SPATIAL QUERY ON VANILLA DATABASES

Julian Hyde (Calcite PMC)

Spatial Query on Vanilla Databases

Spatial and GIS applications have traditionally required specialized databases, or at least specialized data structures like r-trees. Unfortunately this means that hybrid applications such as spatial analytics are not well served, and many people are unaware of the power of spatial queries because their favorite database does not support them.

In this talk, we describe how Apache Calcite enables efficient spatial queries using generic data structures such as HBase’s key-sorted tables, using techniques like Hilbert space-filling curves and materialized views. Calcite implements much of the OpenGIS function set and recognizes query patterns that can be rewritten to use particular spatial indexes. Calcite is bringing spatial query to the masses!

@julianhyde

SQL
Query planning
Query federation
BI & OLAP
Streaming
Hadoop

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

looker

Apache Calcite

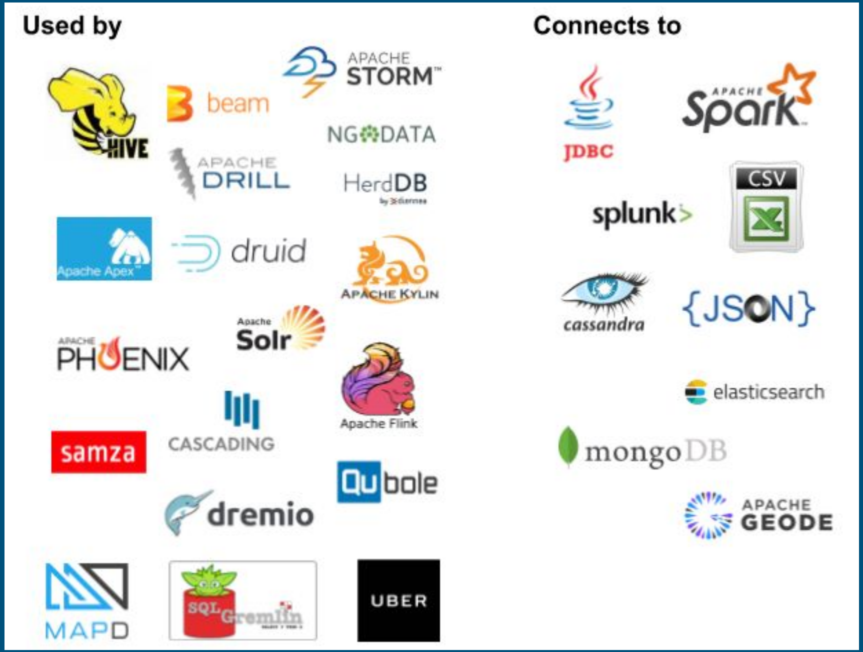
Apache top-level project since 2015

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

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



Relational algebra

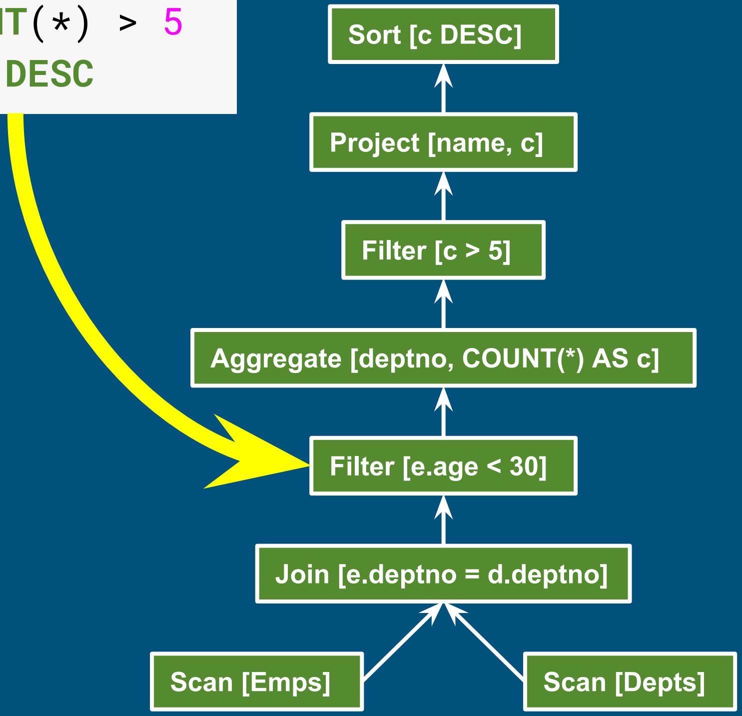
Based on set theory, plus operators: Project, Filter, Aggregate, Union, Join, Sort

Requires: declarative language (SQL), query planner

Original goal: data independence

Enables: query optimization, new algorithms and data structures

```
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
```



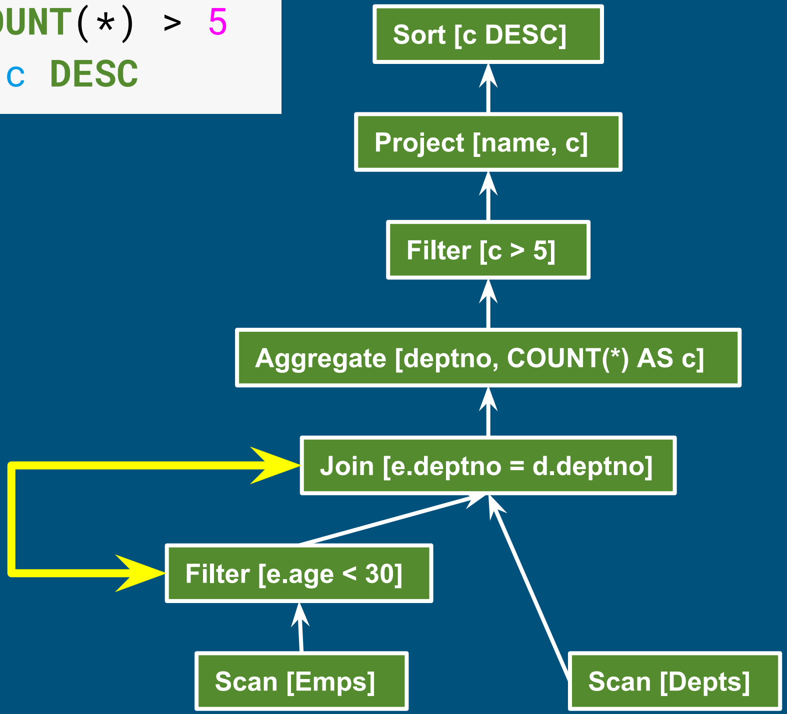
Algebraic rewrite

Optimize by applying rewrite rules that preserve semantics

Hopefully the result is less expensive; but it's OK if it's not (planner keeps "before" and "after")

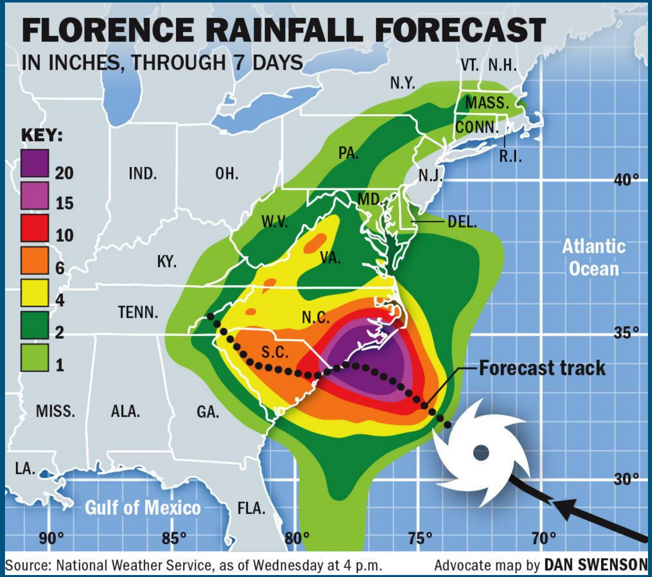
Planner uses dynamic programming, seeking the lowest total cost

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



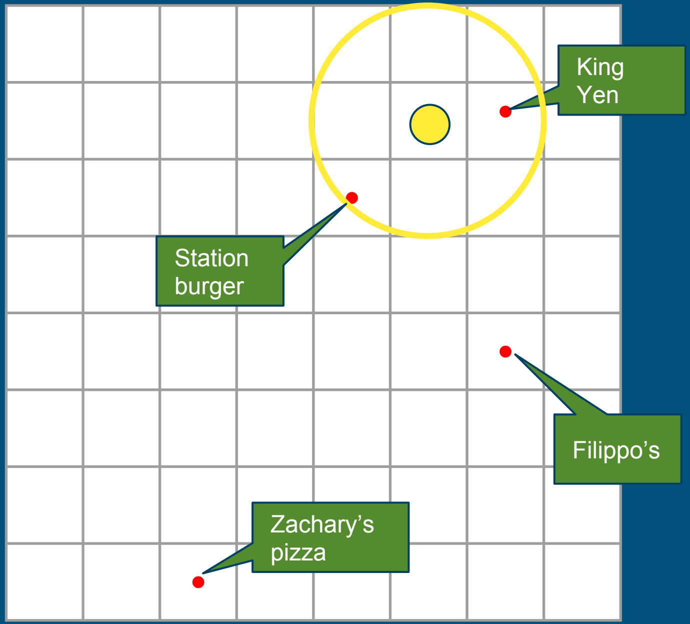
Relational

Spatial



A spatial query

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



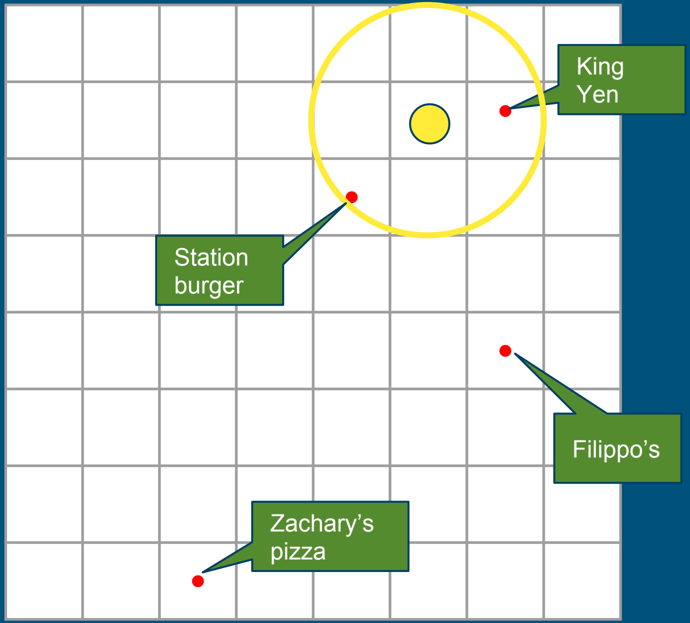
restaurant	x	y
Zachary’s pizza	3	1
King Yen	7	7
Filippo’s	7	4
Station burger	5	6

A 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

Simple implementation

Using ESRI's `geometry-api-java` library, almost all `ST_` functions were easy to implement in Calcite.

Slow – one row at a time.

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

```
package org.apache.calcite.runtime;

import com.esri.core.geometry.*;

/** Simple implementations of built-in geospatial functions. */
public class GeoFunctions {
  /** Returns the distance between g1 and g2. */
  public static double ST_Distance(Geom g1, Geom g2) {
    return GeometryEngine.distance(g1.g(), g2.g(), g1.sr());
  }

  /**  Constructs a 2D point from coordinates. */
  public static Geom ST_MakePoint(double x, double y) {
    final Geometry g = new Point(x, y);
    return new SimpleGeom(g);
  }

  /** Geometry. It may or may not have a spatial reference
  * associated with it. */
  public interface Geom {
    Geometry g();
    SpatialReference sr();
    Geom transform(int srid);
    Geom wrap(Geometry g);
  }

  static class SimpleGeom implements Geom { … }
}
```

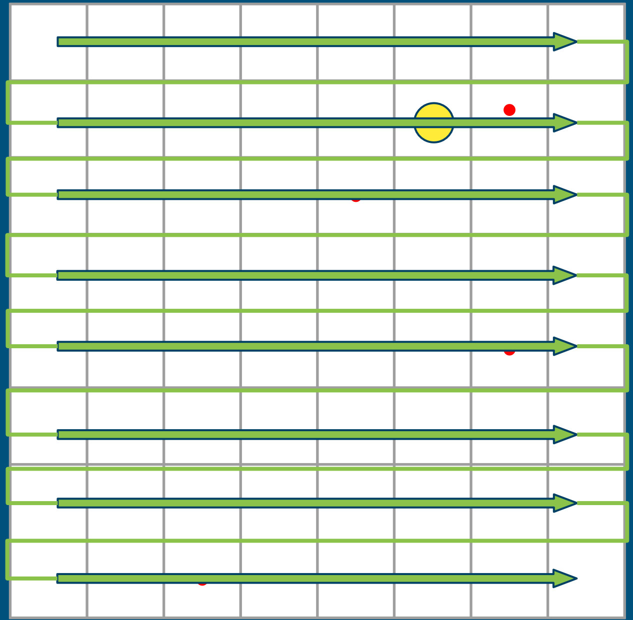
Traditional DB indexing techniques don't work

Sort

```
CREATE /* b-tree */ INDEX
  I_Restaurants
ON Restaurants(x, y);
```

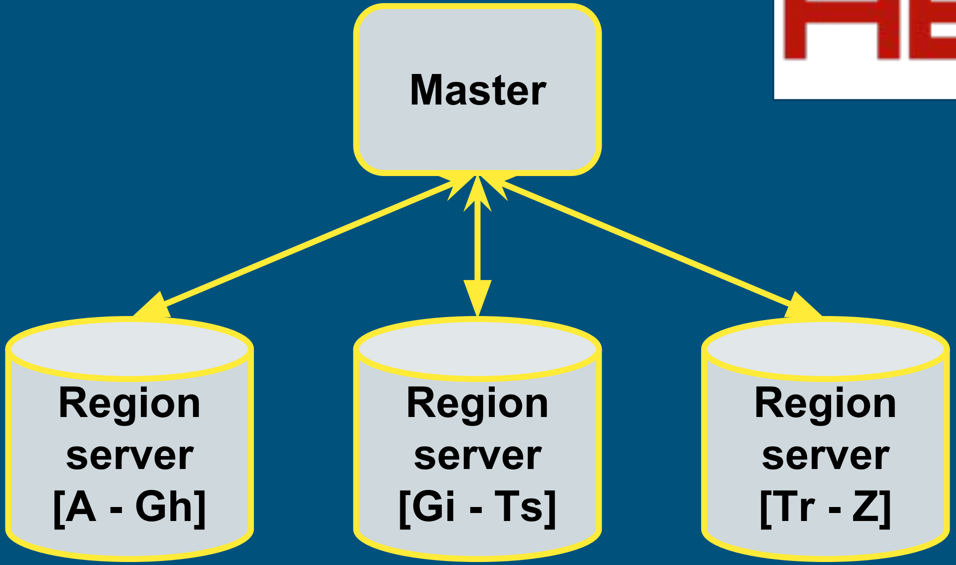
Hash

```
CREATE TABLE Restaurants(
    restaurant VARCHAR(20),
    x INTEGER,
    y INTEGER)
PARTITION BY (MOD(x + 5279 * y, 1024));
```



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

A “vanilla database”



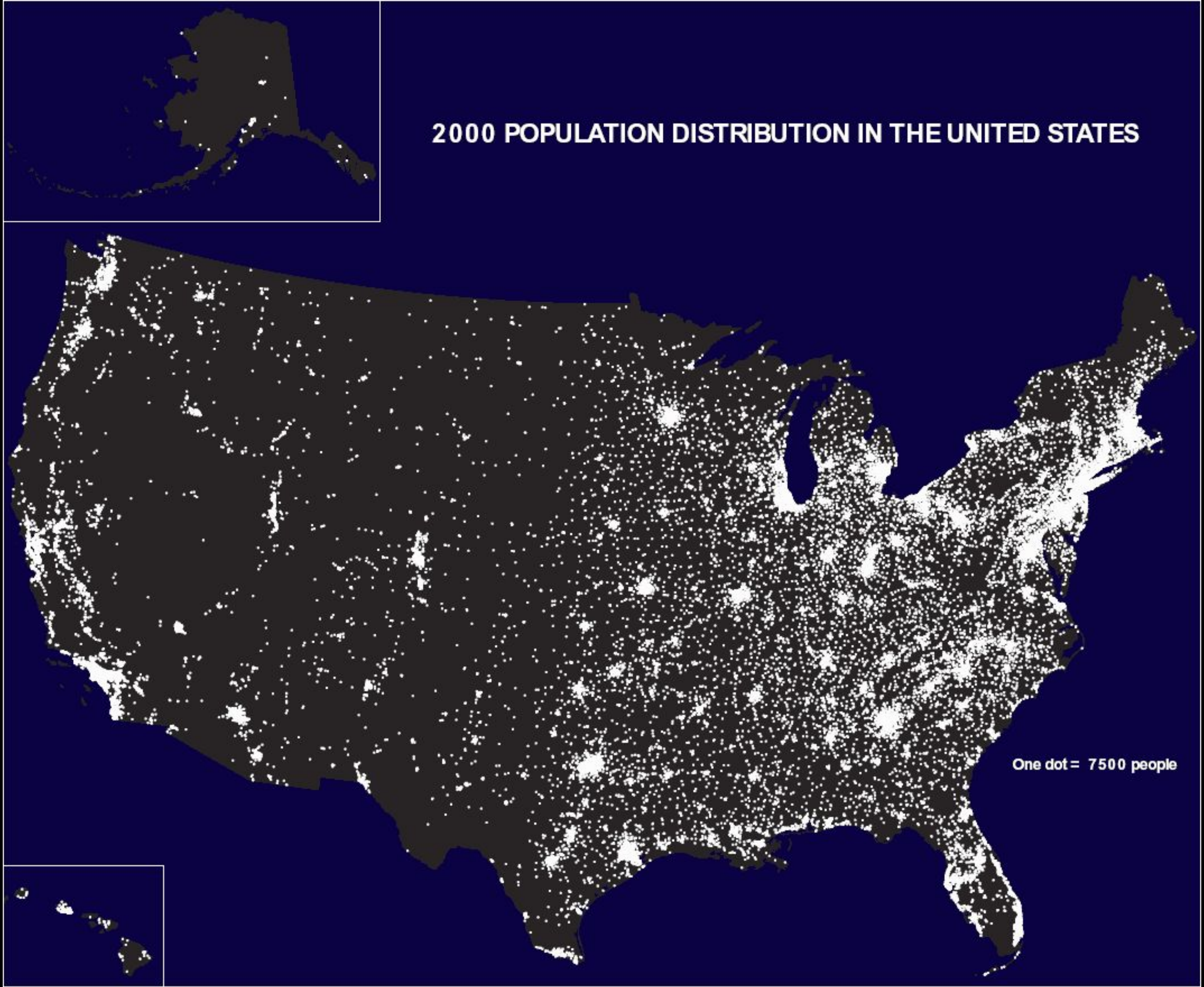
Spatial data structures and algorithms

The challenge: Reduce dimensionality while preserving locality

- **Reduce dimensionality** – We want to warp the information space so that we can access on one composite attribute rather than several
- **Preserve locality** – If two items are close in 2D, we want them to be close in the information space (and in the same cache line or disk block)

Two main approaches to spatial data structures:

- Data-oriented
- Space-oriented

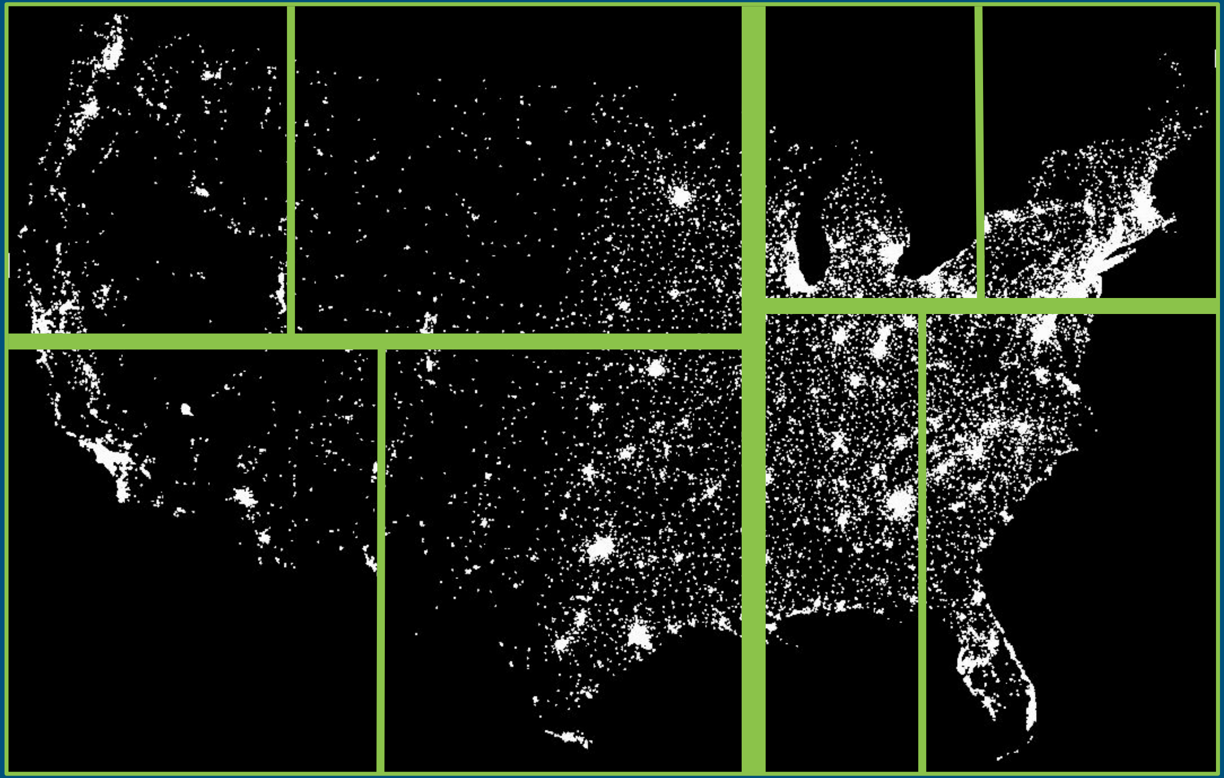

2000 POPULATION DISTRIBUTION IN THE UNITED STATES
One dot = 7500 people

R-tree (a data-oriented structure)

R-tree (split vertically into 2)

R-tree (split horizontally into 4)

R-tree (split vertically into 8)

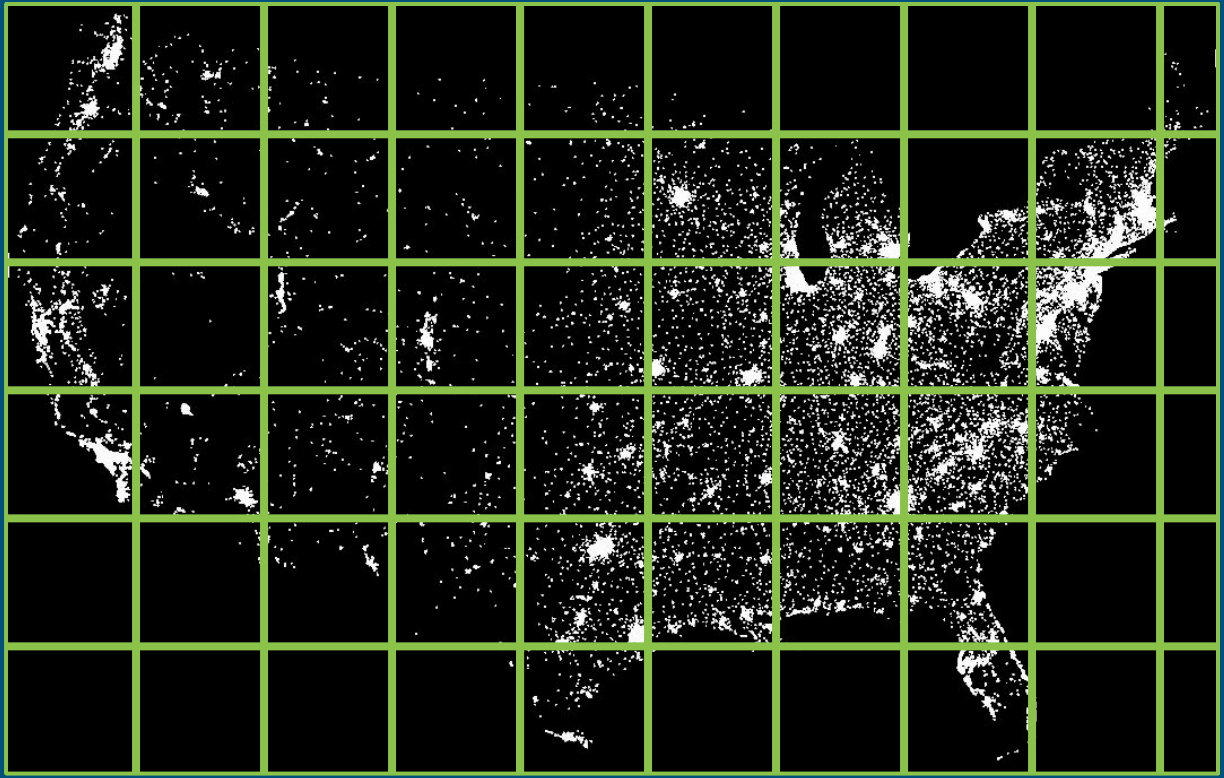


R-tree (split horizontally into 16)

R-tree (split vertically into 32)

Grid (a space-oriented structure)

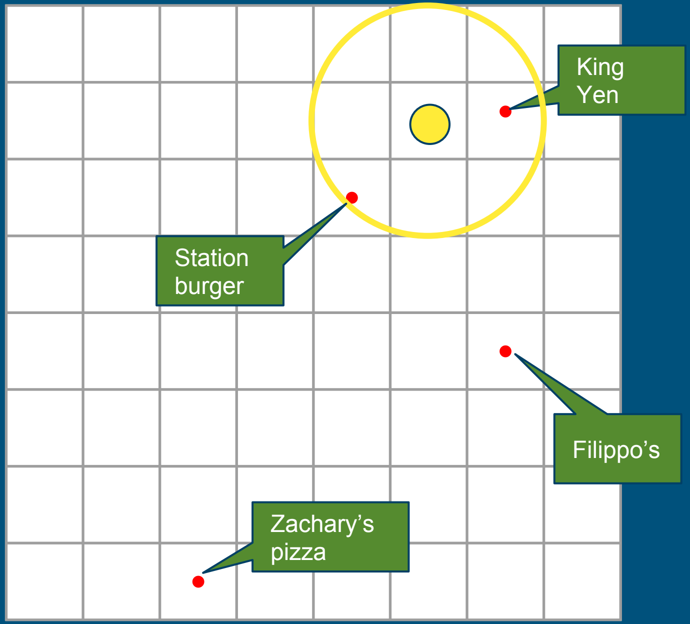
Grid (a space-oriented structure)



Spatial query

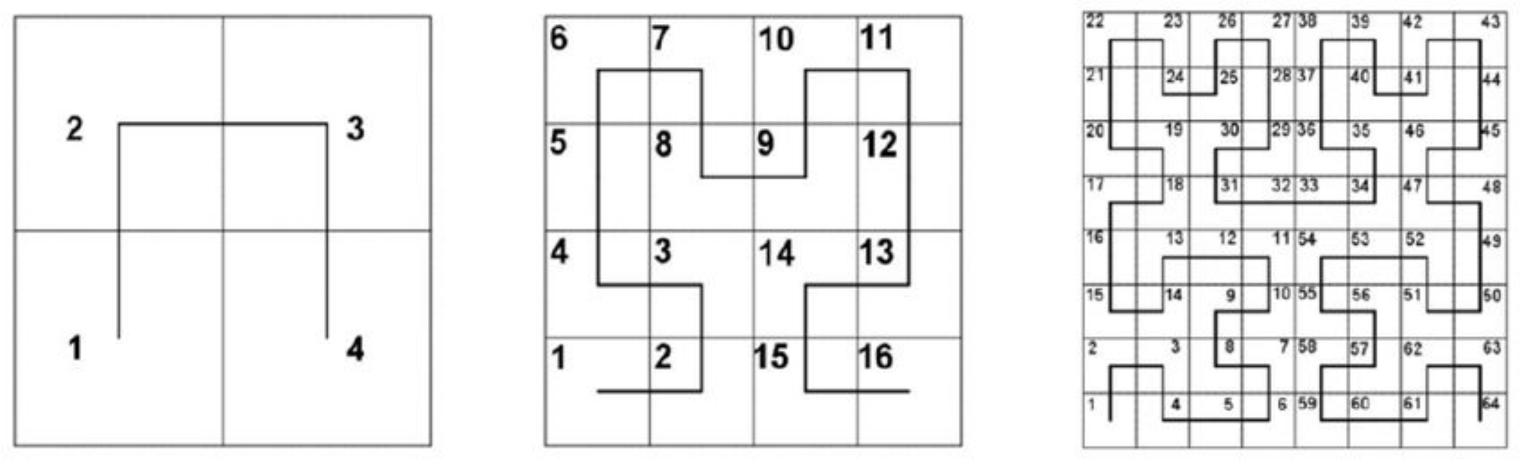
Find all restaurants within 1.5 distance units of where I am:

```
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



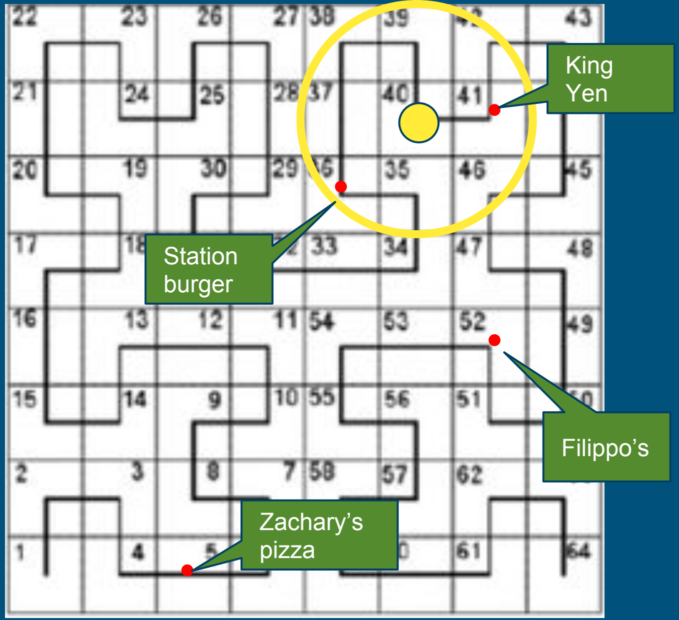
- A space-filling curve invented by mathematician David Hilbert
- Every (x, y) point has a unique position on the curve
- Points near to each other typically have Hilbert indexes close together

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

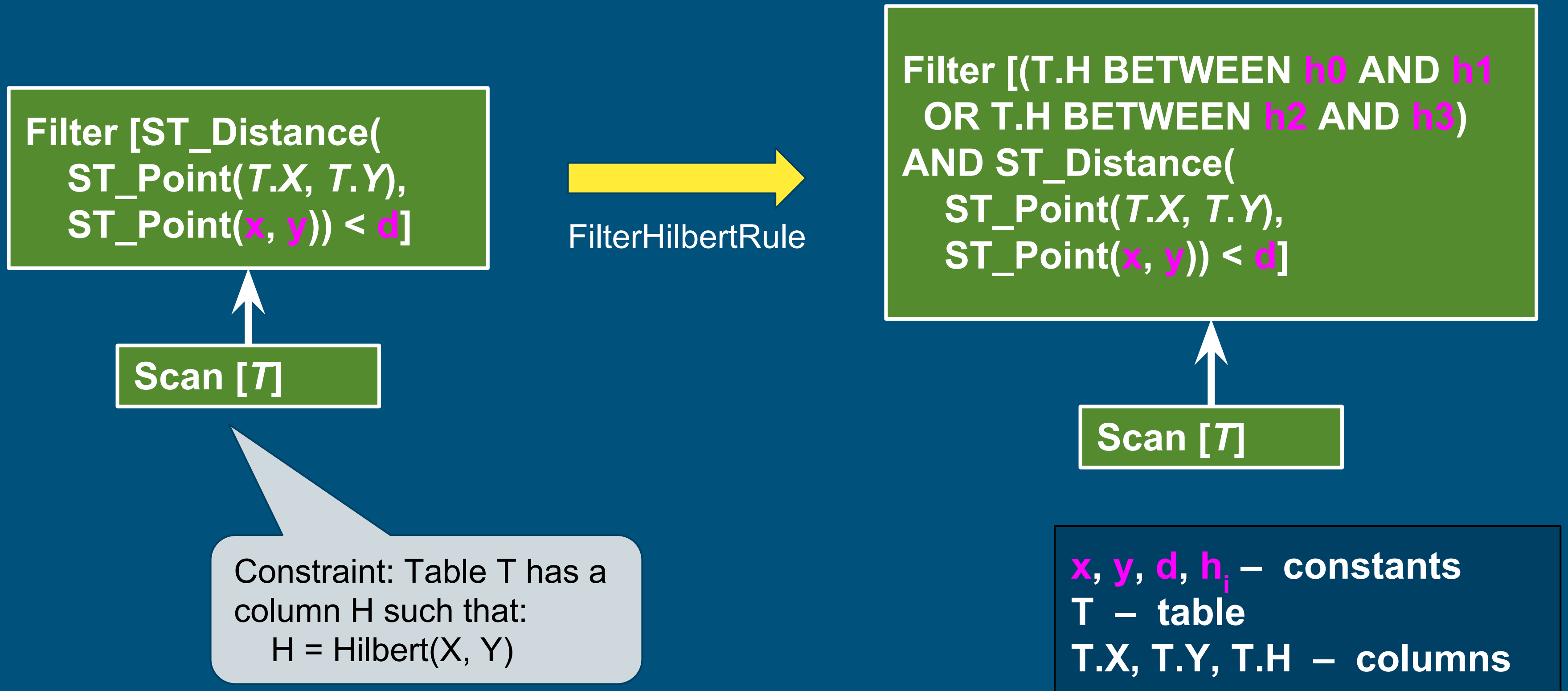
Does not require specialized spatial index such as r-tree

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

Algebraic rewrite



Variations on a theme

Several ways to say the same thing using OpenGIS functions:

- ST_Distance(ST_Point(*X*, *Y*), ST_Point(x, y)) < d
- ST_Distance(ST_Point(x, y), ST_Point(*X*, *Y*)) < d
- ST_DWithin(ST_Point(x, y), ST_Point(*X*, *Y*), d)
- ST_Contains(ST_Buffer(ST_Point(x, y), d), ST_Point(*X*, *Y*))

Other patterns can use Hilbert functions:

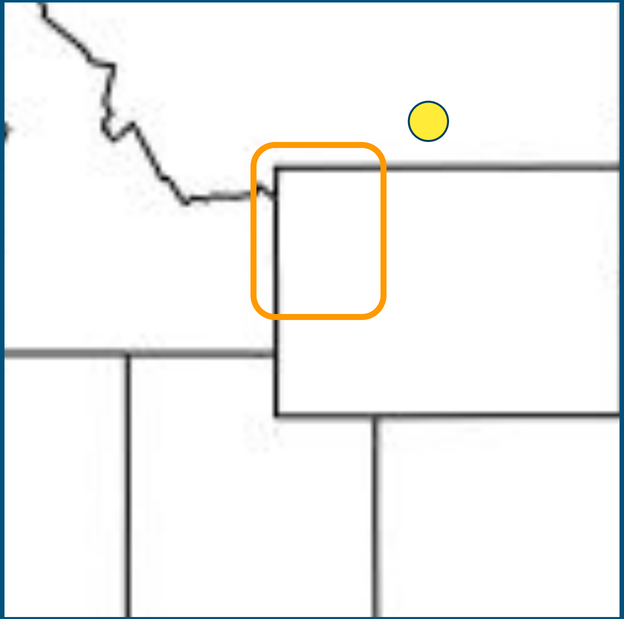
- ST_DWithin(ST_MakeLine(ST_Point(x1, y1), ST_Point(x2, y2)),
  ST_Point(*X*, *Y*), d)
- ST_Contains(ST_PolyFromText('POLYGON((0 0,20 0,20 20,0 20,0 0))'),
  ST_Point(*X*, *Y*), d)

More spatial queries

What state am I in? (1-point-to-1-polygon)

Which states does Yellowstone NP intersect? (1-polygon-to-many-polygons)

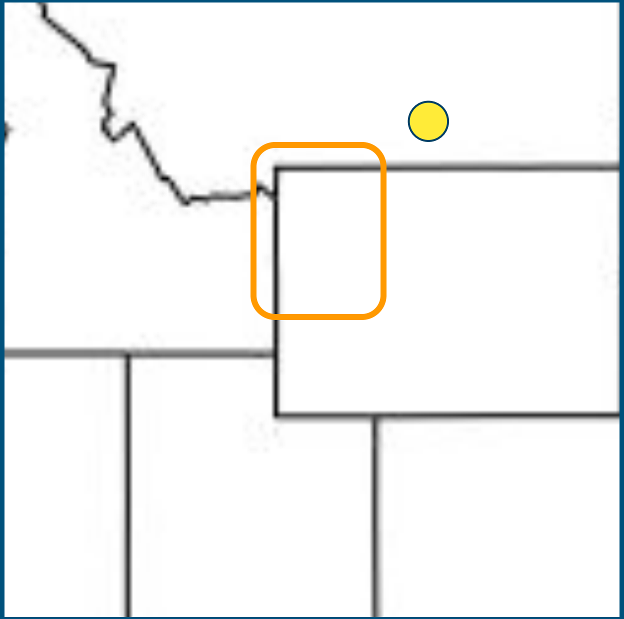
Which US national park intersects with the most states? (many-polygons-to-many-polygons, followed by sort/limit)



More spatial queries

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

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



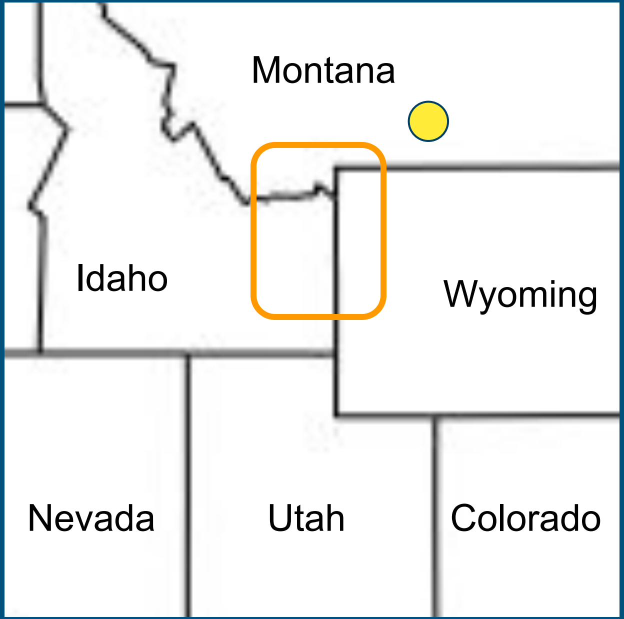
```sql
SELECT *
FROM States AS s
WHERE ST_Intersects(s.geometry,
  ST_MakePoint(6, 7))
```

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

Tile index

We cannot use space-filling curves, because each region (state or park) is a set of points and not known as planning time.

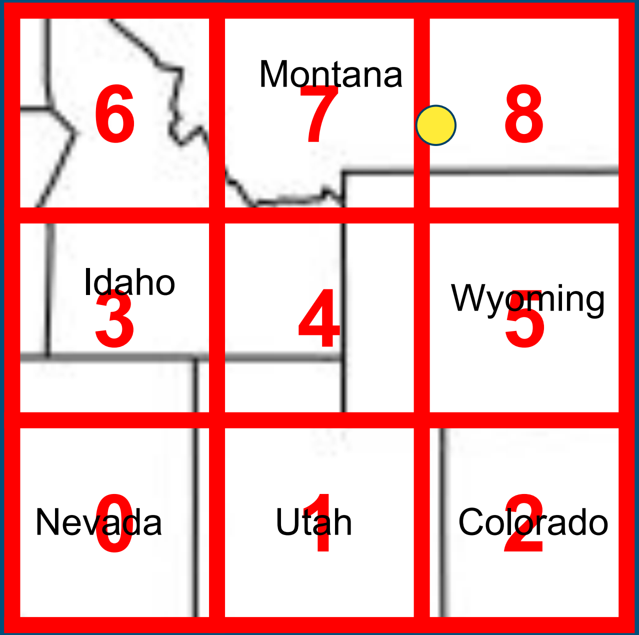
Divide regions into a (coarse) set of tiles. They intersect only if some of their tiles intersect.



Tile index

tileId	state
0	Nevada
0	Utah
1	Utah
2	Colorado
2	Utah
3	Idaho
3	Nevada
3	Utah
4	Idaho

tileId	state
4	Utah
4	Wyoming
5	Wyoming
6	Idaho
6	Montana
7	Montana
7	Wyoming
8	Montana
8	Wyoming

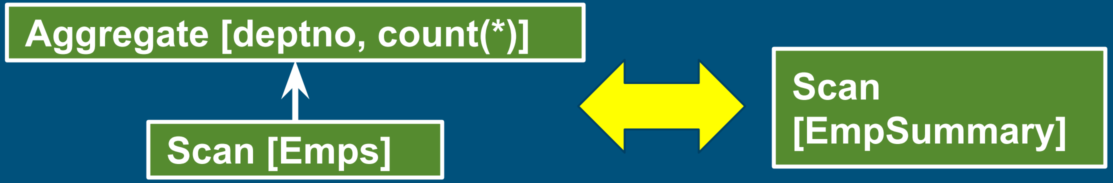


Aside: Materialized views

```
CREATE MATERIALIZED
  VIEW EmpSummary AS
SELECT deptno, COUNT(*) AS c
FROM Emp
GROUP BY deptno;
```

A materialized view is a table that is defined by a query

The planner knows about the mapping and can transparently rewrite queries to use it



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

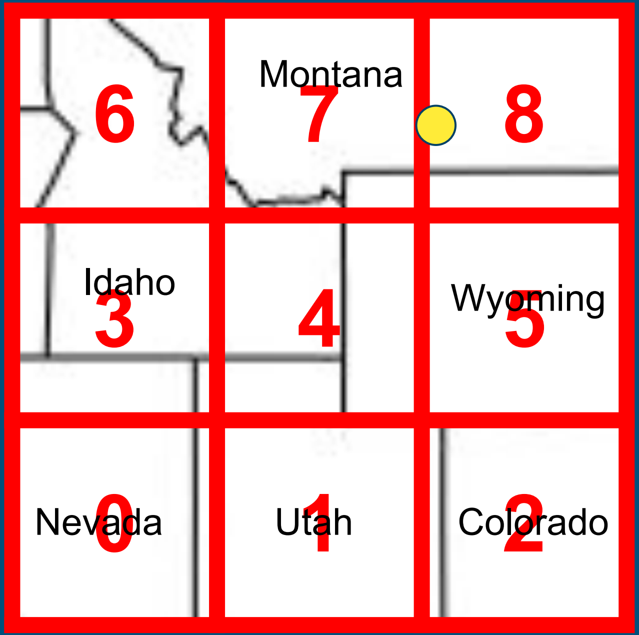
deptno	c
10	2
20	1
30	1

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

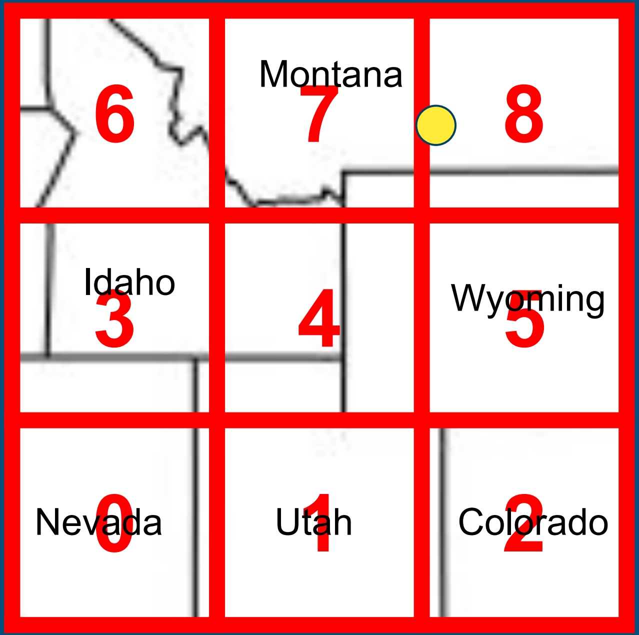



```
CREATE MATERIALIZED VIEW StateTiles AS
SELECT s.stateId, t.tileId
FROM States AS s,
  LATERAL TABLE(ST_MakeGrid(s.geometry, 4, 4)) AS t
```

Point-to-polygon query

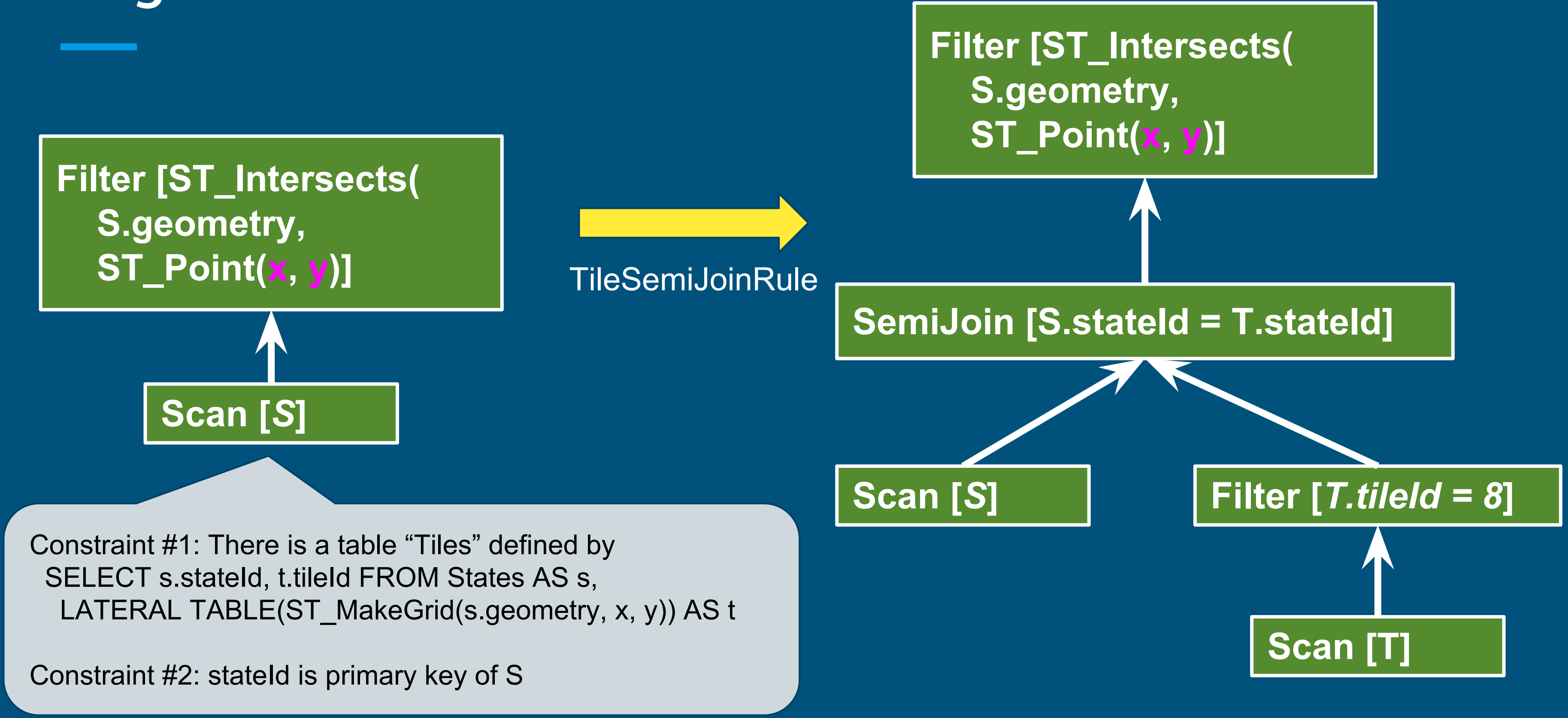
What state am I in? (point-to-polygon)

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))
```

Algebraic rewrite



Streaming + spatial

Example query: 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)
```

Summary

Traditional DB techniques (sort, hash) don’t work for 2-dimensional data

Spatial presents tough design choices:

- Space-oriented vs data-oriented algorithms
- General-purpose vs specialized data structures

Relational algebra unifies traditional and spatial:

- Use general-purpose structures
- Compose techniques (transactions, analytics, spatial, streaming)
- Must use space-oriented algorithms, because their dimensionality-reducing mapping is known at planning time

APACHECON

Thank you! Questions?

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

Resources & credits

- [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”
- https://www.census.gov/geo/maps-data/maps/2000popdistribution.html
- https://www.nasa.gov/mission_pages/NPP/news/earth-at-night.html

THE APACHE SOFTWARE FOUNDATION

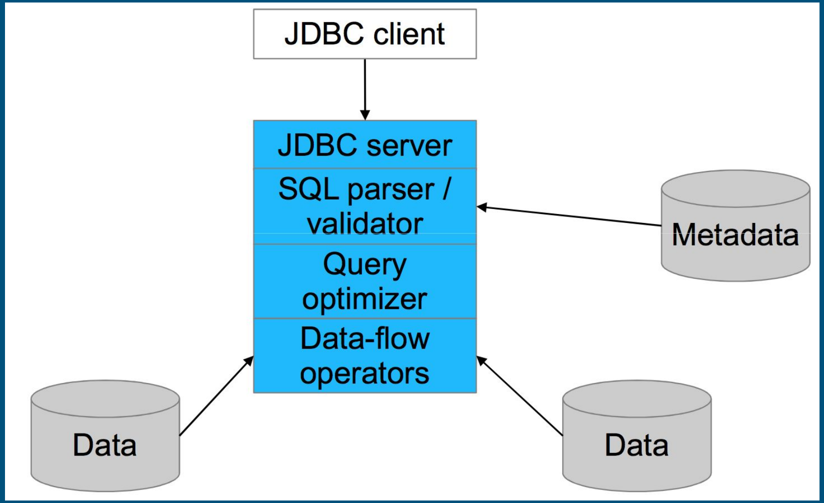
Apache Calcite

looker

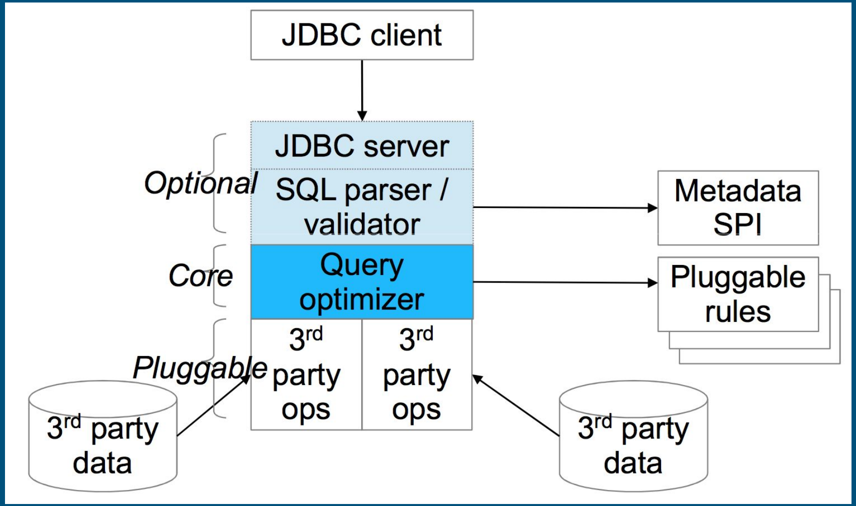
Extra slides

Architecture

Conventional database

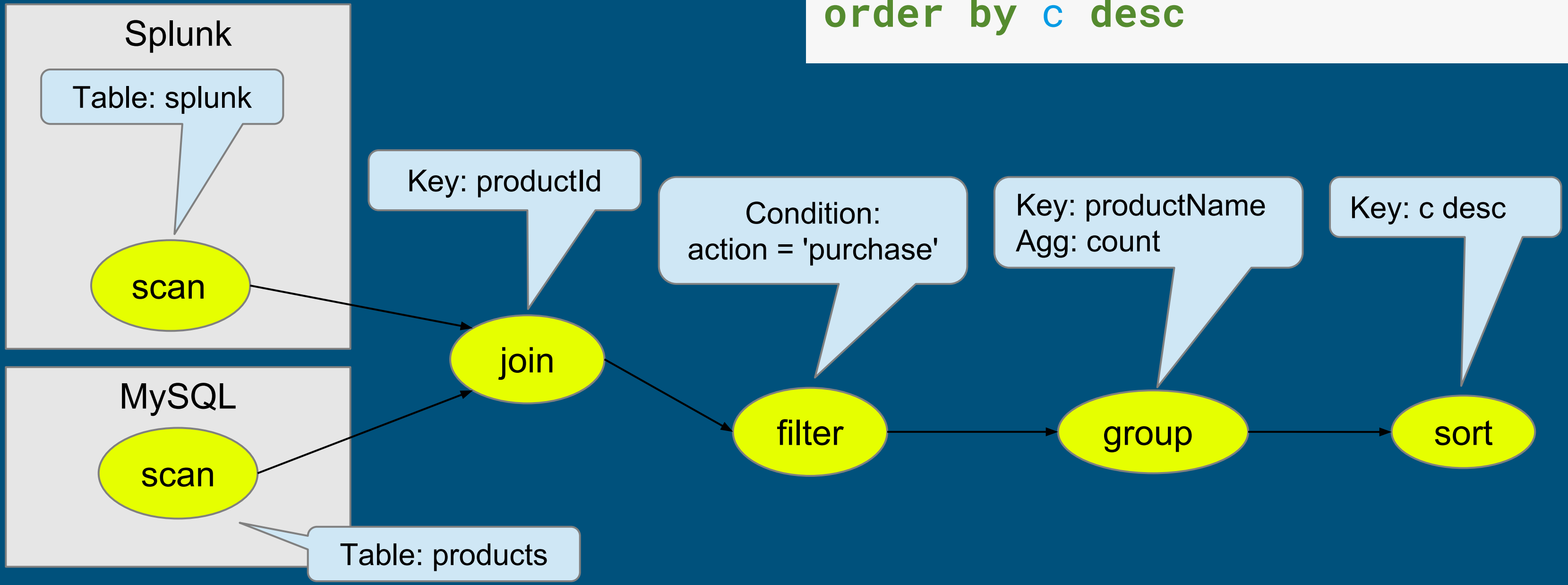


Calcite



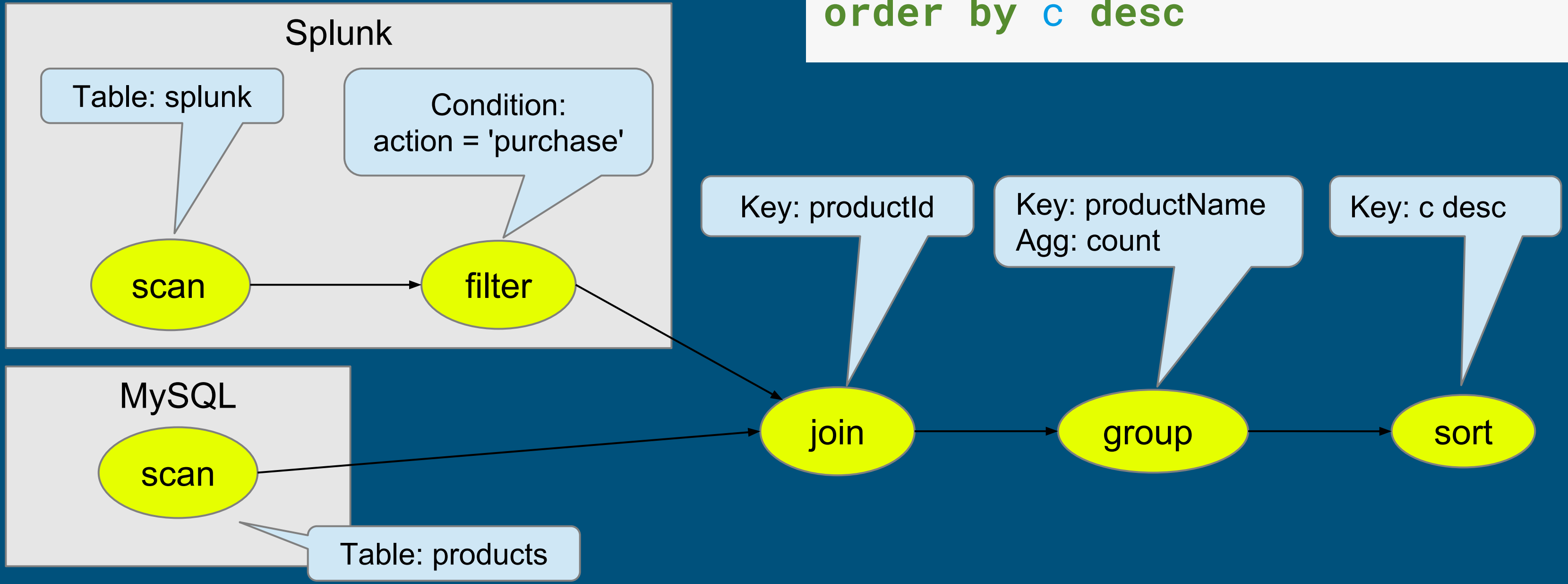
Planning queries

```
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
- RelMdColumnUniquenss
- RelMdDistinctRowCount
- RelMdSelectivity