



UNIVERSITY OF
WATERLOO

Data-Intensive Distributed Computing

CS 451/651 431/631 (Winter 2018)

Part 5: Analyzing Relational Data (2/3)

February 13, 2018

Jimmy Lin

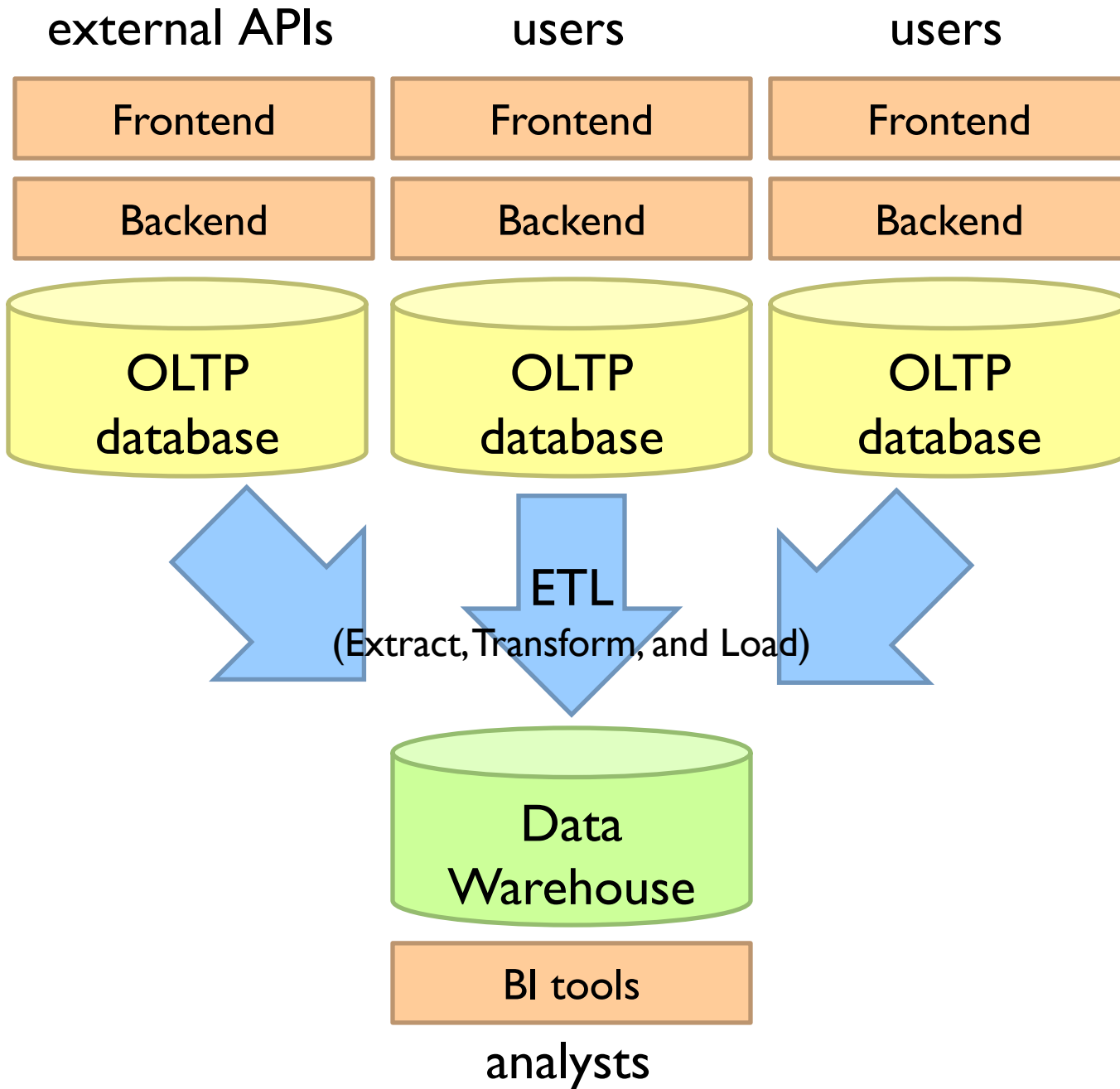
David R. Cheriton School of Computer Science

University of Waterloo

These slides are available at <http://lintool.github.io/bigdata-2018w/>

This work is licensed under a Creative Commons Attribution-Noncommercial-Share Alike 3.0 United States
See <http://creativecommons.org/licenses/by-nc-sa/3.0/us/> for details





facebook®

Jeff Hammerbacher, Information Platforms and the Rise of the Data Scientist.
In, *Beautiful Data*, O'Reilly, 2009.

“On the first day of logging the Facebook clickstream, more than 400 gigabytes of data was collected. The load, index, and aggregation processes for this data set really taxed the Oracle data warehouse. Even after significant tuning, we were unable to aggregate a day of clickstream data in less than 24 hours.”

users

Frontend

Backend

“OLTP”

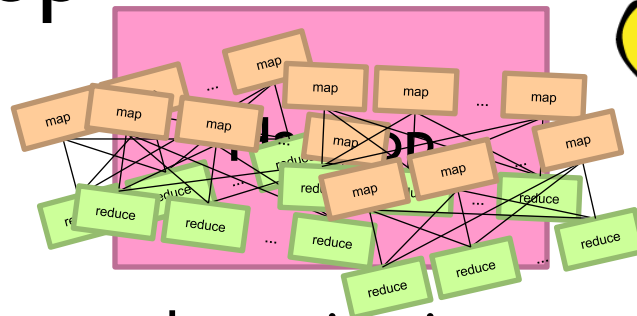
ETL

(Extract, Transform, and Load)

Wait, so why not use a database to begin with?

Cost + Scalability

SQL-on-Hadoop



data scientists



Databases are great...

If your data has structure (and you know what the structure is)

If your data is reasonably clean

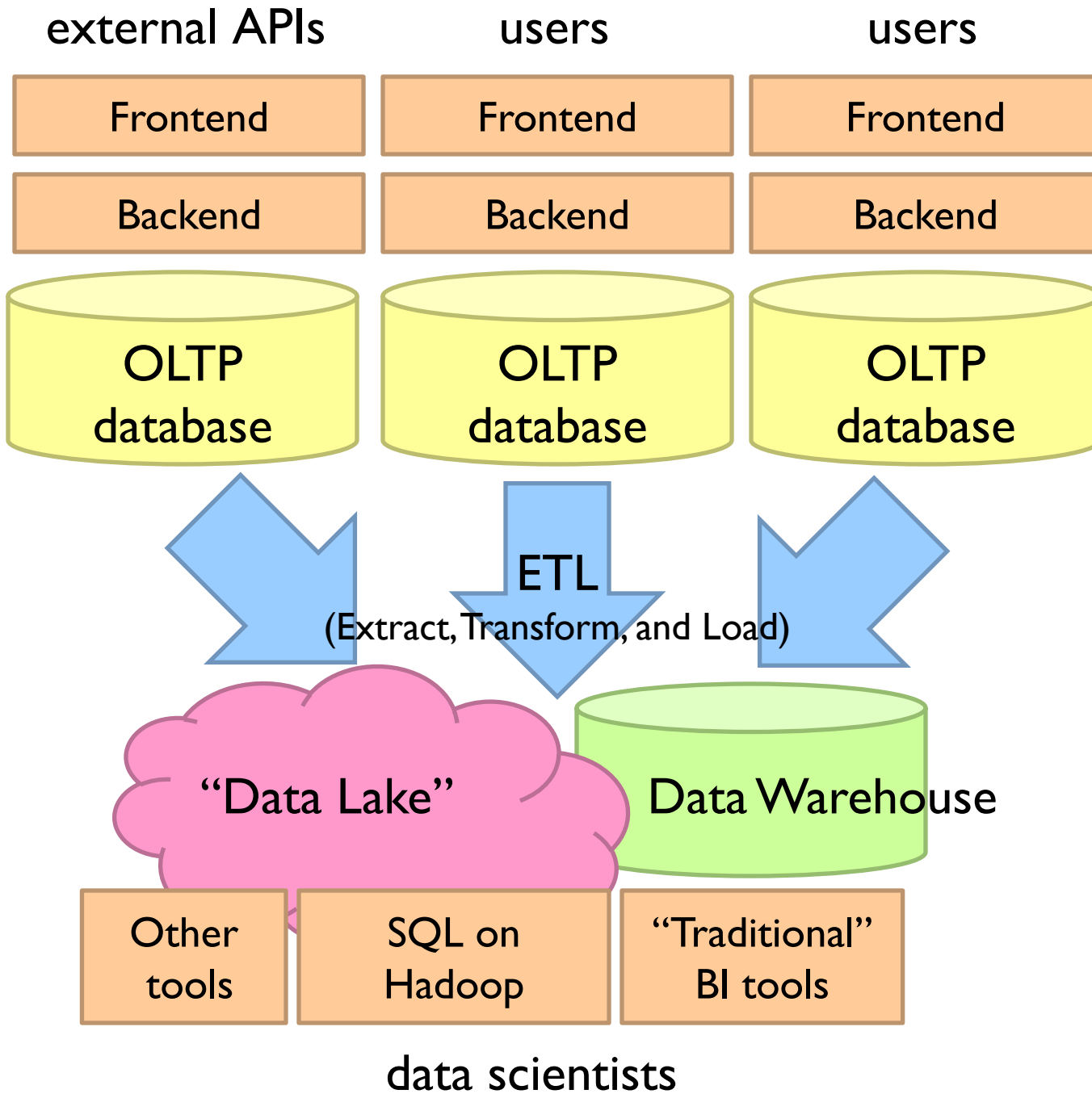
If you know what queries you're going to run ahead of time

Databases are not so great...

If your data has little structure (or you don't know the structure)

If your data is messy and noisy

If you don't know what you're looking for



external APIs

users

users

Frontend

Frontend

Frontend

Backend

Backend

Backend

OLTP
database

OLTP
database

OLTP
database

ETL

(Extract, Transform, and Load)

"Data Lake"

Data Warehouse

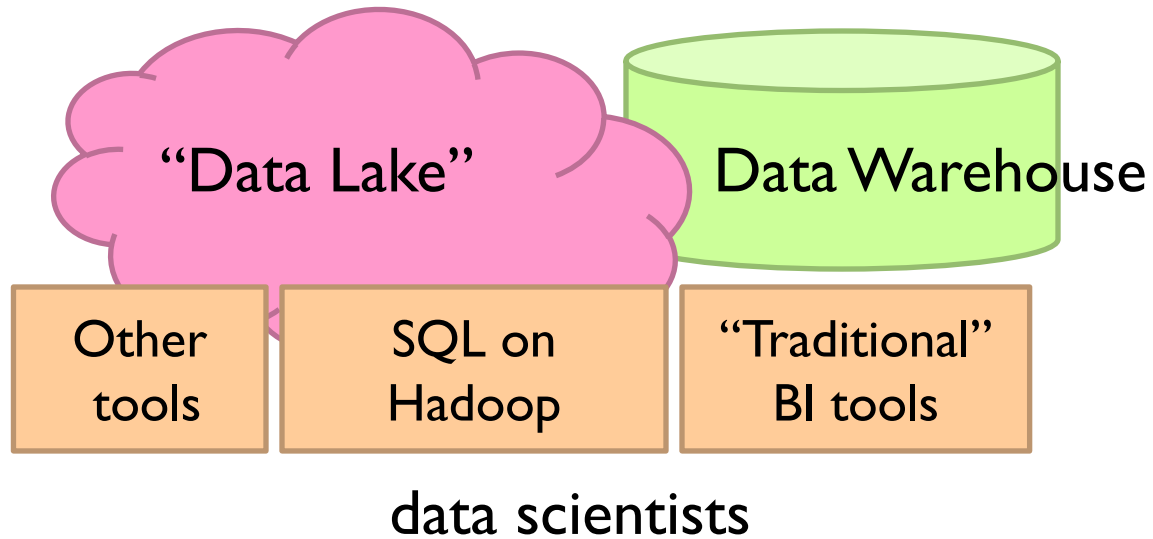
Other
tools

SQL on
Hadoop

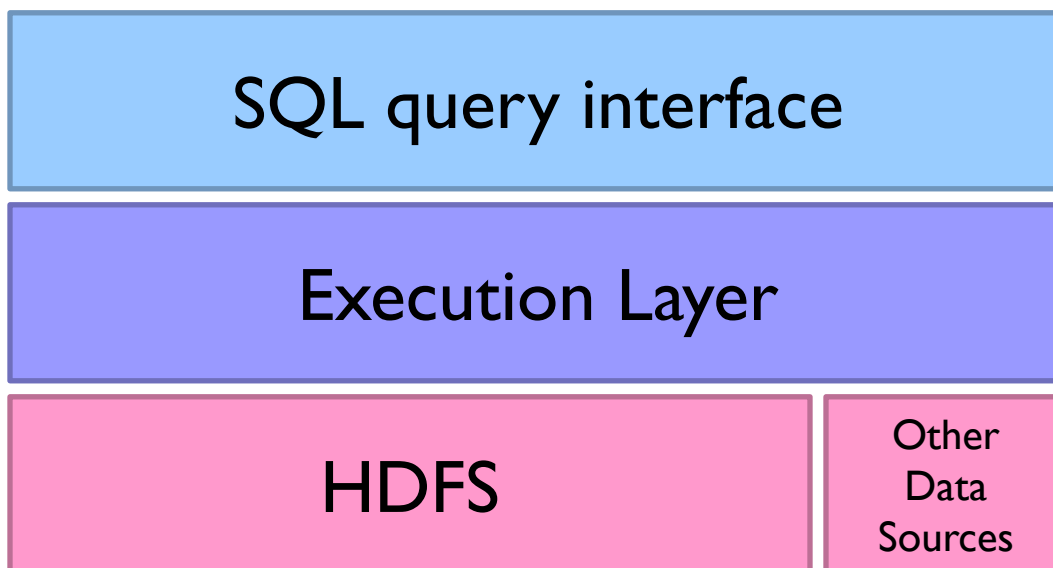
"Traditional"
BI tools

data scientists

What's the selling point of SQL-on-Hadoop?
Trade (a little?) performance for flexibility



SQL-on-Hadoop



Today: How all of this works...

Hive: Example

Relational join on two tables:

Table of word counts from Shakespeare collection

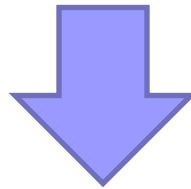
Table of word counts from the bible

```
SELECT s.word, s.freq, k.freq FROM shakespeare s
JOIN bible k ON (s.word = k.word) WHERE s.freq >= 1 AND k.freq >= 1
ORDER BY s.freq DESC LIMIT 10;
```

the	25848	62394
I	23031	8854
and	19671	38985
to	18038	13526
of	16700	34654
a	14170	8057
you	12702	2720
my	11297	4135
in	10797	12445
is	8882	6884

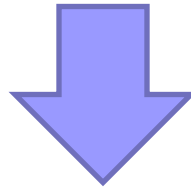
Hive: Behind the Scenes

```
SELECT s.word, s.freq, k.freq FROM shakespeare s
JOIN bible k ON (s.word = k.word) WHERE s.freq >= 1 AND k.freq >= 1
ORDER BY s.freq DESC LIMIT 10;
```



(Abstract Syntax Tree)

```
(TOK_QUERY (TOK_FROM (TOK_JOIN (TOK_TABREF shakespeare s) (TOK_TABREF bible k) (= (. (TOK_TABLE_OR_COL s)
word) (. (TOK_TABLE_OR_COL k) word)))) (TOK_INSERT (TOK_DESTINATION (TOK_DIR TOK_TMP_FILE)) (TOK_SELECT
(TOK_SELEXPR (. (TOK_TABLE_OR_COL s) word)) (TOK_SELEXPR (. (TOK_TABLE_OR_COL s) freq)) (TOK_SELEXPR (.
(TOK_TABLE_OR_COL k) freq)))) (TOK_WHERE (AND (>= (. (TOK_TABLE_OR_COL s) freq) 1) (>= (. (TOK_TABLE_OR_COL k)
freq) 1))) (TOK_ORDERBY (TOK_TABSORTCOLNAMEDESC (. (TOK_TABLE_OR_COL s) freq))) (TOK_LIMIT 10)))
```



(one or more of MapReduce jobs)

Hive: Behind the Scenes

STAGE DEPENDENCIES:

Stage-1 is a root stage
Stage-2 depends on stages: Stage-1
Stage-0 is a root stage

STAGE PLANS:

Stage: Stage-1

Map Reduce

Alias -> Map Operator Tree:

s

TableScan

alias: s

Filter Operator

predicate:

expr: (freq >= 1)

type: boolean

Reduce Output Operator

key expressions:

expr: word

type: string

sort order: +

Map-reduce partition columns:

expr: word

type: string

tag: 0

value expressions:

expr: freq

type: int

expr: word

type: string

k

TableScan

alias: k

Filter Operator

predicate:

expr: (freq >= 1)

type: boolean

Reduce Output Operator

key expressions:

expr: word

type: string

sort order: +

Map-reduce partition columns:

expr: word

type: string

tag: 1

value expressions:

expr: freq

type: int

Reduce Operator Tree:

Join Operator

condition map:

Inner Join 0 to 1

condition expressions:

0 {VALUE._col0} {VALUE._col1}

1 {VALUE._col0}

outputColumnNames: _col0, _col1, _col2

Filter Operator

predicate:

expr: ((_col0 >= 1) and (_col2 >= 1))

type: boolean

Select Operator

expressions:

expr: _col1

type: string

expr: _col0

type: int

expr: _col2

type: int

outputColumnNames: _col0, _col1, _col2

File Output Operator

compressed: false

GlobalTableId: 0

table:

input format: org.apache.hadoop.mapred.SequenceFileInputFormat

output format: org.apache.hadoop.hive ql.io.HiveSequenceFileOutputFormat

Stage: Stage-2

Map Reduce

Alias -> Map Operator Tree:

hdfs://localhost:8022/tmp/hive-training/364214370/10002

Reduce Output Operator

key expressions:

expr: _col1

type: int

sort order: -

tag: -1

value expressions:

expr: _col0

type: string

expr: _col1

type: int

expr: _col2

type: int

Reduce Operator Tree:

Extract

Limit

File Output Operator

compressed: false

GlobalTableId: 0

table:

input format: org.apache.hadoop.mapred.TextInputFormat

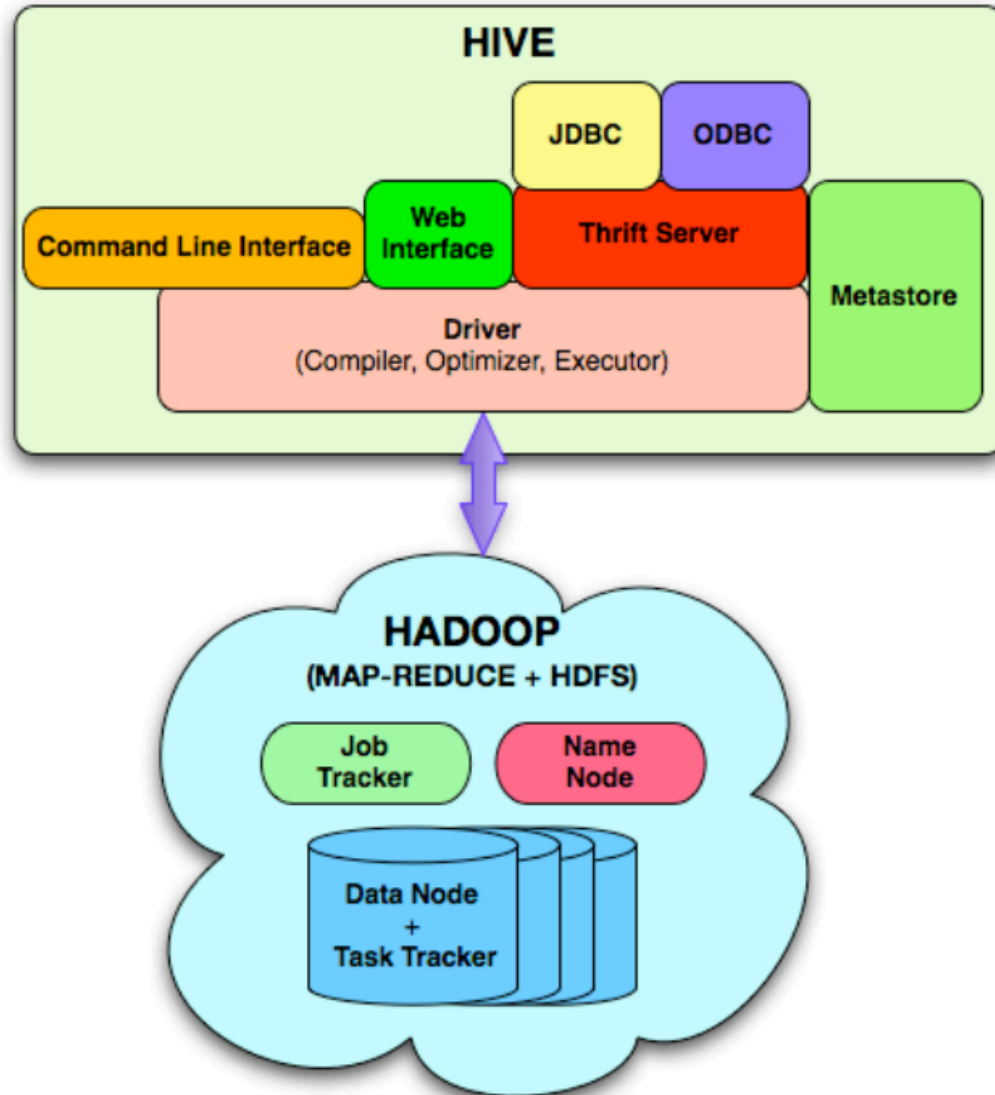
output format: org.apache.hadoop.hive ql.io.HiveIgnoreKeyTextOutputFormat

Stage: Stage-0

Fetch Operator

limit: 10

Hive Architecture



Hive Implementation

Metastore holds metadata

Tables schemas (field names, field types, etc.) and encoding
Permission information (roles and users)

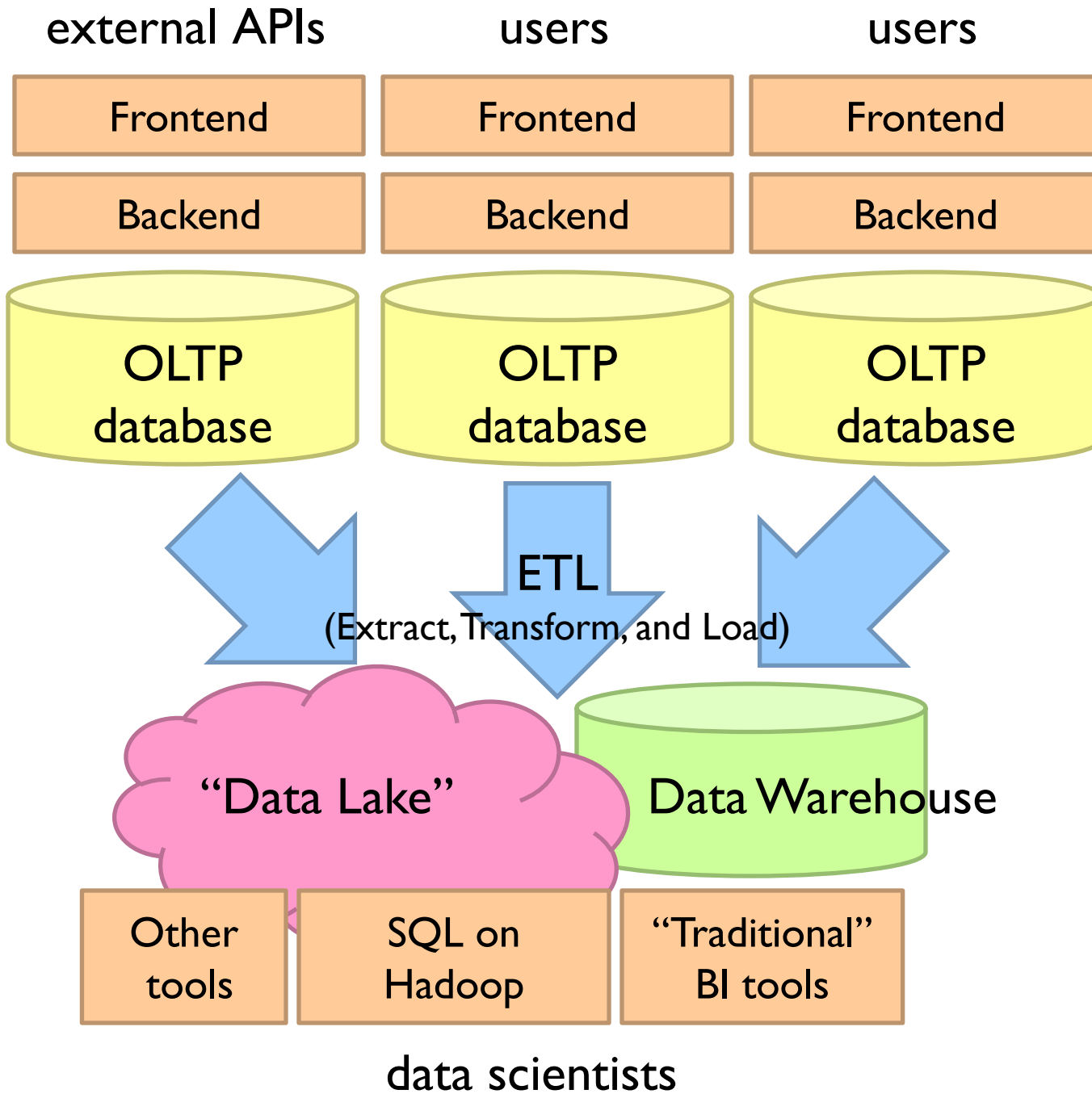
Hive data stored in HDFS

Tables in directories

Partitions of tables in sub-directories

Actual data in files (plain text or binary encoded)

Feature or bug?
(this is the essence of SQL-on-Hadoop)



external APIs

users

users

Frontend

Frontend

Frontend

Backend

Backend

Backend

OLTP
database

OLTP
database

OLTP
database

ETL

(Extract, Transform, and Load)

"Data Lake"

Data Warehouse

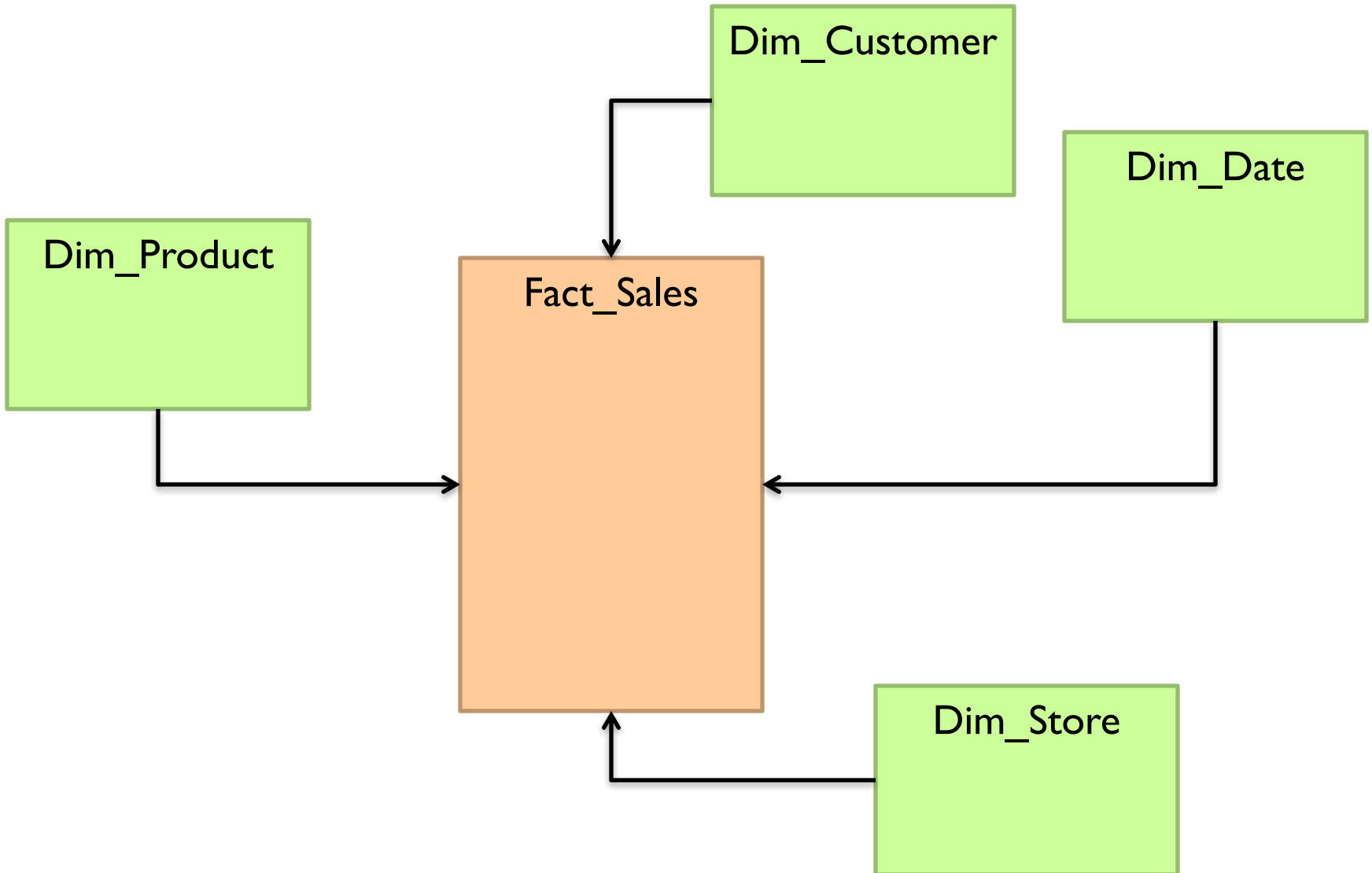
Other
tools

SQL on
Hadoop

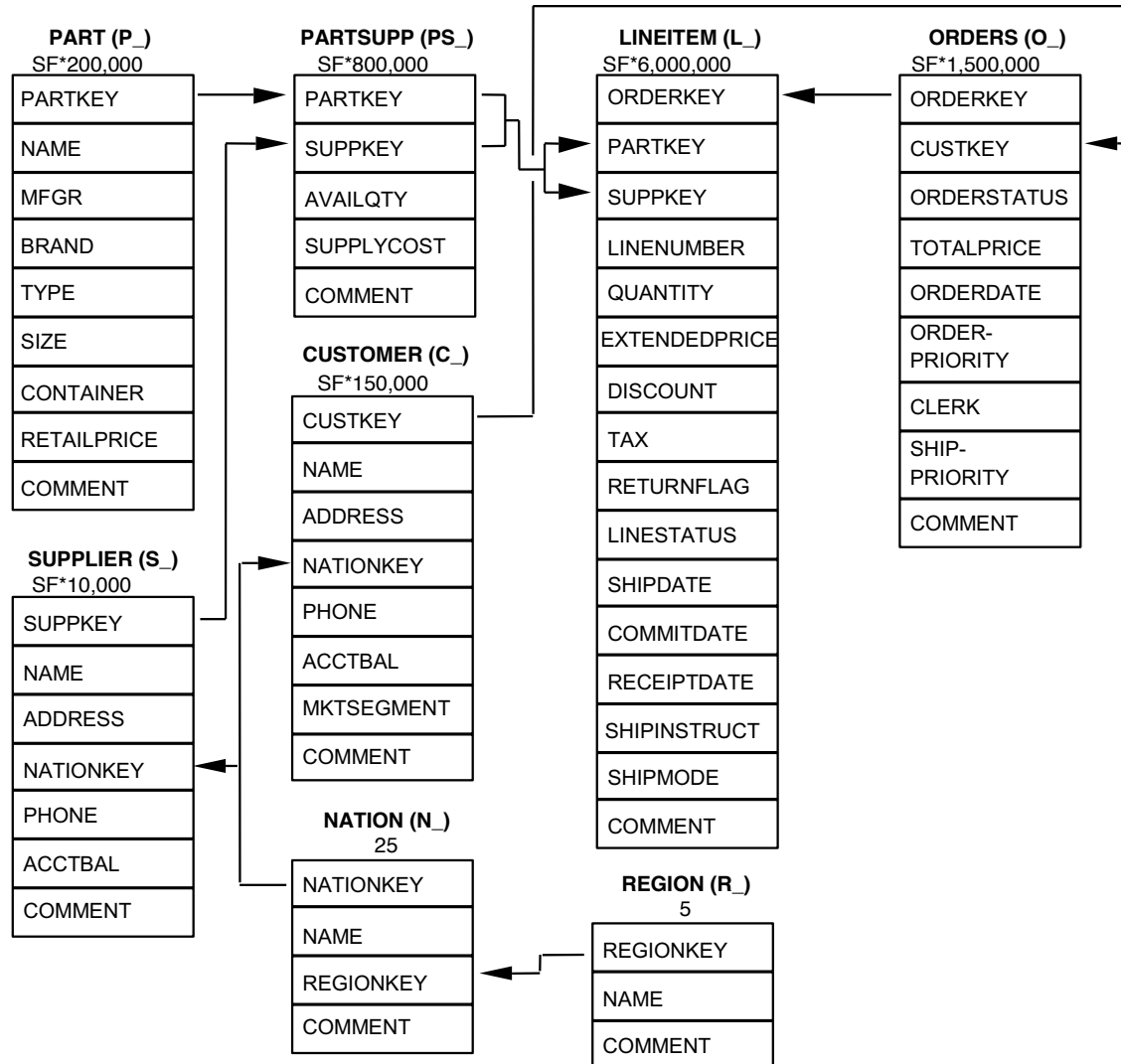
"Traditional"
BI tools

data scientists

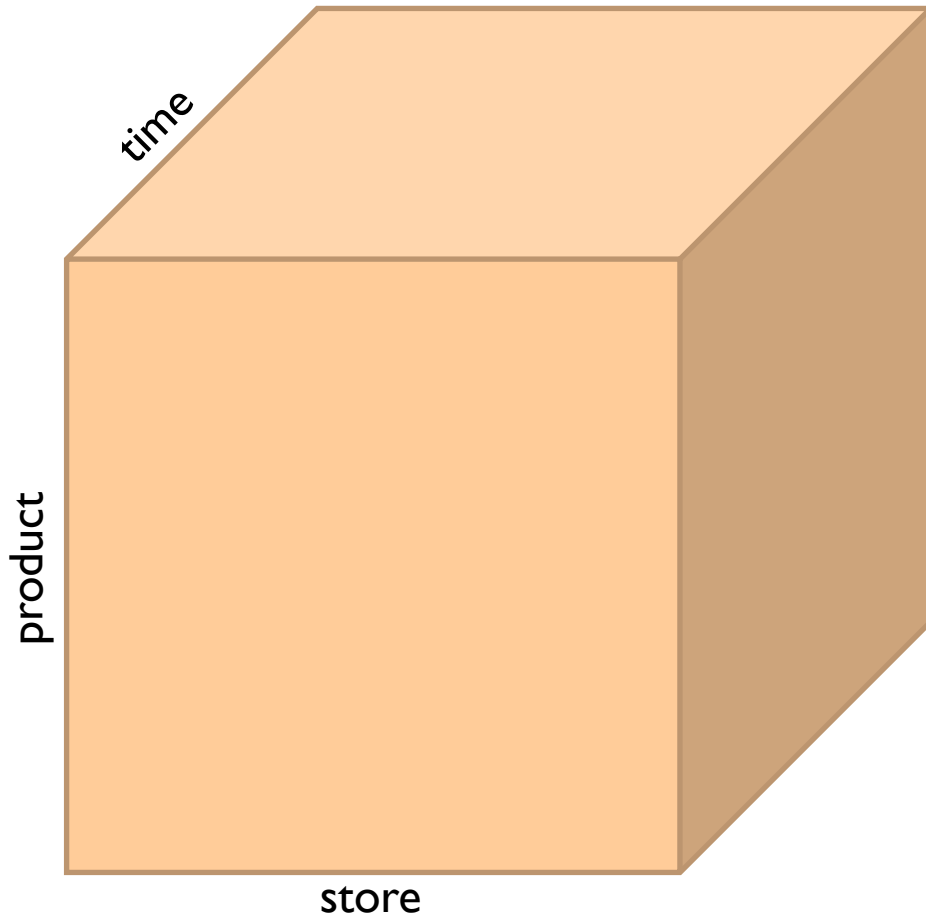
A Simple OLAP Schema



TPC-H Data Warehouse



OLAP Cubes



Common operations

slice and dice

roll up/drill down

pivot

A black and white photograph of a garden bed. The bed is filled with concrete blocks, some of which are stacked to form a border. Several plants, including a small tree and various leafy plants, are growing within the bed. The background shows more foliage and a fence.

MapReduce algorithms for processing relational data

Relational Algebra

Primitives

Projection (π)

Selection (σ)

Cartesian product (\times)

Set union (\cup)

Set difference ($-$)

Rename (ρ)

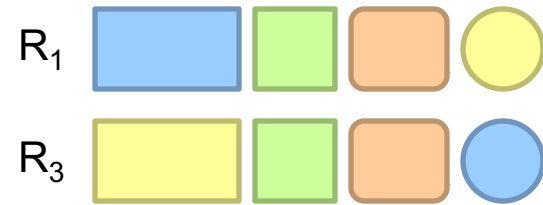
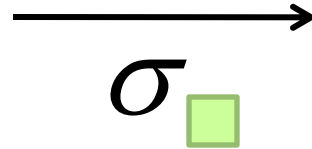
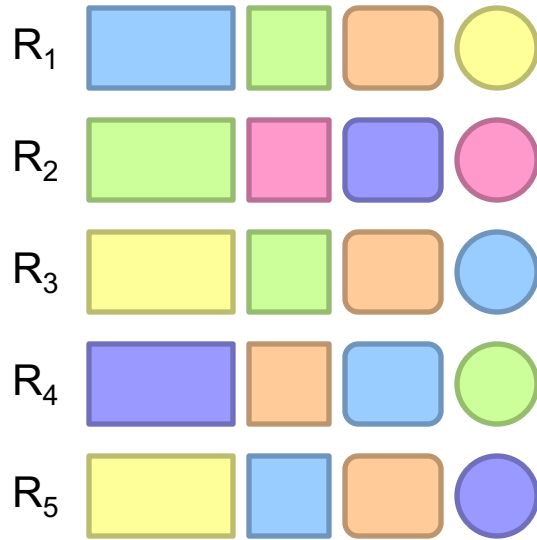
Other Operations

Join (\bowtie)

Group by... aggregation

...

Selection



Selection in MapReduce

Easy!

In mapper: process each tuple, only emit tuples that meet criteria

Can be pipelined with projection

No reducers necessary (unless to do something else)

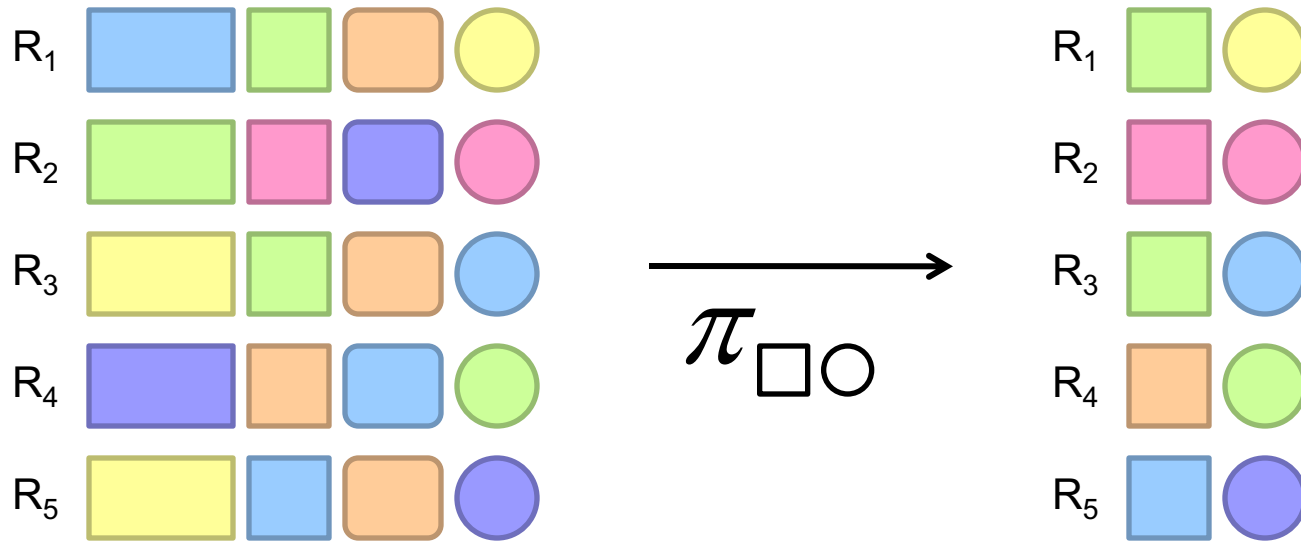
Performance mostly limited by HDFS throughput

Speed of encoding/decoding tuples becomes important

Take advantage of compression when available

Semistructured data? No problem!

Projection



Projection in MapReduce

Easy!

In mapper: process each tuple, re-emit with only projected attributes

Can be pipelined with selection

No reducers necessary (unless to do something else)

Implementation detail: bookkeeping required

Need to keep track of attribute mappings after projection

e.g., name was `r[4]`, becomes `r[1]` after projection

Performance mostly limited by HDFS throughput

Speed of encoding/decoding tuples becomes important

Take advantage of compression when available

Semistructured data? No problem!

Group by... Aggregation

Aggregation functions:

AVG, MAX, MIN, SUM, COUNT, ...

MapReduce implementation:

Map over dataset, emit tuples, keyed by group by attribute
Framework automatically groups values by group by attribute

Compute aggregation function in reducer

Optimize with combiners, in-mapper combining

You already know how to do this!

Remember this!
(week 2)

Combiner Design

Combiners and reducers share same method signature

Sometimes, reducers can serve as combiners

Often, not...

Remember: combiners are optional optimizations

Should not affect algorithm correctness

May be run 0, 1, or multiple times

Example: find average of integers associated with the same key

```
SELECT key, AVG(value) FROM r GROUP BY key;
```

Computing the Mean: Version I

```
class Mapper {
  def map(key: Text, value: Int, context: Context) = {
    context.write(key, value)
  }
}

class Reducer {
  def reduce(key: Text, values: Iterable[Int], context: Context) {
    for (value <- values) {
      sum += value
      cnt += 1
    }
    context.write(key, sum/cnt)
  }
}
```

Computing the Mean: Version 2

```
class Mapper {
  def map(key: Text, value: Int, context: Context) =
    context.write(key, value)
}
class Combiner {
  def reduce(key: Text, values: Iterable[Int], context: Context) = {
    for (value <- values) {
      sum += value
      cnt += 1
    }
    context.write(key, (sum, cnt))
  }
}
class Reducer {
  def reduce(key: Text, values: Iterable[Pair], context: Context) = {
    for (value <- values) {
      sum += value.left
      cnt += value.right
    }
    context.write(key, sum/cnt)
  }
}
```

Computing the Mean: Version 3

```
class Mapper {
  def map(key: Text, value: Int, context: Context) =
    context.write(key, (value, 1))
}
class Combiner {
  def reduce(key: Text, values: Iterable[Pair], context: Context) = {
    for (value <- values) {
      sum += value.left
      cnt += value.right
    }
    context.write(key, (sum, cnt))
  }
}
class Reducer {
  def reduce(key: Text, values: Iterable[Pair], context: Context) = {
    for (value <- values) {
      sum += value.left
      cnt += value.right
    }
    context.write(key, sum/cnt)
  }
}
```

Computing the Mean: Version 4

```
class Mapper {
  val sums = new HashMap()
  val counts = new HashMap()

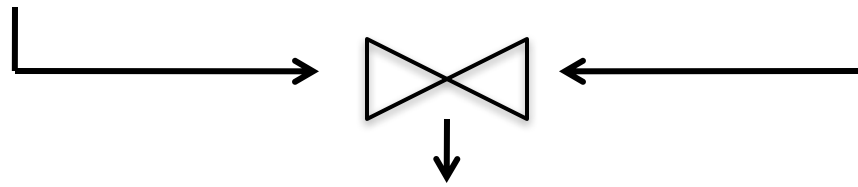
  def map(key: Text, value: Int, context: Context) = {
    sums(key) += value
    counts(key) += 1
  }

  def cleanup(context: Context) = {
    for (key <- counts) {
      context.write(key, (sums(key), counts(key)))
    }
  }
}
```

Relational Joins

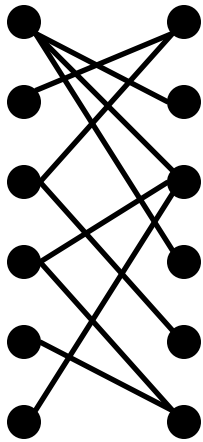


Relational Joins

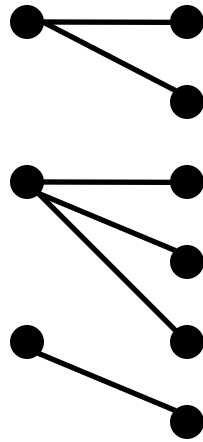


(More precisely, an inner join)

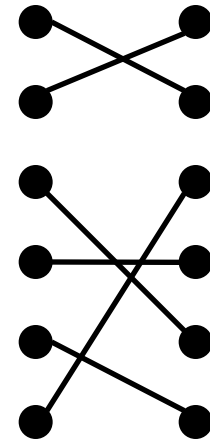
Types of Relationships



Many-to-Many



One-to-Many



One-to-One

Join Algorithms in MapReduce

Reduce-side join
aka repartition join
aka shuffle join

Map-side join
aka sort-merge join

Hash join
aka broadcast join
aka replicated join

Reduce-side Join

aka repartition join, shuffle join

Basic idea: group by join key

Map over both datasets <Huh?

Emit tuple as value with join key as the intermediate key

Execution framework brings together tuples sharing the same key

Perform join in reducer

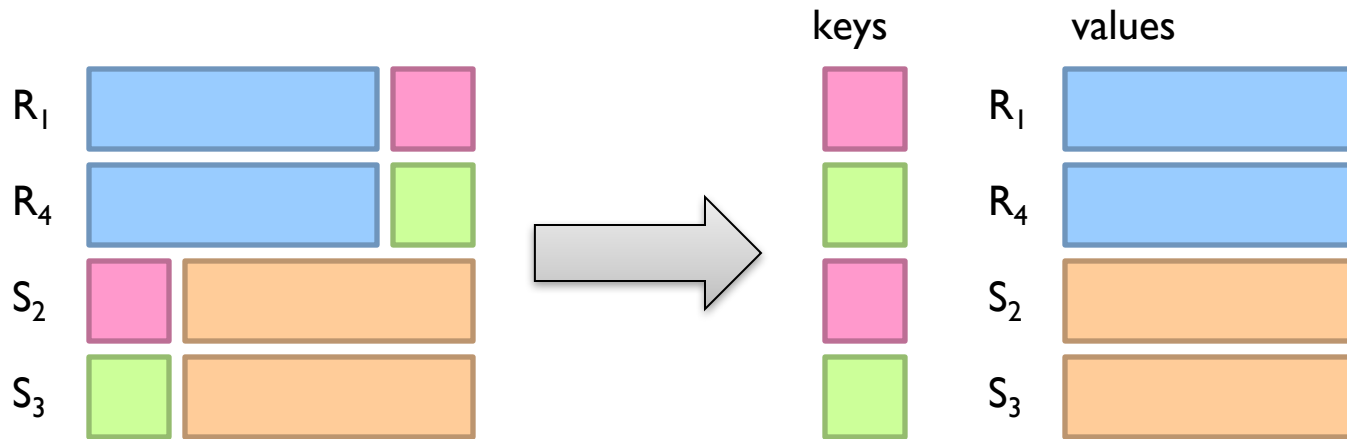
Two variants

I-to-I joins

I-to-many and many-to-many joins

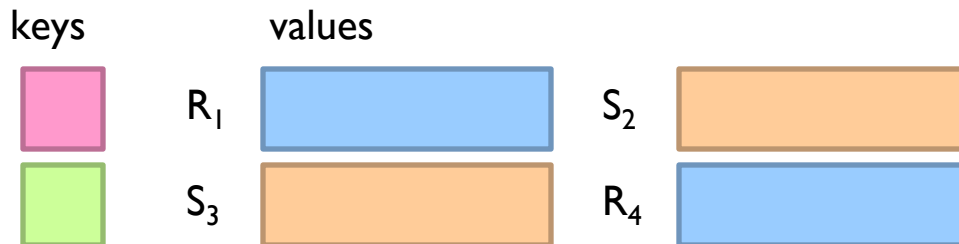
Reduce-side Join: 1-to-1

Map



Remember to "tag" the tuple as being from R or S...

Reduce

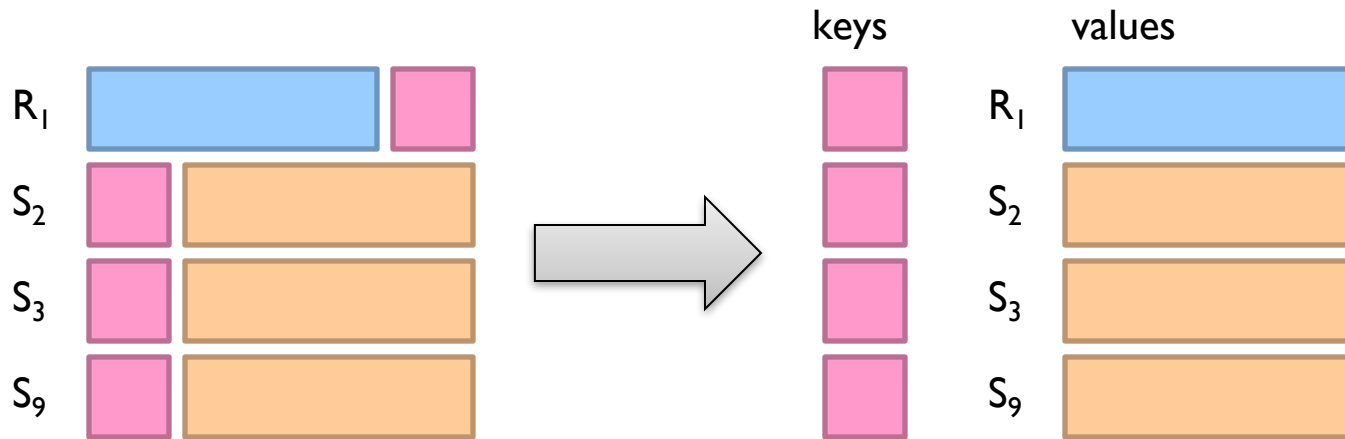


Note: no guarantee if R is going to come first or S

More precisely, an inner join: What about outer joins?

Reduce-side Join: 1-to-many

Map



Reduce



What's the problem?

Secondary Sorting

MapReduce sorts input to reducers by key

Values may be arbitrarily ordered

What if we want to sort value also?

E.g., $k \rightarrow (v_1, r), (v_3, r), (v_4, r), (v_8, r) \dots$

Secondary Sorting: Solutions

Solution 1

Buffer values in memory, then sort

Why is this a bad idea?

Solution 2

“Value-to-key conversion” : form composite intermediate key, (k, v_i)

Let the execution framework do the sorting

Preserve state across multiple key-value pairs to handle processing

Anything else we need to do?

Value-to-Key Conversion

Before

$k \rightarrow (v_8, r_4), (v_1, r_1), (v_4, r_3), (v_3, r_2) \dots$

Values arrive in arbitrary order...

After

$(k, v_1) \rightarrow r_1$

$(k, v_3) \rightarrow r_2$

$(k, v_4) \rightarrow r_3$

$(k, v_8) \rightarrow r_4$

...

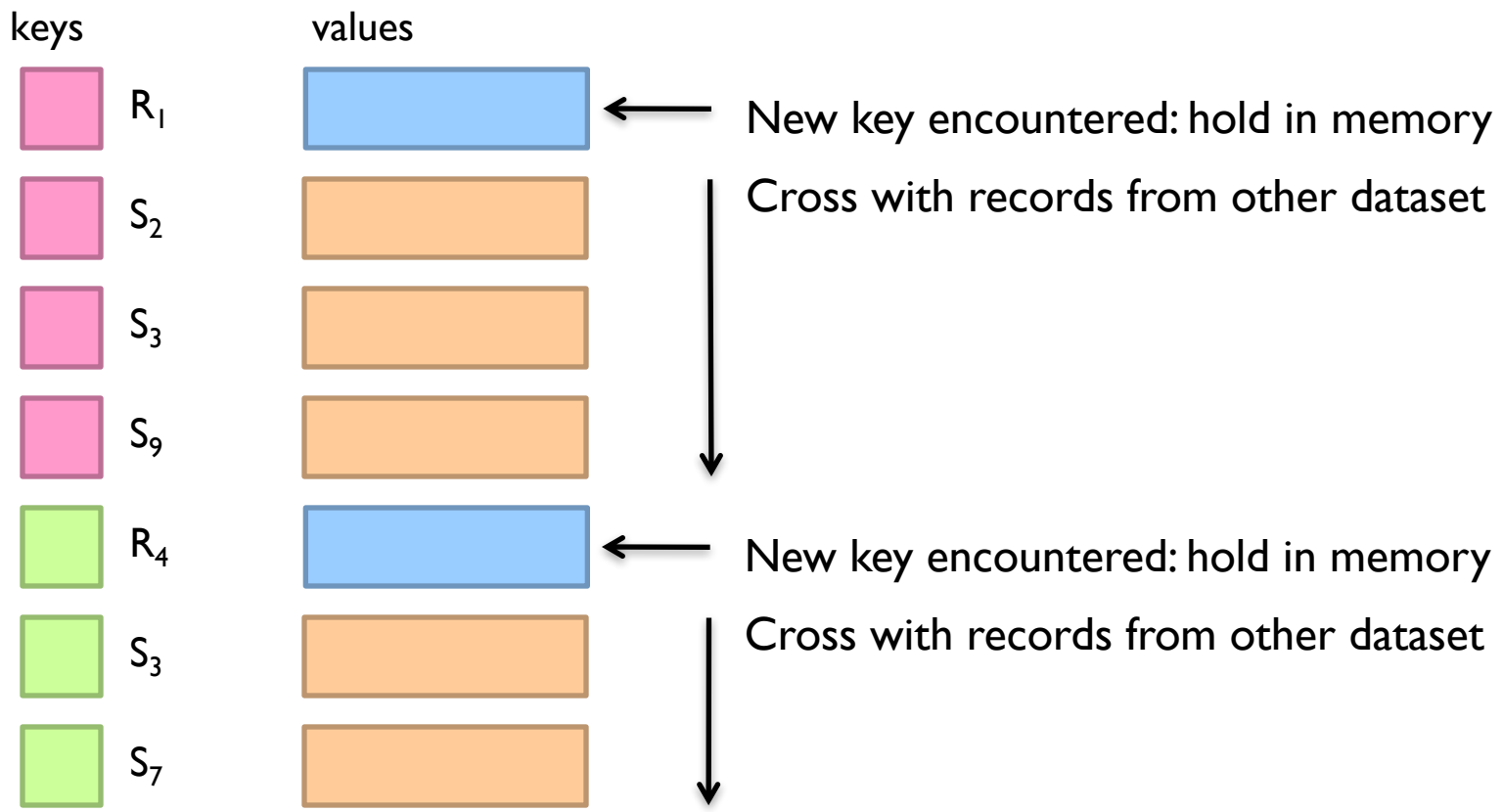
Values arrive in sorted order...

Process by preserving state across multiple keys

Remember to partition correctly!

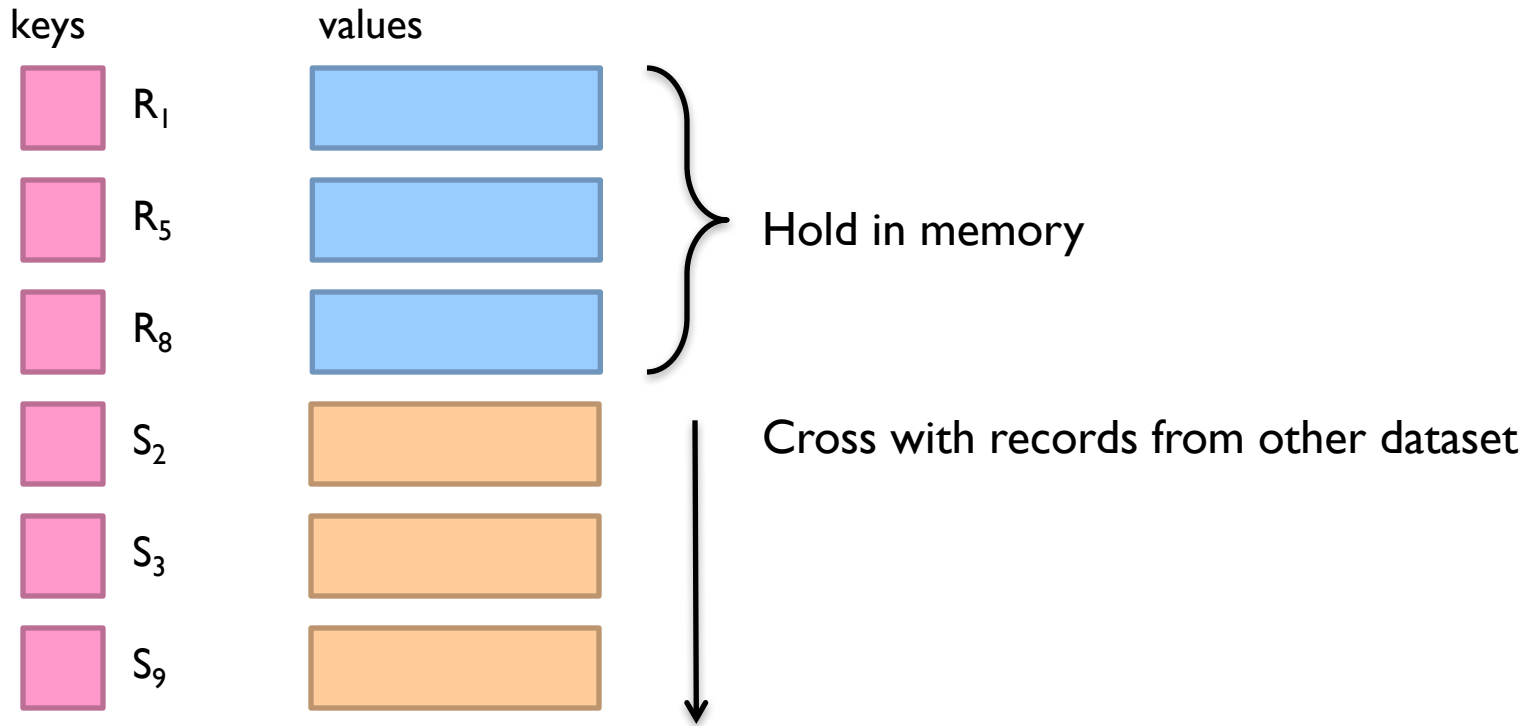
Reduce-side Join: V-to-K Conversion

In reducer...



Reduce-side Join: many-to-many

In reducer...

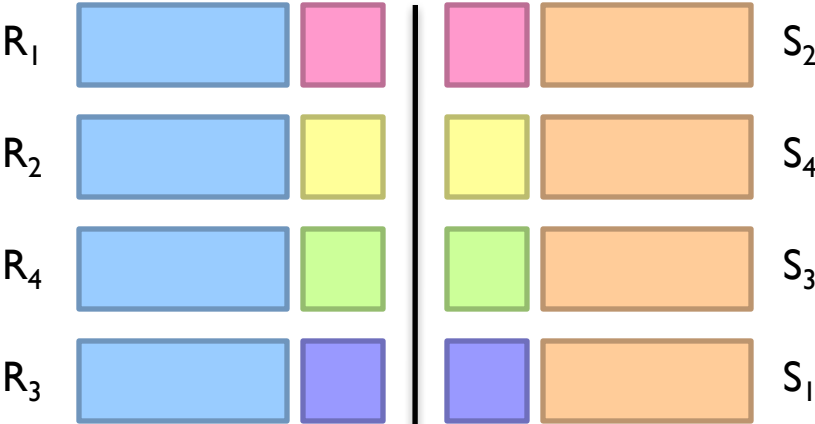


What's the problem?

Map-side Join

aka sort-merge join

Assume two datasets are sorted by the join key:

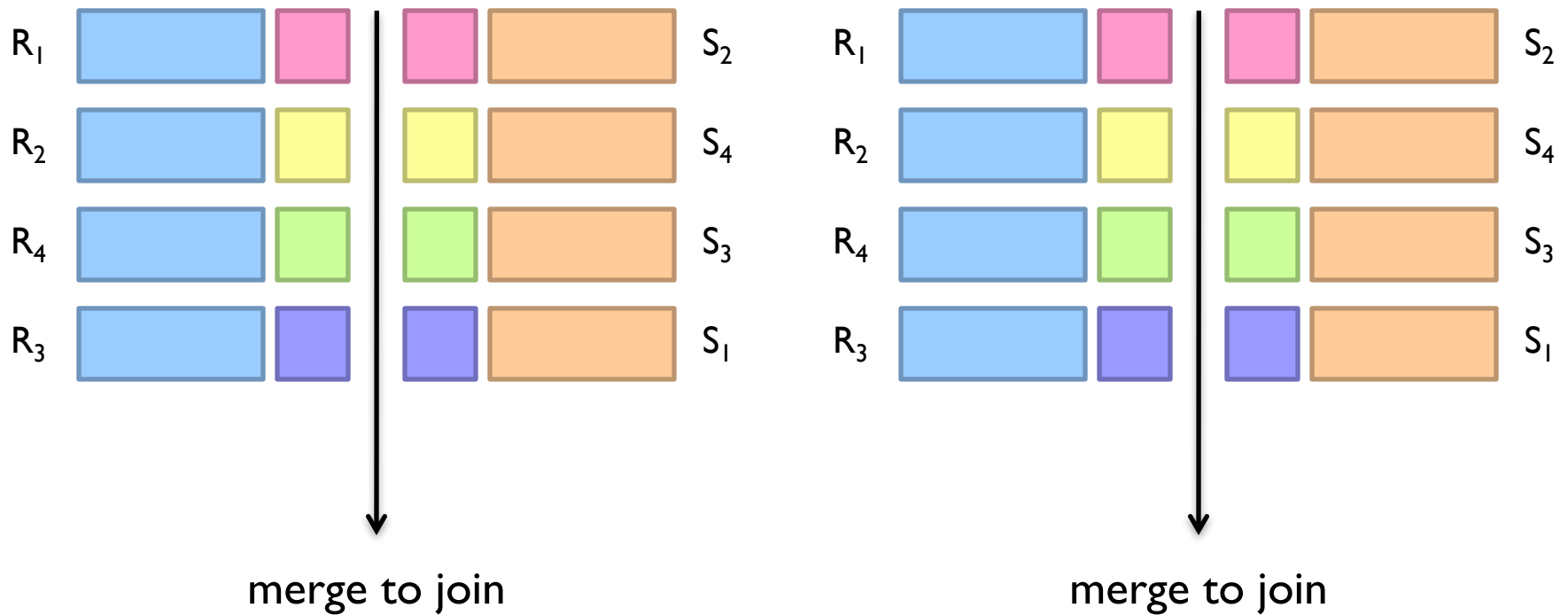


merge to join

Map-side Join

aka sort-merge join

Assume two datasets are sorted by the join key:



How can we parallelize this? Co-partitioning

Map-side Join

aka sort-merge join

Works if...

Two datasets are co-partitioned
Sorted by join key

MapReduce implementation:

Map over one dataset, read from other corresponding partition
No reducers necessary (unless to do something else)

Co-partitioned, sorted datasets: realistic to expect?

Hash Join

aka broadcast join, replicated join

Basic idea:

Load one dataset into memory in a hashmap, keyed by join key
Read other dataset, probe for join key

Works if...

$R \ll S$ and R fits into memory *<When?*

MapReduce implementation:

Distribute R to all nodes (e.g., DistributedCache)

Map over S , each mapper loads R in memory and builds the hashmap

For every tuple in S , probe join key in R

No reducers necessary (unless to do something else)

Hash Join Variants

Co-partitioned variant:

R and S co-partitioned (but not sorted)?

Only need to build hashmap on the corresponding partition

Striped variant:

R too big to fit into memory?

Divide R into R_1, R_2, R_3, \dots s.t. each R_n fits into memory

Perform hash join: $\forall n, R_n \bowtie S$

Take the union of all join results

Use a global key-value store:

Load R into memcached (or Redis)

Probe global key-value store for join key

Which join to use?

Hash join > map-side join > reduce-side join

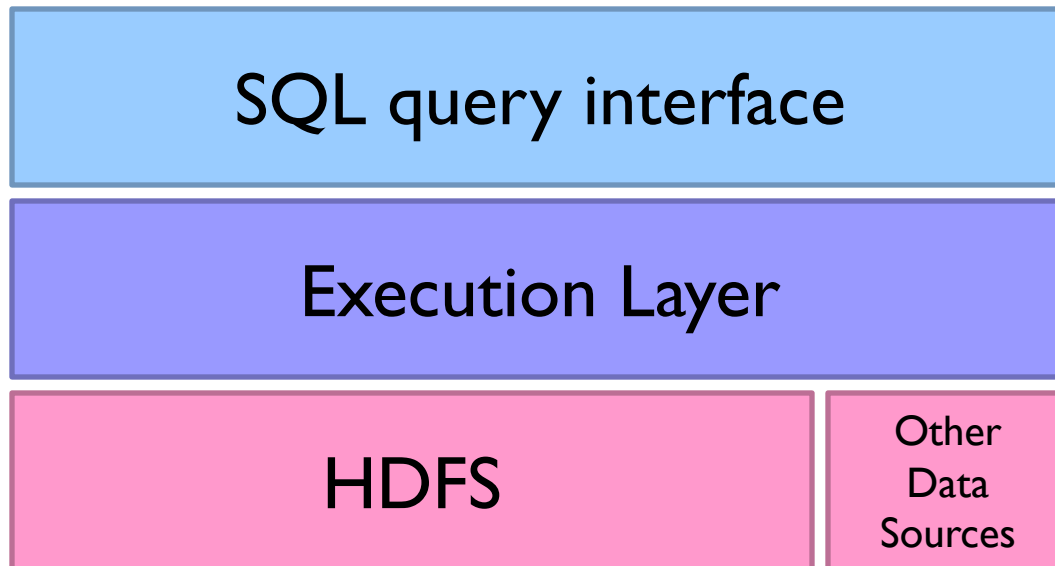
Limitations of each?

In-memory join: memory

Map-side join: sort order and partitioning

Reduce-side join: general purpose

SQL-on-Hadoop



Putting Everything Together

```
SELECT big1.fx, big2.fy, small.fz
FROM big1
JOIN big2 ON big1.id1 = big2.id1
JOIN small ON big1.id2 = small.id2
WHERE big1.fx = 2015 AND
       big2.f1 < 40 AND
       big2.f2 > 2;
```

Build logical plan

Optimize logical plan

Select physical plan

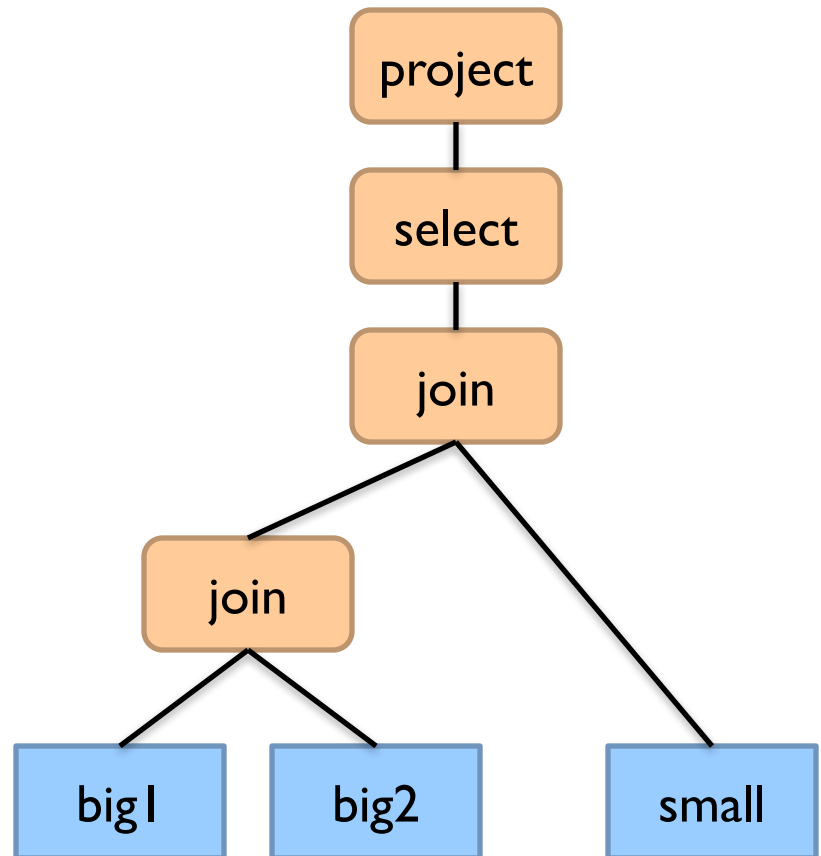
Putting Everything Together

```
SELECT big1.fx, big2.fy, small.fz
FROM big1
JOIN big2 ON big1.id1 = big2.id1
JOIN small ON big1.id2 = small.id2
WHERE big1.fx = 2015 AND
       big2.f1 < 40 AND
       big2.f2 > 2;
```

Build logical plan

Optimize logical plan

Select physical plan



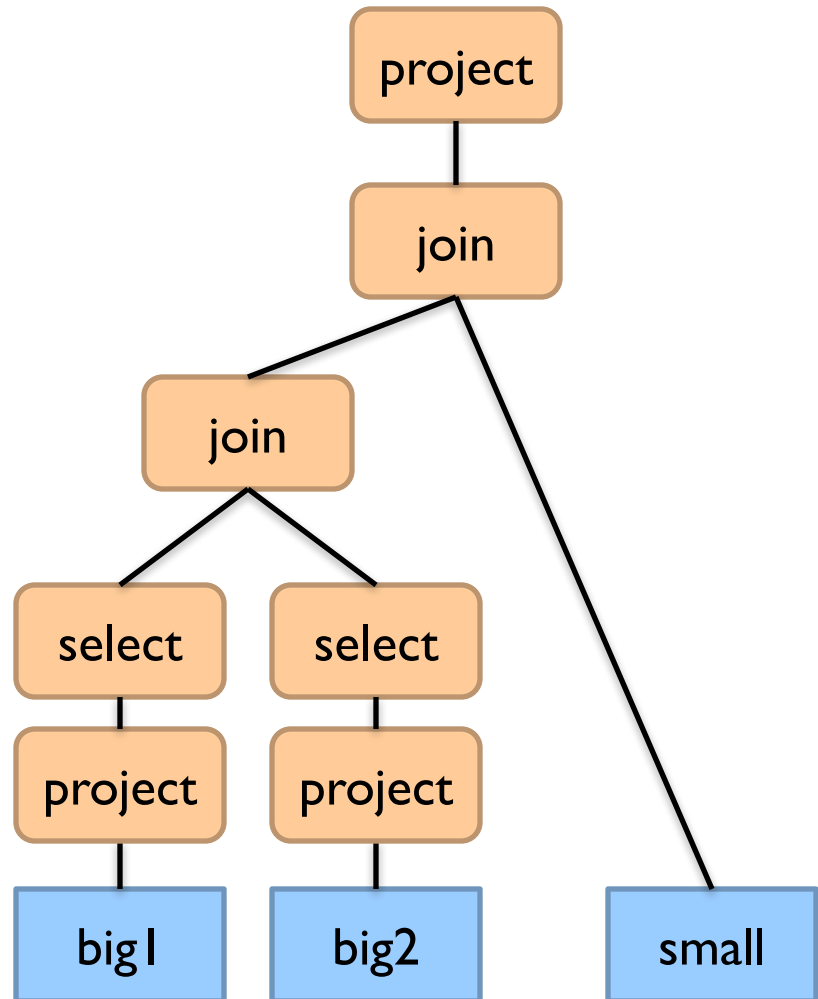
Putting Everything Together

```
SELECT big1.fx, big2.fy, small.fz
FROM big1
JOIN big2 ON big1.id1 = big2.id1
JOIN small ON big1.id2 = small.id2
WHERE big1.fx = 2015 AND
       big2.f1 < 40 AND
       big2.f2 > 2;
```

Build logical plan

Optimize logical plan

Select physical plan



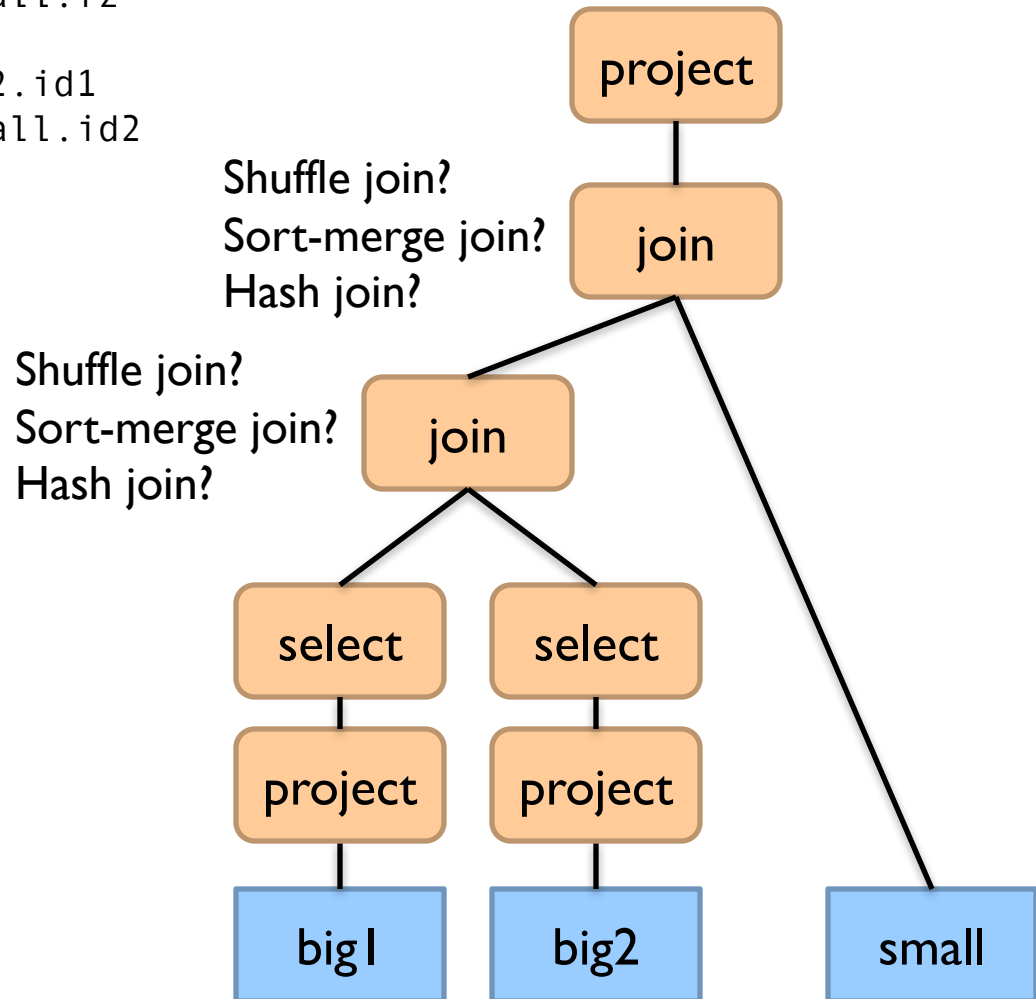
Putting Everything Together

```
SELECT big1.fx, big2.fy, small.fz
FROM big1
JOIN big2 ON big1.id1 = big2.id1
JOIN small ON big1.id2 = small.id2
WHERE big1.fx = 2015 AND
       big2.f1 < 40 AND
       big2.f2 > 2;
```

Build logical plan

Optimize logical plan

Select physical plan



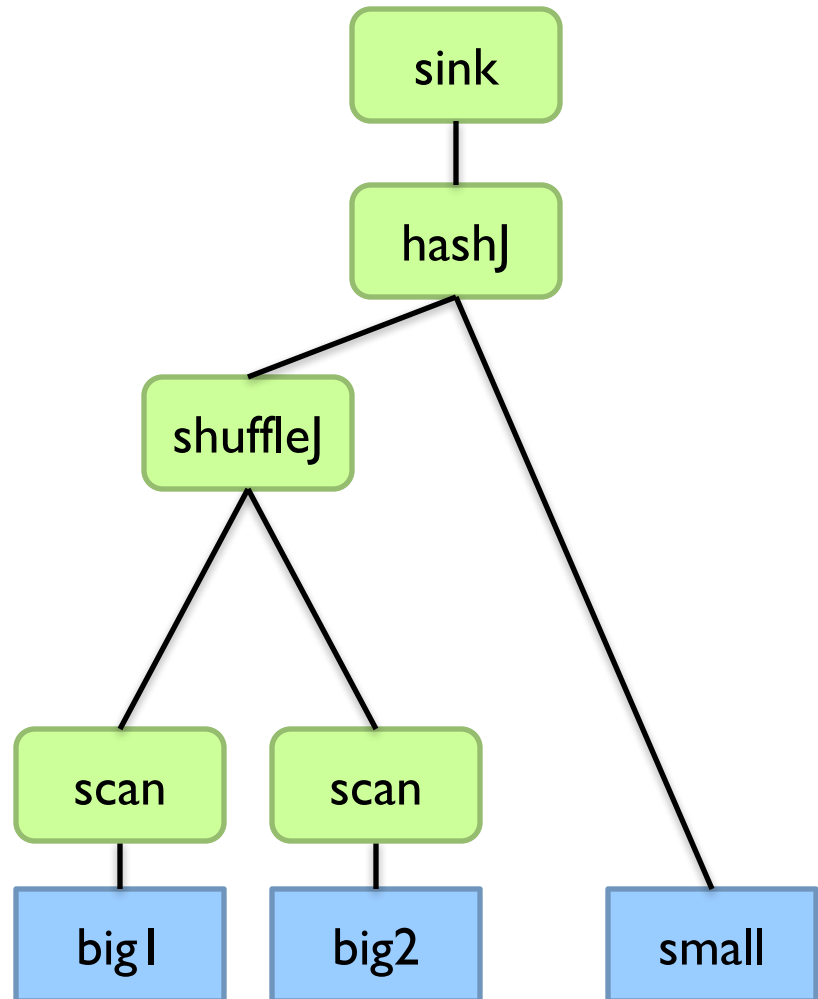
Putting Everything Together

```
SELECT big1.fx, big2.fy, small.fz
FROM big1
JOIN big2 ON big1.id1 = big2.id1
JOIN small ON big1.id2 = small.id2
WHERE big1.fx = 2015 AND
       big2.f1 < 40 AND
       big2.f2 > 2;
```

Build logical plan

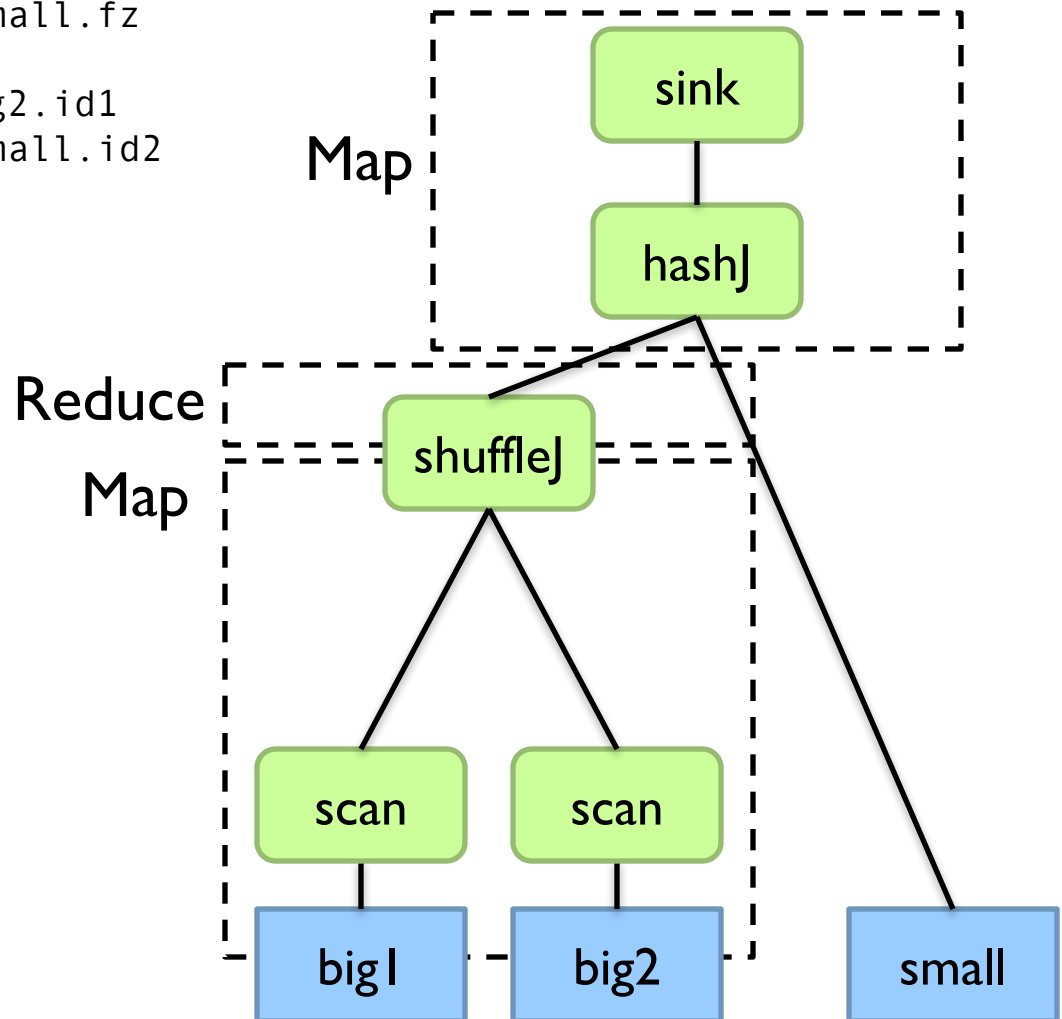
Optimize logical plan

Select physical plan



Putting Everything Together

```
SELECT big1.fx, big2.fy, small.fz
FROM big1
JOIN big2 ON big1.id1 = big2.id1
JOIN small ON big1.id2 = small.id2
WHERE big1.fx = 2015 AND
      big2.f1 < 40 AND
      big2.f2 > 2;
```



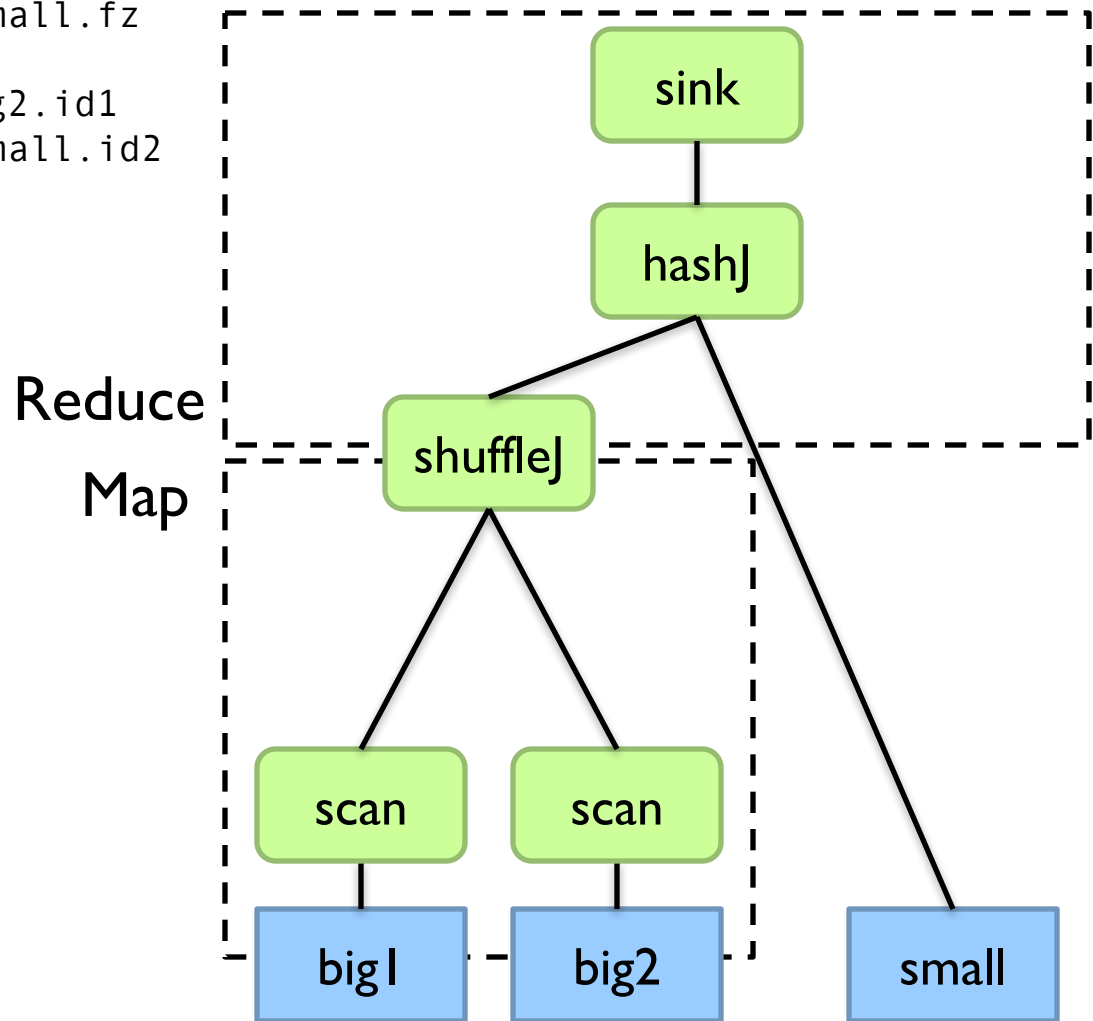
Build logical plan

Optimize logical plan

Select physical plan

Putting Everything Together

```
SELECT big1.fx, big2.fy, small.fz
FROM big1
JOIN big2 ON big1.id1 = big2.id1
JOIN small ON big1.id2 = small.id2
WHERE big1.fx = 2015 AND
       big2.f1 < 40 AND
       big2.f2 > 2;
```



Build logical plan

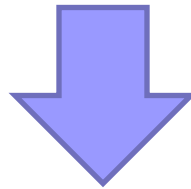
Optimize logical plan

Select physical plan

Hive: Behind the Scenes

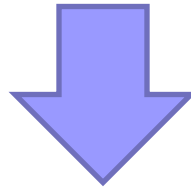
Now you understand what's going on here!

```
SELECT s.word, s.freq, k.freq FROM shakespeare s
JOIN bible k ON (s.word = k.word) WHERE s.freq >= 1 AND k.freq >= 1
ORDER BY s.freq DESC LIMIT 10;
```



(Abstract Syntax Tree)

```
(TOK_QUERY (TOK_FROM (TOK_JOIN (TOK_TABREF shakespeare s) (TOK_TABREF bible k) (= (. (TOK_TABLE_OR_COL s)
word) (. (TOK_TABLE_OR_COL k) word)))) (TOK_INSERT (TOK_DESTINATION (TOK_DIR TOK_TMP_FILE)) (TOK_SELECT
(TOK_SELEXPR (. (TOK_TABLE_OR_COL s) word)) (TOK_SELEXPR (. (TOK_TABLE_OR_COL s) freq)) (TOK_SELEXPR (.
(TOK_TABLE_OR_COL k) freq)))) (TOK_WHERE (AND (>= (. (TOK_TABLE_OR_COL s) freq) 1) (>= (. (TOK_TABLE_OR_COL k)
freq) 1))) (TOK_ORDERBY (TOK_TABSORTCOLNAMEDESC (. (TOK_TABLE_OR_COL s) freq))) (TOK_LIMIT 10)))
```



(one or more of MapReduce jobs)

Hive: Behind the Scenes

Now you understand what's going on here!

STAGE DEPENDENCIES:

Stage-1 is a root stage
Stage-2 depends on stages: Stage-1
Stage-0 is a root stage

STAGE PLANS:

Stage: Stage-1

Map Reduce

Alias -> Map Operator Tree:

s

TableScan

alias: s

Filter Operator

predicate:

expr: (freq >= 1)

type: boolean

Reduce Output Operator

key expressions:

expr: word

type: string

sort order: +

Map-reduce partition columns:

expr: word

type: string

tag: 0

value expressions:

expr: freq

type: int

expr: word

type: string

k

TableScan

alias: k

Filter Operator

predicate:

expr: (freq >= 1)

type: boolean

Reduce Output Operator

key expressions:

expr: word

type: string

sort order: +

Map-reduce partition columns:

expr: word

type: string

tag: 1

value expressions:

expr: freq

type: int

Reduce Operator Tree:

Join Operator

condition map:

Inner Join 0 to 1

condition expressions:

0 {VALUE._col0} {VALUE._col1}

1 {VALUE._col0}

outputColumnNames: _col0, _col1, _col2

Filter Operator

predicate:

expr: ((_col0 >= 1) and (_col2 >= 1))

type: boolean

Select Operator

expressions:

expr: _col1

type: string

expr: _col0

type: int

expr: _col2

type: int

outputColumnNames: _col0, _col1, _col2

File Output Operator

compressed: false

GlobalTableId: 0

table:

input format: org.apache.hadoop.mapred.SequenceFileInputFormat

output format: org.apache.hadoop.hive ql.io.HiveSequenceFileOutputFormat

Stage: Stage-2

Map Reduce

Alias -> Map Operator Tree:

hdfs://localhost:8022/tmp/hive-training/364214370/10002

Reduce Output Operator

key expressions:

expr: _col1

type: int

sort order: -

tag: -1

value expressions:

expr: _col0

type: string

expr: _col1

type: int

expr: _col2

type: int

Reduce Operator Tree:

Extract

Limit

File Output Operator

compressed: false

GlobalTableId: 0

table:

input format: org.apache.hadoop.mapred.TextInputFormat

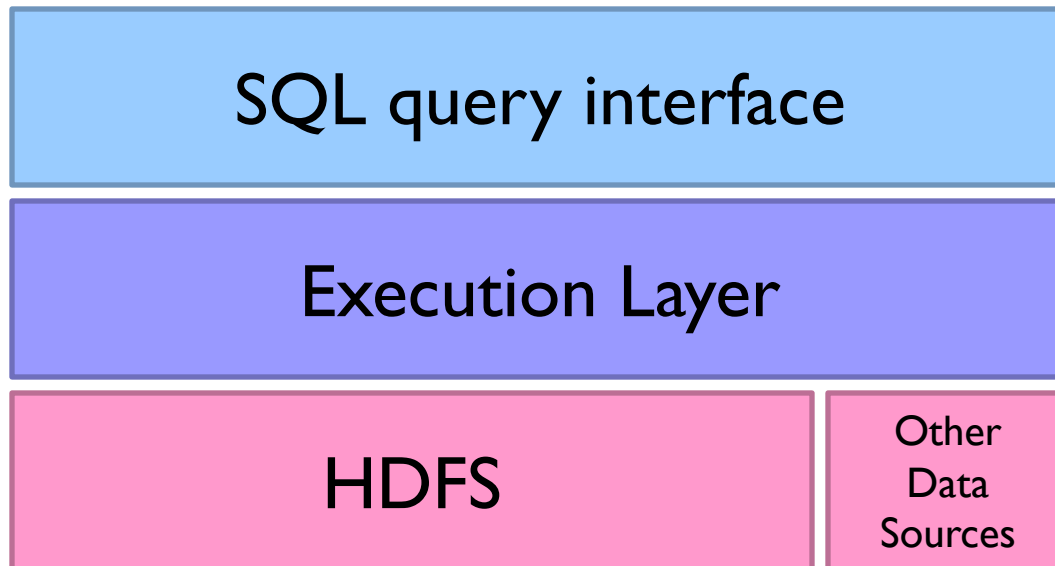
output format: org.apache.hadoop.hive ql.io.HiveIgnoreKeyTextOutputFormat

Stage: Stage-0

Fetch Operator

limit: 10

SQL-on-Hadoop



What about Spark SQL?

Based on the DataFrame API:

A distributed collection of data organized into named columns

Two ways of specifying SQL queries:

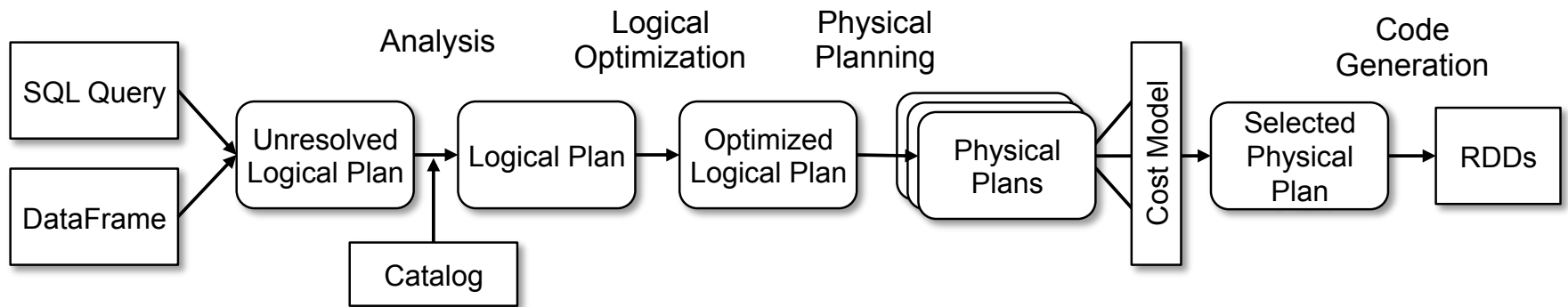
Directly:

```
val sqlContext = ... // An existing SQLContext
val df = sqlContext.sql("SELECT * FROM table")
// df is a dataframe, can be further manipulated...
```

Via DataFrame API:

```
// employees is a dataframe:
employees
  .join(dept, employees ("deptId") === dept ("id"))
  .where(employees("gender") === "female")
  .groupBy(dept("id"), dept ("name"))
  .agg(count("name"))
```

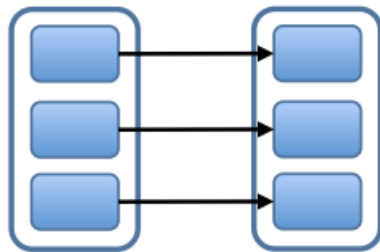
Spark SQL: Query Planning



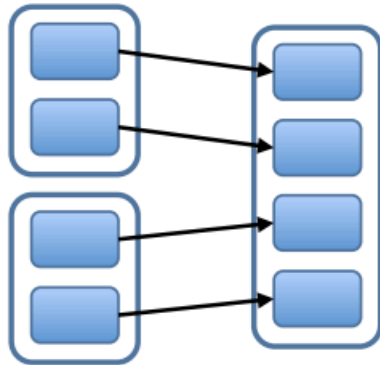
At the end of the day... it's transformations on RDDs

Spark SQL: Physical Execution

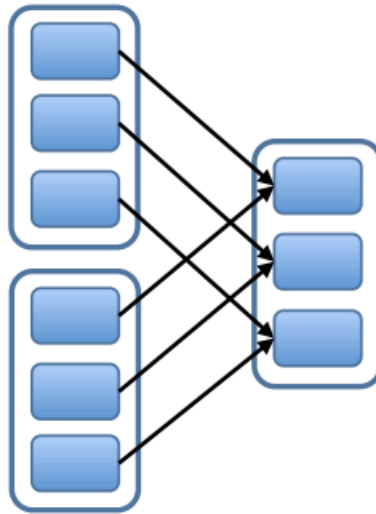
Narrow Dependencies:



map, filter



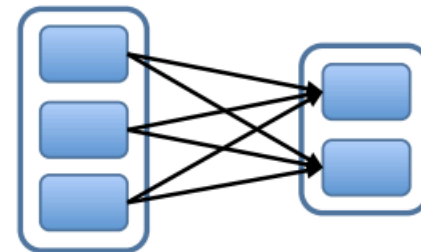
union



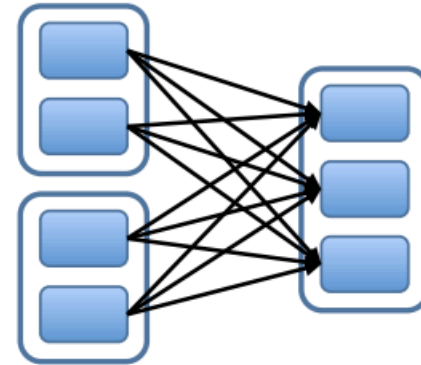
join with inputs
co-partitioned

= Map-side join

Wide Dependencies:



groupByKey



join with inputs not
co-partitioned

= Reduce-side join

Hash join with broadcast variables

Hadoop Data Warehouse Design

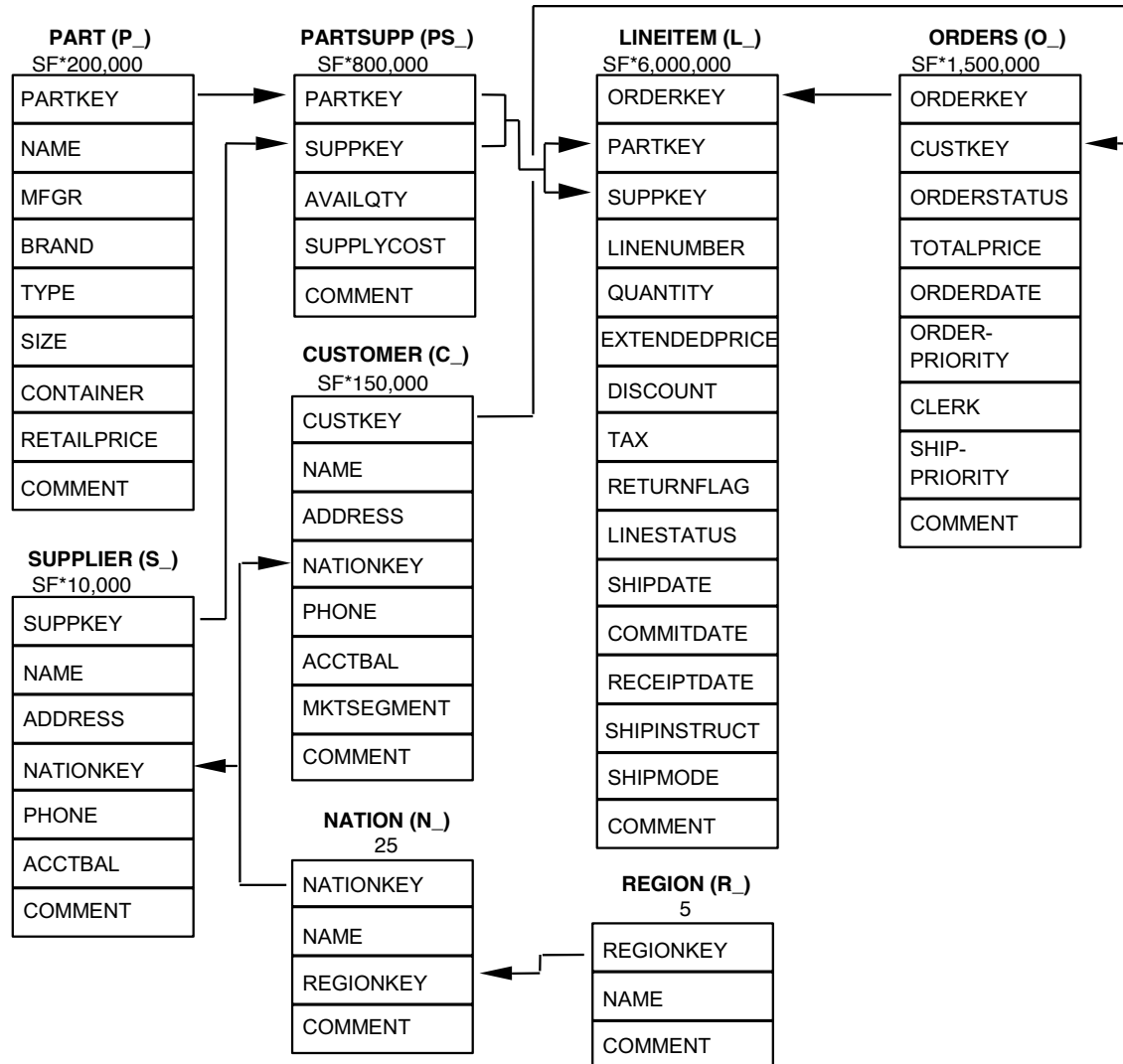
Observation:

Joins are relatively expensive
OLAP queries frequently involve joins

Solution: denormalize

What's normalization again?
Why normalize to begin with?
Fundamentally a time-space tradeoff
How much to denormalize?
What about consistency?

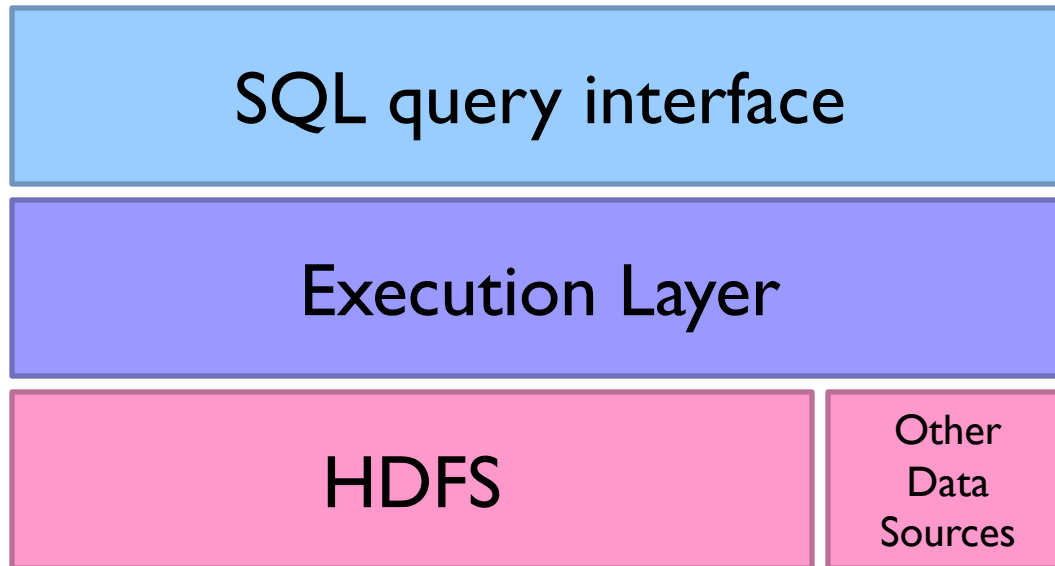
Denormalization Opportunities?



“Denormalizing the snowflake”

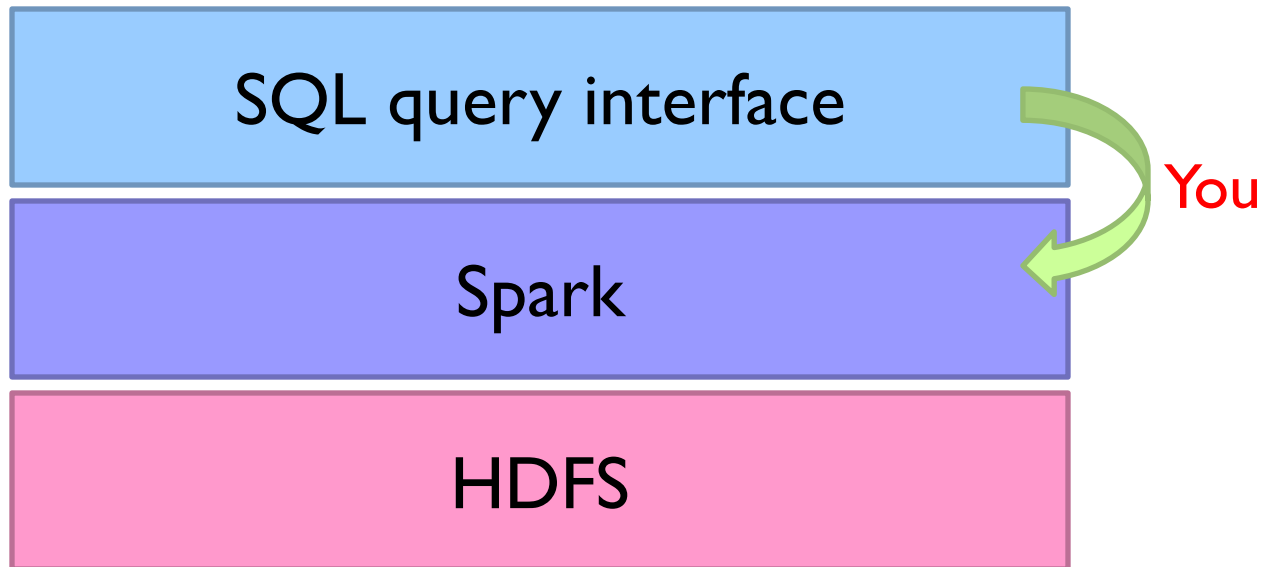
What's the assignment?

SQL-on-Hadoop

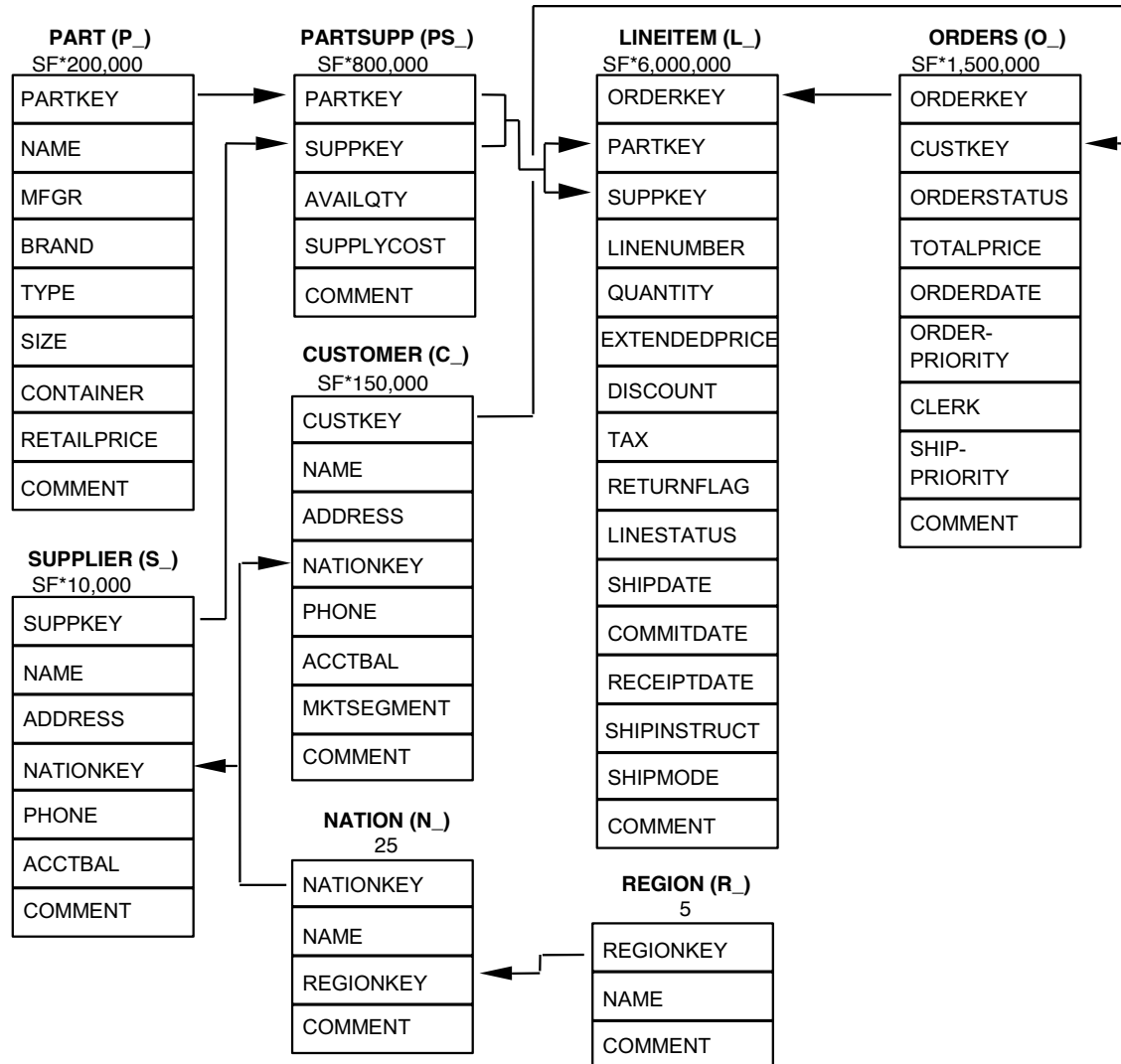


What's the assignment?

SQL-on-Hadoop



What's the assignment?



What's the assignment?

```
select
  l_returnflag,
  l_linestatus,
  sum(l_quantity) as sum_qty,
  sum(l_extendedprice) as sum_base_price,
  sum(l_extendedprice*(1-l_discount)) as sum_disc_price,
  sum(l_extendedprice*(1-l_discount)*(1+l_tax)) as sum_charge,
  avg(l_quantity) as avg_qty,
  avg(l_extendedprice) as avg_price,
  avg(l_discount) as avg_disc,
  count(*) as count_order
from lineitem
where
  l_shipdate = 'YYYY-MM-DD'
group by l_returnflag, l_linestatus;
```

SQL query  Raw Spark program

Your task...

input parameter 



Questions?