```

empno	name	deptno
100	Fred	20
110	Barney	10
120	Wilma	30
130	Dino	10

→

deptno	name	rowid
10	Barney	af5634.0001
10	Dino	af5634.0003
20	Fred	af5634.0000
30	Wilma	af5634.0002

Covering index

Add the remaining columns

No longer need “rowid”

Lossless

During planning, treat indexes as tables, and index lookups as joins

```
CREATE INDEX I_Emp_Deptno2 (
  deptno INTEGER,
  name VARCHAR(20))
COVER (empno);
```

empno	name	deptno
100	Fred	20
110	Barney	10
120	Wilma	30
130	Dino	10

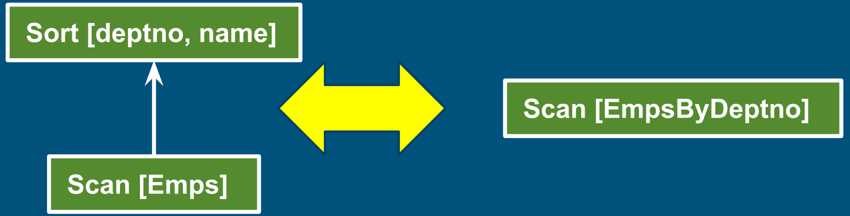
deptno	name	empno
10	Barney	100
10	Dino	130
20	Fred	20
30	Wilma	30

Materialized view

As a materialized view, an index is now just another table

Several tables contain the information necessary to answer the query - just pick the best

```
CREATE MATERIALIZED
  VIEW EmpsByDeptno AS
SELECT deptno, name, deptno
FROM Emp
ORDER BY deptno, name;
```



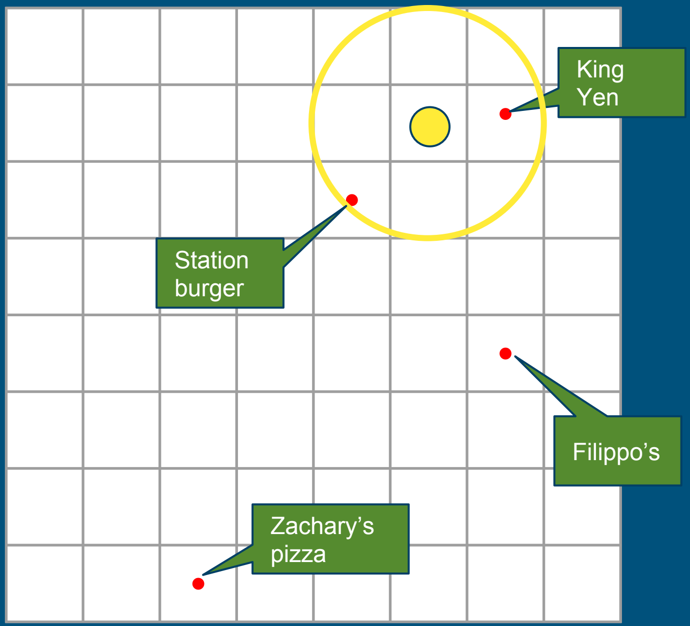
empno	name	deptno
100	Fred	20
110	Barney	10
120	Wilma	30
130	Dino	10

deptno	name	empno
10	Barney	100
10	Dino	130
20	Fred	20
30	Wilma	30

Spatial query

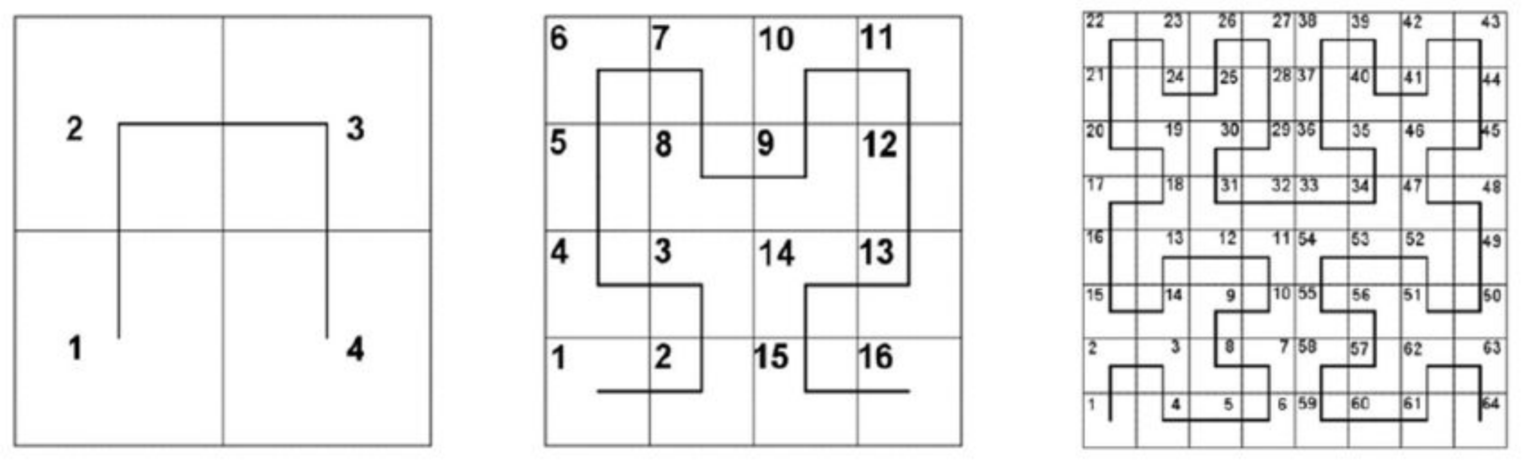
Find all restaurants within 1.5 distance units of where I am:

```
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

Hilbert space-filling curve



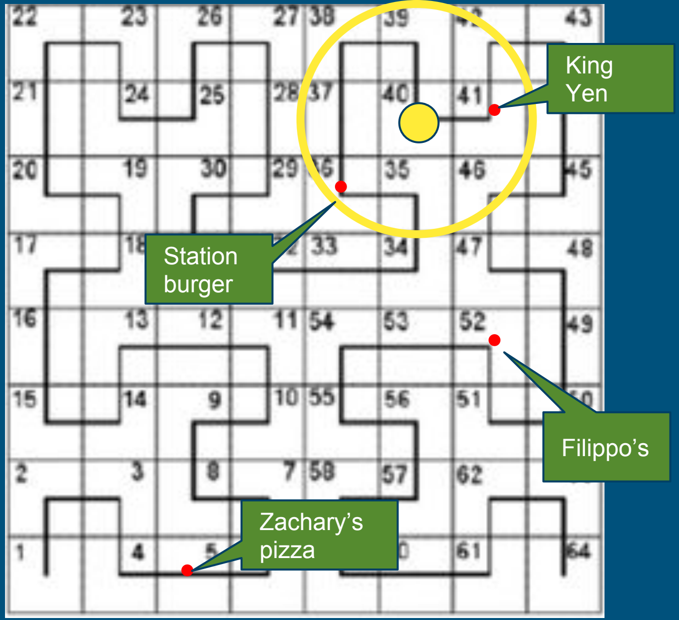
- A space-filling curve invented by mathematician David Hilbert
- Every (x, y) point has a unique position on the curve
- Points near to each other typically have Hilbert indexes close together

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

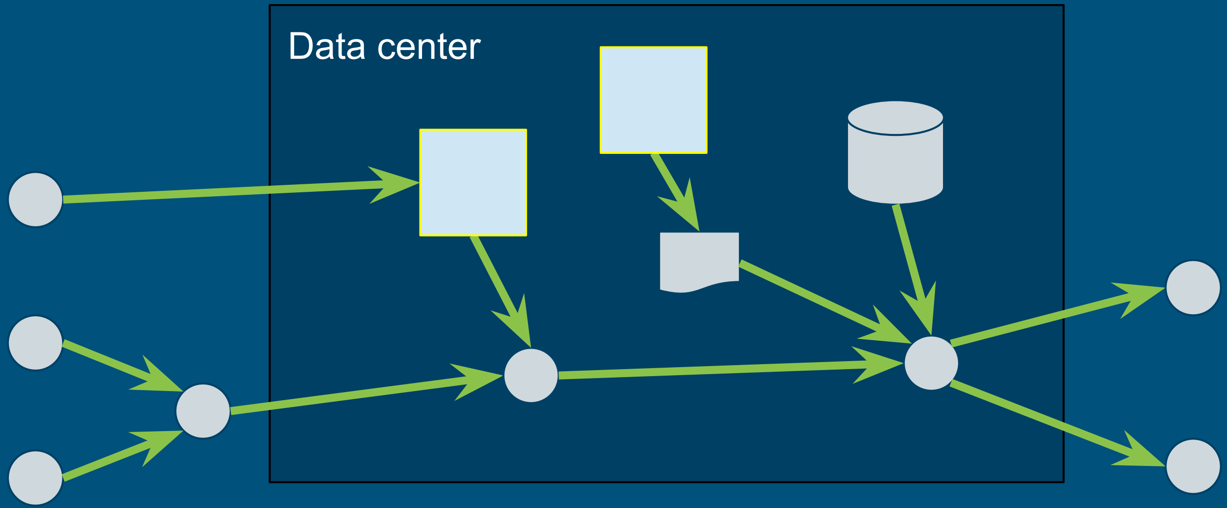
```
CREATE TABLE Restaurants (
  restaurant VARCHAR(20),
  x DOUBLE,
  y DOUBLE,
  h DOUBLE GENERATED ALWAYS AS
     ST_Hilbert(x, y) STORED)
SORT KEY (h);
```

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

Streaming

Much valuable data is “data in flight”

Use SQL to query streams (or streams + tables)



Streaming query

```
SELECT STREAM *
FROM Orders
WHERE units > 1000
```

Historic query

```
SELECT AVG(unitPrice)
FROM Orders
WHERE units > 1000
AND orderDate
  BETWEEN ‘2014-06-01’
  AND ‘2015-12-31’
```

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)
```

Summary - data optimization via materialized views

Many forms of data optimization can be modeled as materialized views:

- Blocks in cache
- B-tree indexes
- Summary tables
- Spatial indexes
- History of streams

Allows the optimizer to “understand” the optimization and use it (if beneficial)

But who designs the optimizations?

2. Learning

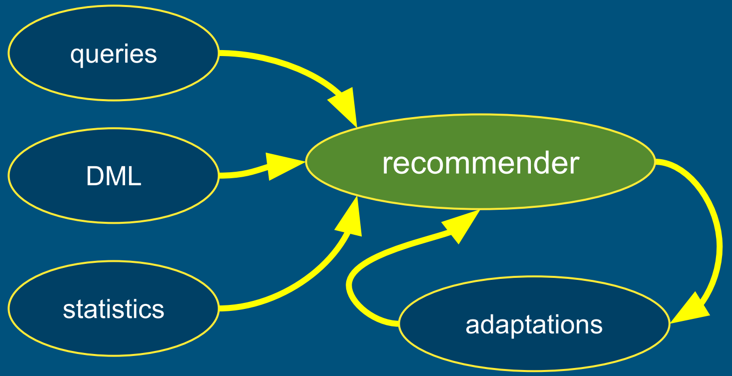
How do data systems learn?

Goals

- Improve response time, throughput, storage cost
- Predictable, adaptive (short and long term), allow human intervention

How?

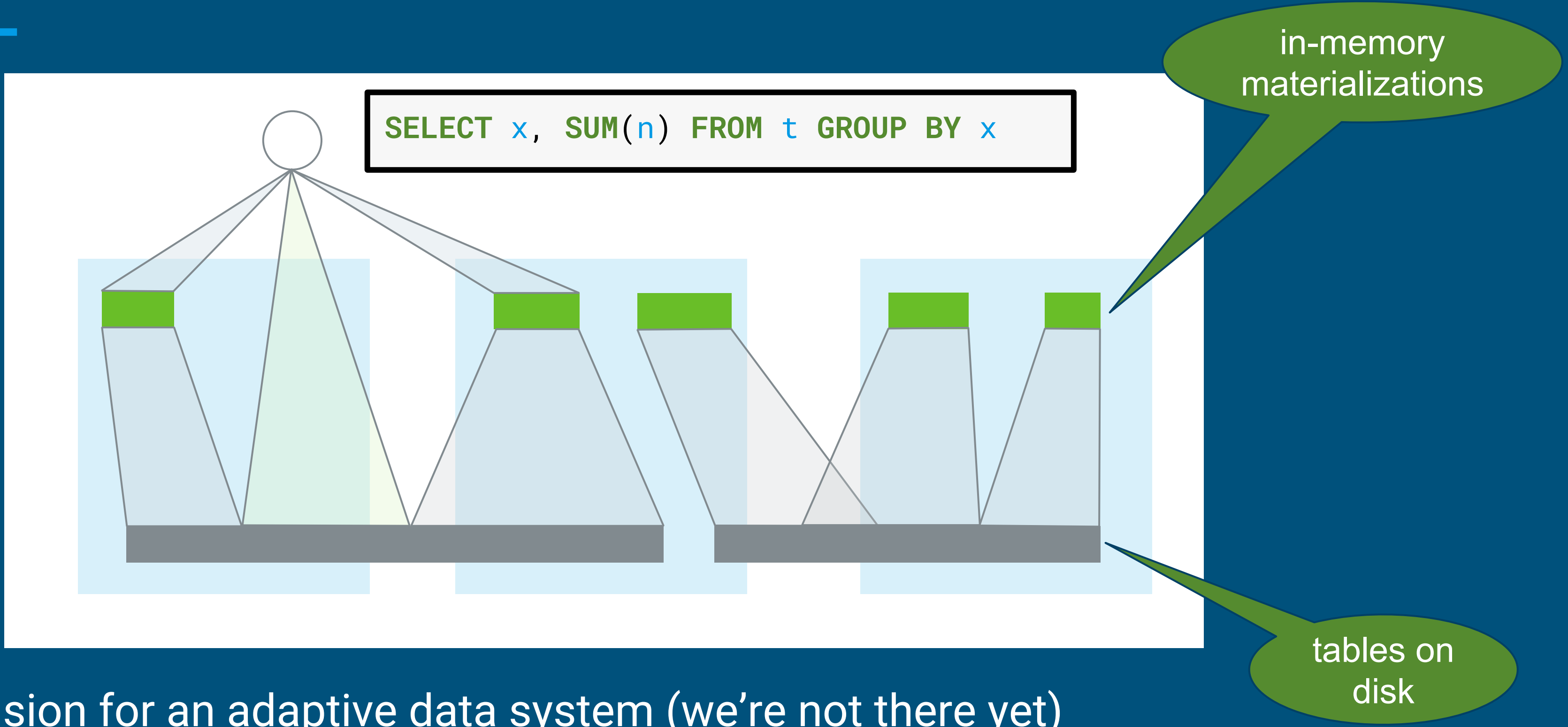
- Humans
- Adaptive systems
- Smart algorithms



Example adaptations

- Cache disk blocks in memory
- Cached query results
- Data organization, e.g. partition on a different key
- Secondary structures, e.g. b-tree and r-tree indexes

Tiled, in-memory materialized views



A vision for an adaptive data system (we're not there yet)

Building materialized views

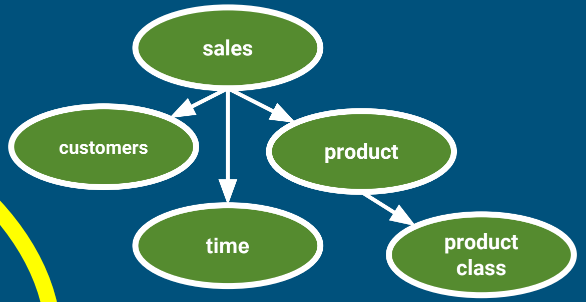
Challenges:

- **Design** Which materializations to create?
- **Populate** Load them with data
- **Maintain** Incrementally populate when data changes
- **Rewrite** Transparently rewrite queries to use materializations
- **Adapt** Design and populate new materializations, drop unused ones
- **Express** Need a rich algebra, to model how data is derived

Initial focus: summary tables (materialized views over star schemas)

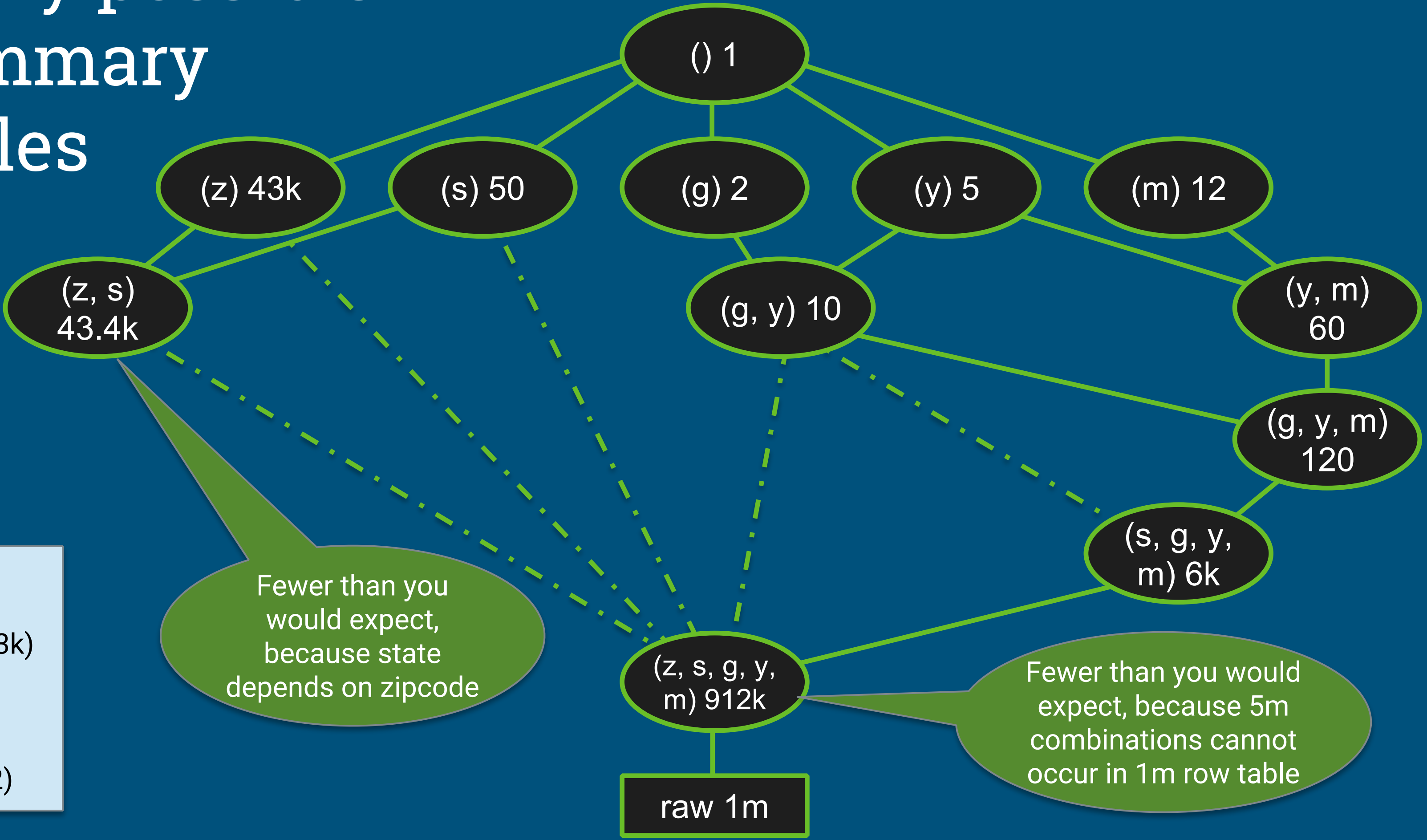
Designing summary tables via lattices

```
CREATE MATERIALIZED VIEW SalesYearZipcode AS
SELECT t.year, c.state, c.zipcode,
  COUNT(*), SUM(units)
FROM Sales AS s
JOIN Time AS t USING (timeId)
JOIN Customers AS c USING (customerId)
GROUP BY 1, 2, 3;
```




```
CREATE LATTICE Sales AS
SELECT t.*, c.*, COUNT(*), SUM(s.units)
FROM Sales AS s
JOIN Time AS t USING (timeId)
JOIN Customers AS c USING (customerId)
JOIN Products AS p USING (productId);
```

Many possible summary tables



Key

z zipcode (43k)
s state (50)
g gender (2)
y year (5)
m month (12)

Algorithm: Design summary tables

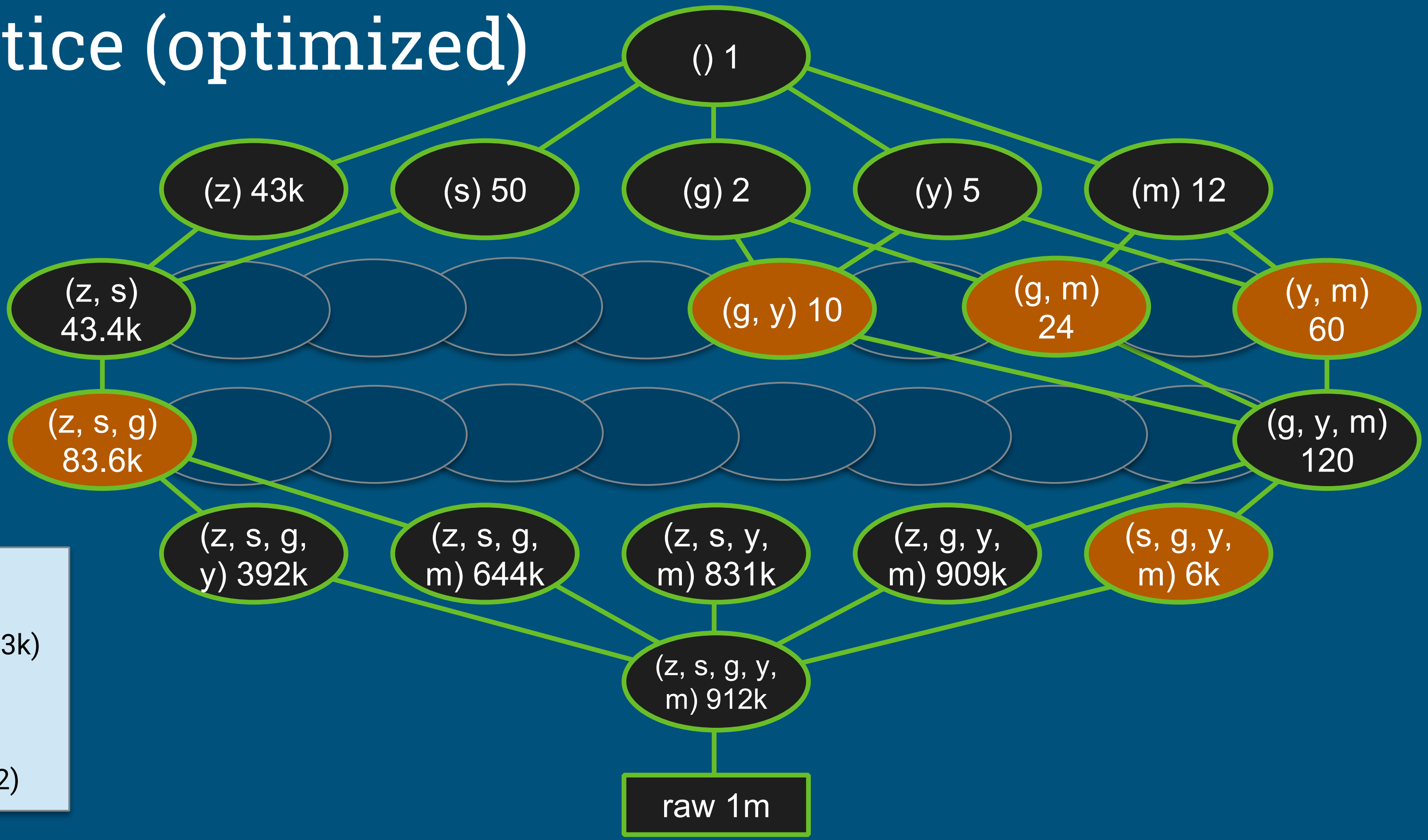
Given a database with 30 columns, 10M rows. Find X summary tables with under Y rows that improve query response time the most.

AdaptiveMonteCarlo algorithm [1]:

- Based on research [2]
- Greedy algorithm that takes a combination of summary tables and tries to find the table that yields the greatest cost/benefit improvement
- Models “benefit” of the table as query time saved over simulated query load
- The “cost” of a table is its size

[1] org.pentaho.aggdes.algorithm.impl.AdaptiveMonteCarloAlgorithm
[2] Harinarayan, Rajaraman, Ullman (1996). “Implementing data cubes efficiently”

Lattice (optimized)



Key

z zipcode (43k)
s state (50)
g gender (2)
y year (5)
m month (12)

Data profiling

Algorithm needs `count(distinct a, b, ...)` for each combination of attributes:

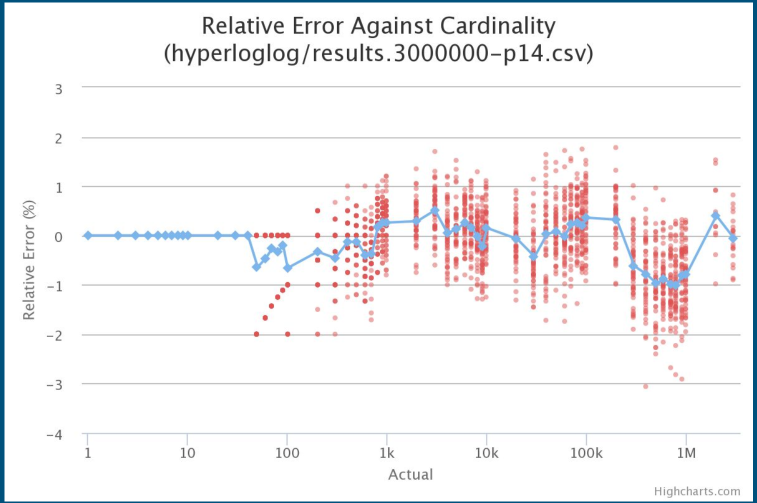
- Previous example had $2^5 = 32$ possible tables
- Schema with 30 attributes has 2^{30} (about 10^9) possible tables
- Algorithm considers a significant fraction of these
- Approximations are OK

Attempts to solve the profiling problem:

1. Compute each combination: scan, sort, unique, count; repeat 2^{30} times!
2. Sketches (HyperLogLog)
3. Sketches + parallelism + information theory [CALCITE-1616]

Sketches

HyperLogLog is an algorithm that computes approximate distinct count. It can estimate cardinalities of 10^9 with a typical error rate of 2%, using 1.5 kB of memory. [3][4]



With 16 MB memory per machine we can compute 10,000 combinations of attributes each pass.

So, we're down from 10^9 to 10^5 passes.

[3] Flajolet, Fusy, Gandouet, Meunier (2007). "Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm"
[4] https://github.com/mrjgreen/HyperLogLog

Combining probability & information theory

Given	Expected cardinality	Actual cardinality	Surprise
(gender): 2 (state): 50	(gender, state): 100.0	100	0.000
(month): 12 (zipcode): 43,000	(month, zipcode): 441,699.3	442,700	0.001
(state): 50 (zipcode): 43,000	(state, zipcode): 799,666.7	43,400	0.897
(state, zipcode): 43,400 (gender, state): 100 (gender, zipcode): 85,995	(gender, state, zipcode): 86,799 = min(86,799, 892,234, 892,228)	83,567	0.019

- Surprise = abs(actual - expected) / (actual + expected)
- E(card (x, y)) = n . (1 - ((n - 1) / n) ^ p)   n = card (x) * card (y), p = row count

Algorithm

Three ways "surprise" can help:

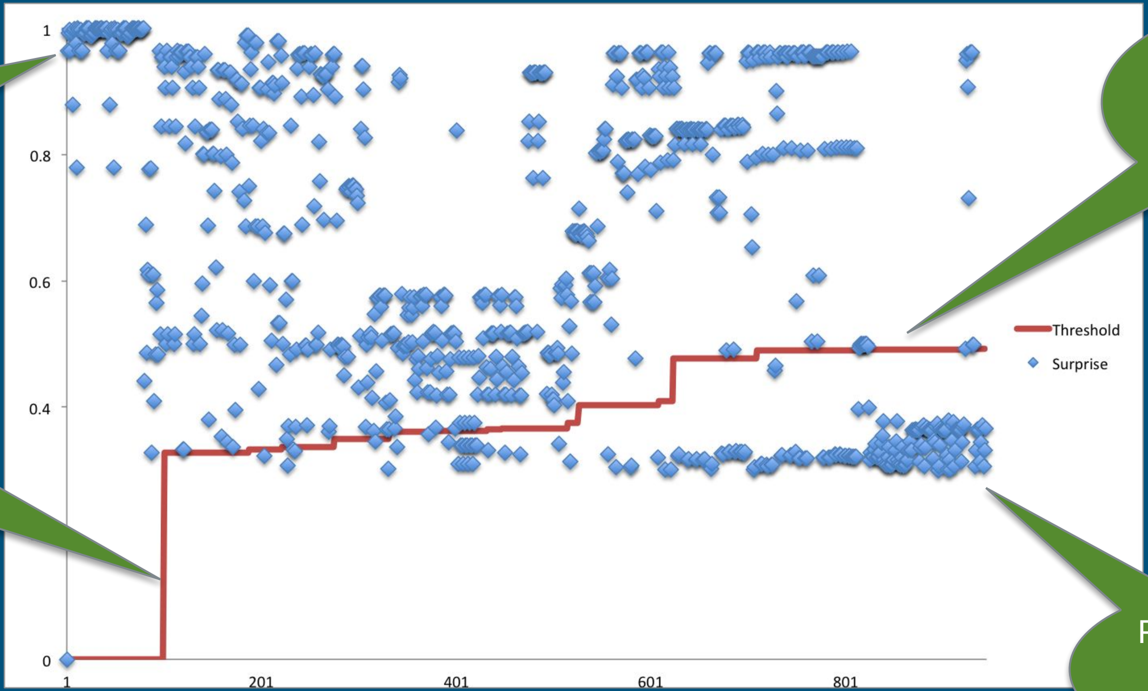
- If a cardinality is not surprising, we don't need to store it -- we can derive it
- If a combination's cardinality is not surprising, it is unlikely to have surprising children
- If we're not seeing surprising results, it's time to stop

```
surprise_threshold := 1
queue := {singleton combinations} // (a), (b), ...
while queue is not empty {
  batch := remove first 10,000 entries in queue
  compute cardinality of each combination in batch
  for each actual (computed) cardinality a {
    e := expected cardinality of combination
    s := surprise(a, e)
    if s > surprise_threshold {
      store combination and its cardinality
      add child combinations to queue // (x, a), (x, b), ...
    }
    increase surprise_threshold
  }
}
```

Algorithm progress and “surprise” threshold

Singleton combinations are have surprise = 1

Surprise threshold rises after we have completed the first batch



Surprise threshold rises as algorithm progresses

Rejected as not sufficiently surprising

Progress of algorithm

Data profiling - summary

The algorithm defeats a combinatorial search space using sketches + information theory + parallelism

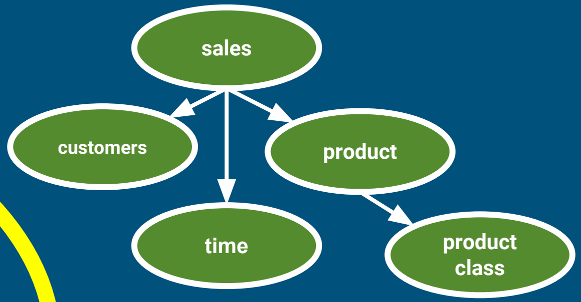
Recommending data structures is an optimization problem; profiling provides the cost & benefit function

As a by-product, the algorithm discovers unique keys, “almost” keys, and foreign keys

But which tables are actually joined together in practice?

Designing summary tables via lattices (2)

```sql
CREATE MATERIALIZED VIEW SalesYearZipcode AS
SELECT t.year, c.state, c.zipcode,
  COUNT(*), SUM(units)
FROM Sales AS s
JOIN Time AS t USING (timeId)
JOIN Customers AS c USING (customerId)
GROUP BY 1, 2, 3;
```

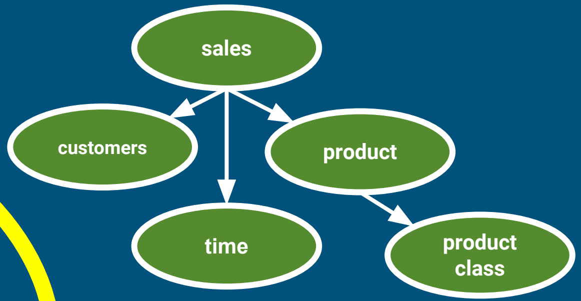


The lattice generates the summary tables. But who writes the lattice?

```sql
CREATE LATTICE Sales AS
SELECT t.*, c.*, COUNT(*), SUM(s.units)
FROM Sales AS s
JOIN Time AS t USING (timeId)
JOIN Customers AS c USING (customerId)
JOIN Products AS p USING (productId);
```

Designing summary tables via lattices (3)

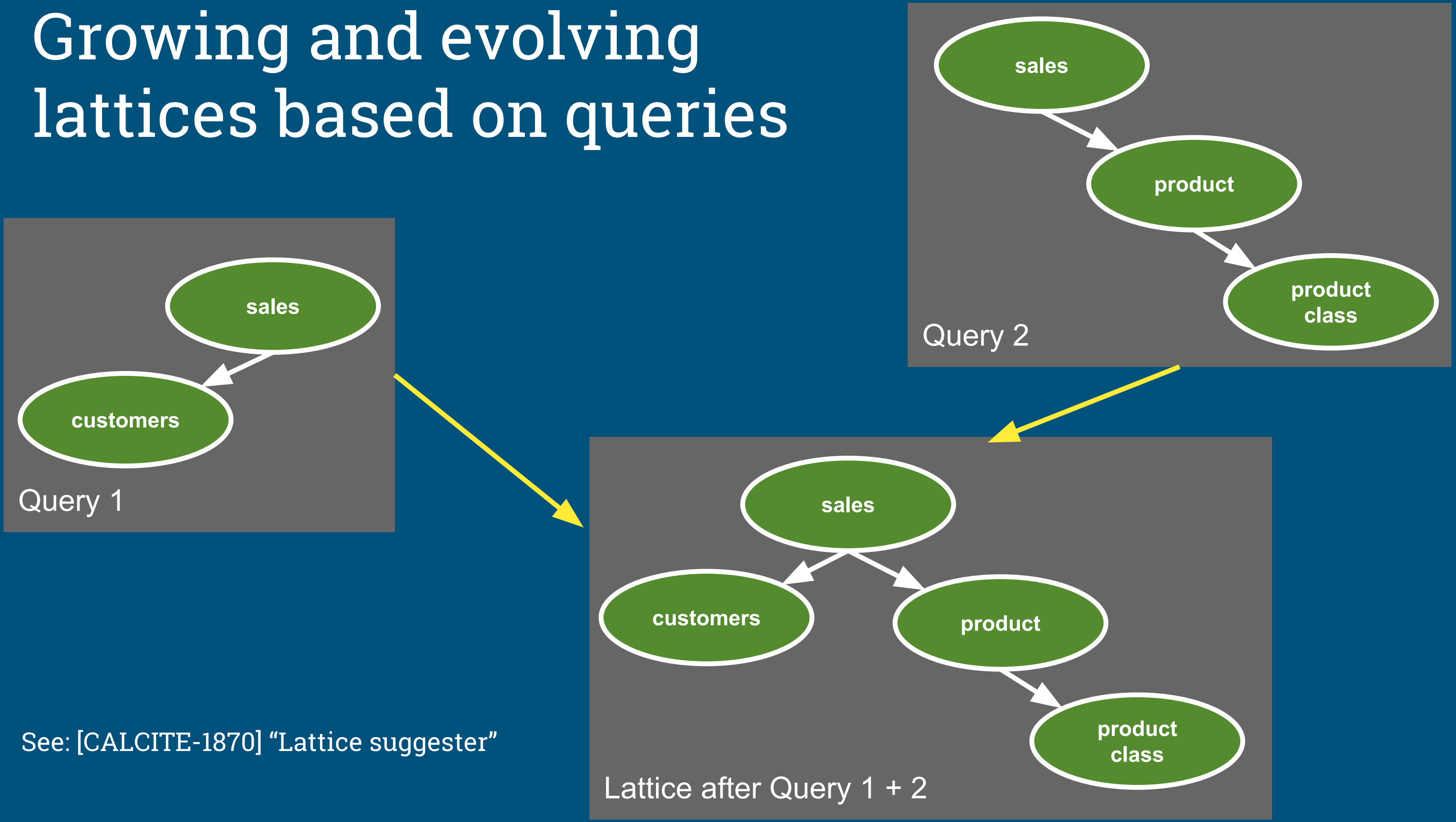
```
CREATE MATERIALIZED VIEW SalesYearZipcode AS
SELECT t.year, c.state, c.zipcode,
  COUNT(*), SUM(units)
FROM Sales AS s
JOIN Time AS t USING (timeId)
JOIN Customers AS c USING (customerId)
GROUP BY 1, 2, 3;
```




```
CREATE LATTICE Sales AS
SELECT t.*, c.*, COUNT(*), SUM(s.units)
FROM Sales AS s
JOIN Time AS t USING (timeId)
JOIN Customers AS c USING (customerId)
JOIN Products AS p USING (productId);
```

```
ALTER SCHEMA Sales
INFER LATTICES;
```

Growing and evolving lattices based on queries



See: [CALCITE-1870] “Lattice suggester”

Summary

Learning systems = manual tuning + adaptive + smart algorithms

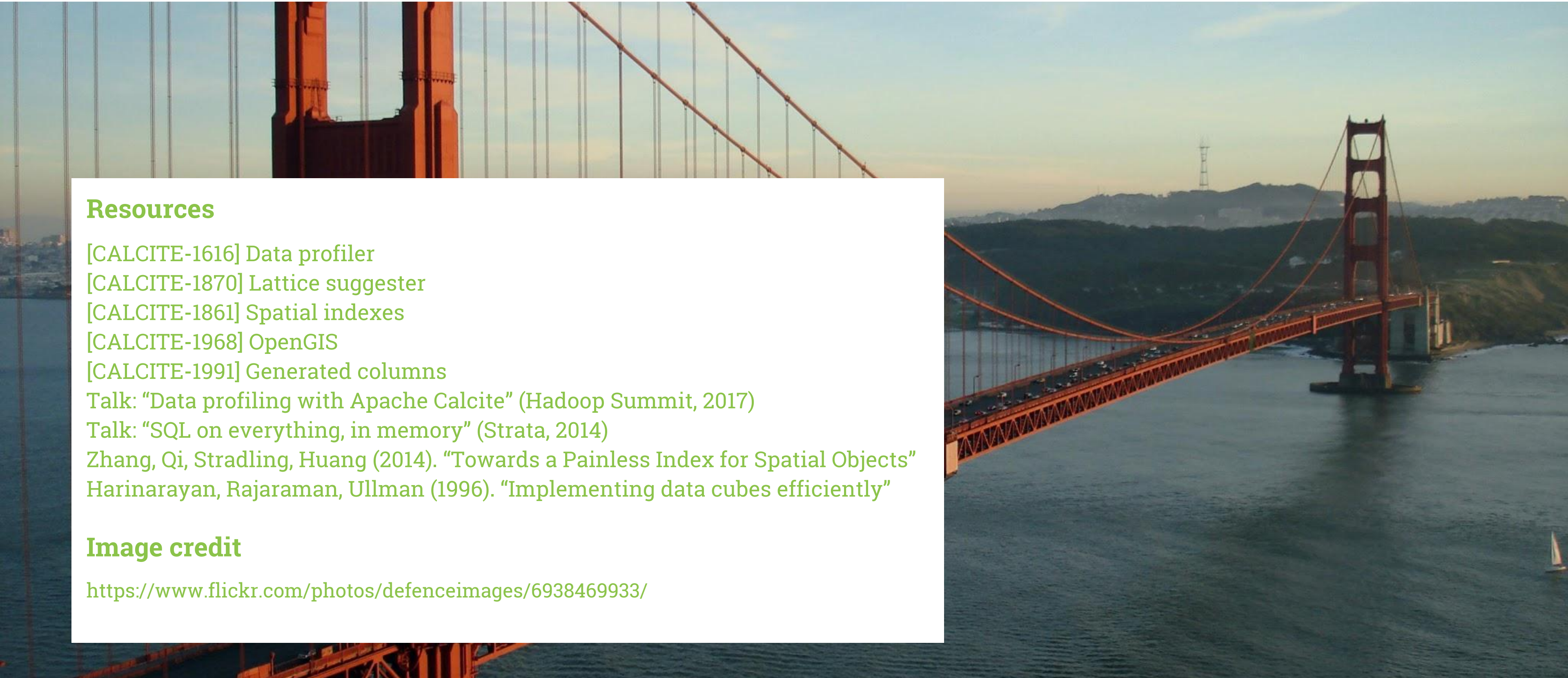
Query history + data profiling→ lattices → summary tables

We have discussed summary tables (materialized views based on join/aggregate in a star schema) but the approach can be applied to other kinds of materialized views

Relational algebra, incorporating materialized views, is a powerful language that allows us to combine many forms of data optimization

Thank you! Questions?

@julianhyde | @ApacheCalcite | http://apache.calcite.org



Resources

[CALCITE-1616] Data profiler
[CALCITE-1870] Lattice suggester
[CALCITE-1861] Spatial indexes
[CALCITE-1968] OpenGIS
[CALCITE-1991] Generated columns
Talk: “Data profiling with Apache Calcite” (Hadoop Summit, 2017)
Talk: “SQL on everything, in memory” (Strata, 2014)
Zhang, Qi, Stradling, Huang (2014). “Towards a Painless Index for Spatial Objects”
Harinarayan, Rajaraman, Ullman (1996). “Implementing data cubes efficiently”

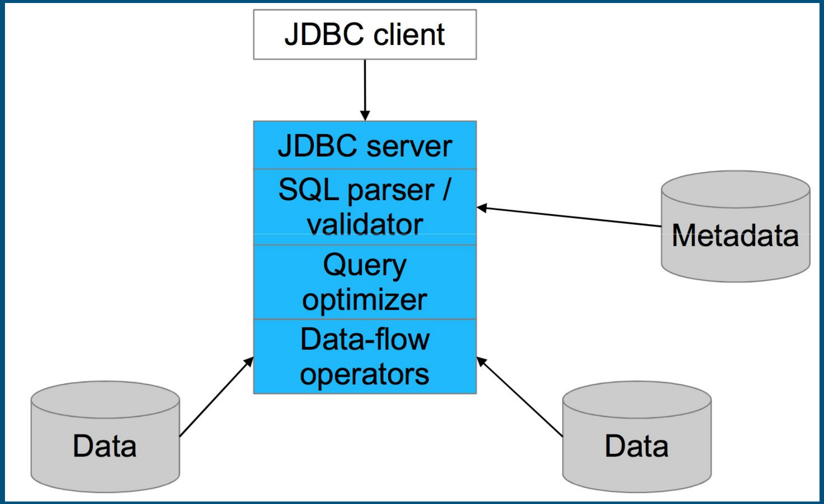
Image credit

https://www.flickr.com/photos/defenceimages/6938469933/

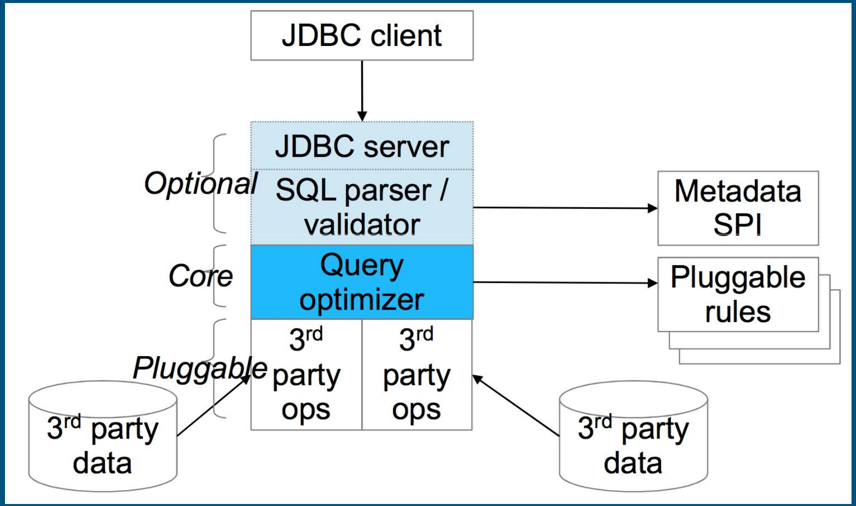
Extra slides

Architecture

Conventional database

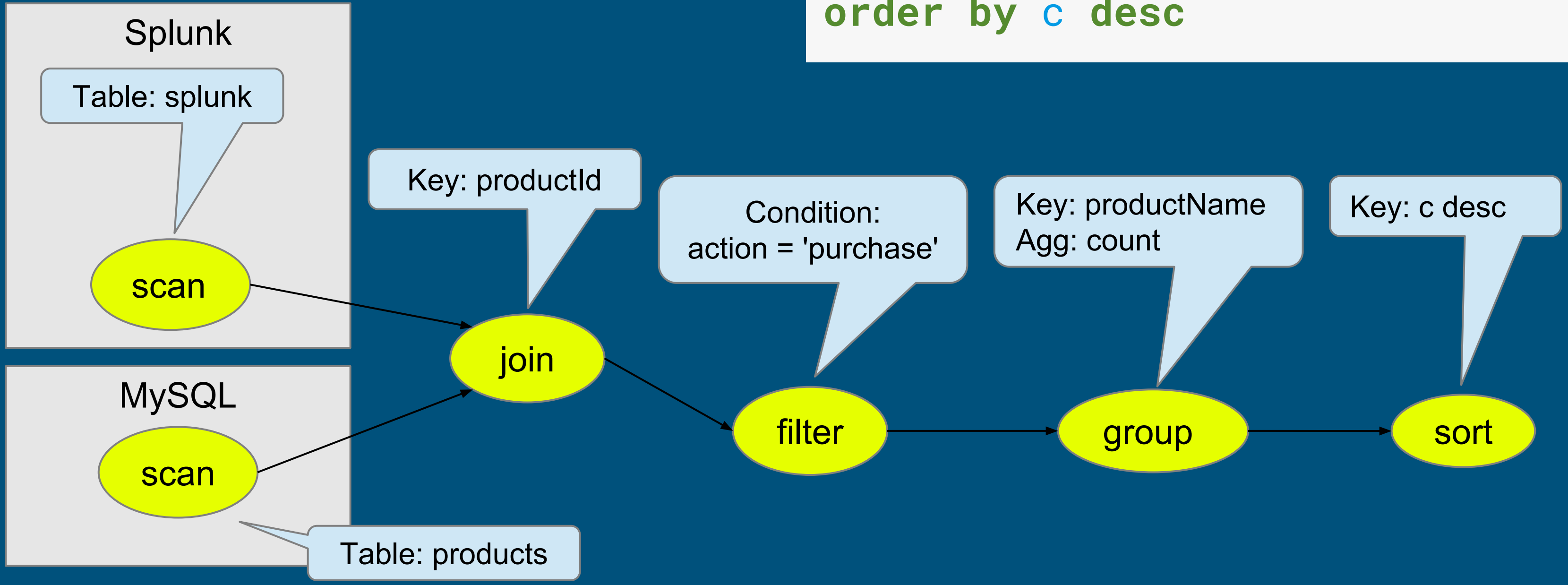


Calcite



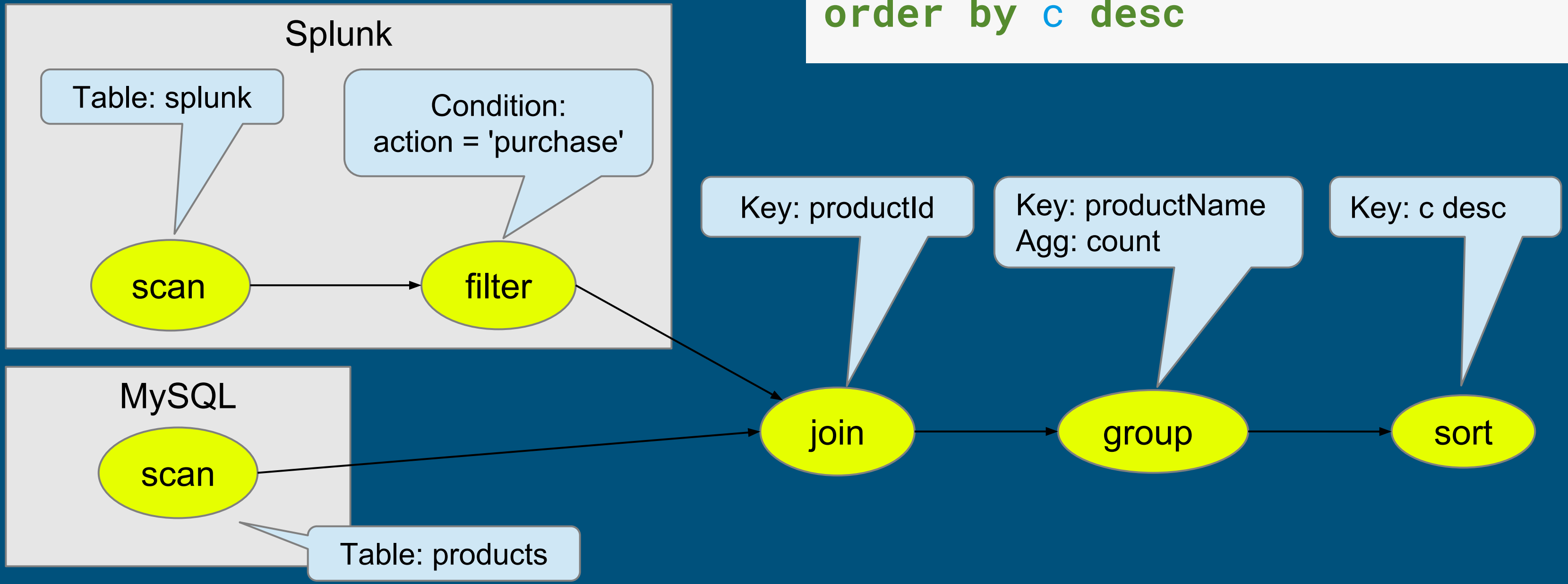
Planning queries

```
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
```



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

Materialized views, lattices, tiles

Materialized view - A table whose contents are guaranteed to be the same as executing a given query.

Lattice - Recommends, builds, and recognizes summary materialized views (tiles) based on a star schema.

A query defines the tables and many:1 relationships in the star schema.

Tile - A summary materialized view that belongs to a lattice. A tile may or may not be materialized. Might be:

- Declared in lattice, or
- Generated via recommender algorithm, or
- Created in response to query.

```
CREATE MATERIALIZED VIEW t AS
SELECT * FROM emps
WHERE deptno = 10;
```

```
CREATE LATTICE star AS
SELECT *
FROM sales_fact_1997 AS s
JOIN product AS p ON …
JOIN product_class AS pc ON …
JOIN customer AS c ON …
JOIN time_by_day AS t ON …;
```

```
CREATE MATERIALIZED VIEW zg IN star
SELECT gender, zipcode,  COUNT(*),
 SUM(unit_sales) FROM star
GROUP BY gender, zipcode;
```

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

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;
```

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 to data profiling

- 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)