CSE 487/587
Data Intensive Computing

Lecture 9: Datawarehouse & Hive

Vipin Chaudhary

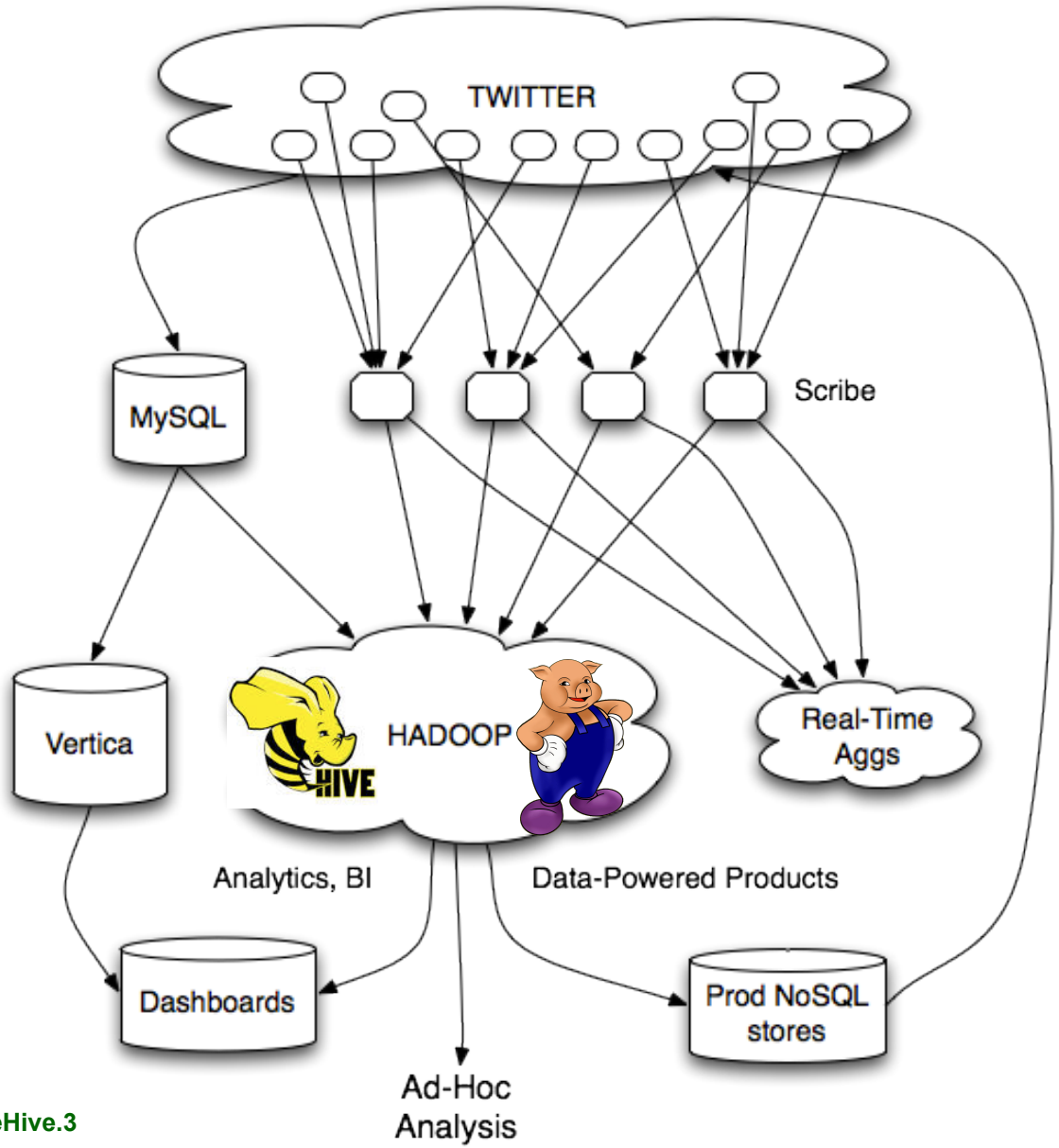
vipin@buffalo.edu

716.645.4740
305 Davis Hall

Overview of Today's Lecture

- Datawarehouse
  - Mapreduce algorithms for processing relational data
  - Evolving roles of relational databases and Mapreduce
  - Thanks to Jimmy Lin for information
- Hive (mostly from Facebook paper)
- Midterm

Analytics



Database Workloads

- OLTP (online transaction processing)
  - Typical applications: e-commerce, banking, airline reservations
  - User facing: real-time, low latency, highly-concurrent
  - Tasks: relatively small set of “standard” transactional queries
  - Data access pattern: random reads, updates, writes (involving relatively small amounts of data)
- OLAP (online analytical processing)
  - Typical applications: business intelligence, data mining
  - Back-end processing: batch workloads, less concurrency
  - Tasks: complex analytical queries, often ad hoc
  - Data access pattern: table scans, large amounts of data per query

One Database or Two?

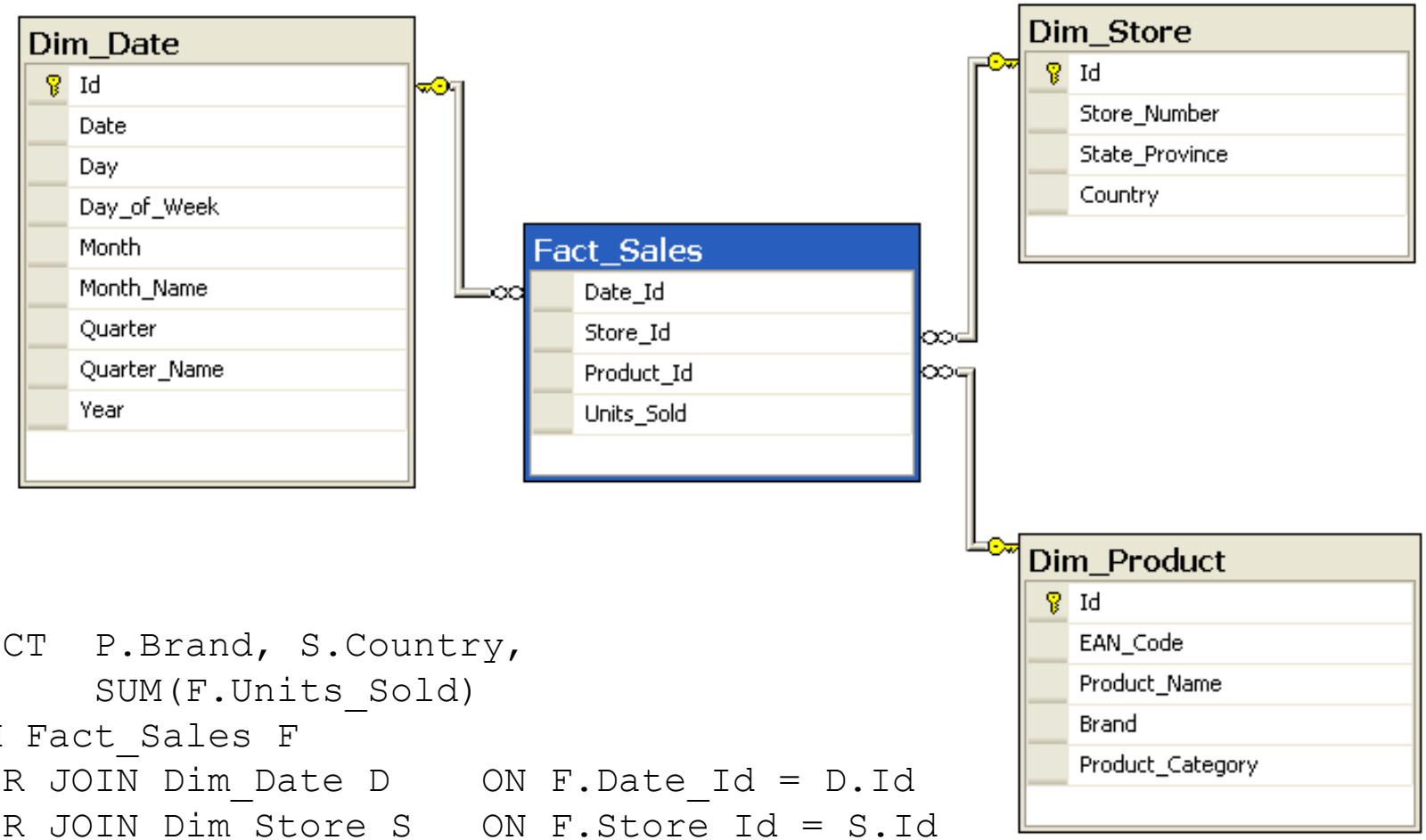
- Downsides of co-existing OLTP and OLAP workloads
  - Poor memory management
  - Conflicting data access patterns
  - Variable latency

- Solution: separate databases
  - User-facing OLTP database for high-volume transactions
  - Data warehouse for OLAP workloads
  - How do we connect the two?

OLTP/OLAP Architecture/Integration

- OLTP database for user-facing transactions
- Extract-Transform-Load (ETL)
  - Extract records from source
  - Transform: clean data, check integrity, aggregate, etc.
  - Load into OLAP database
- OLAP database for data warehousing

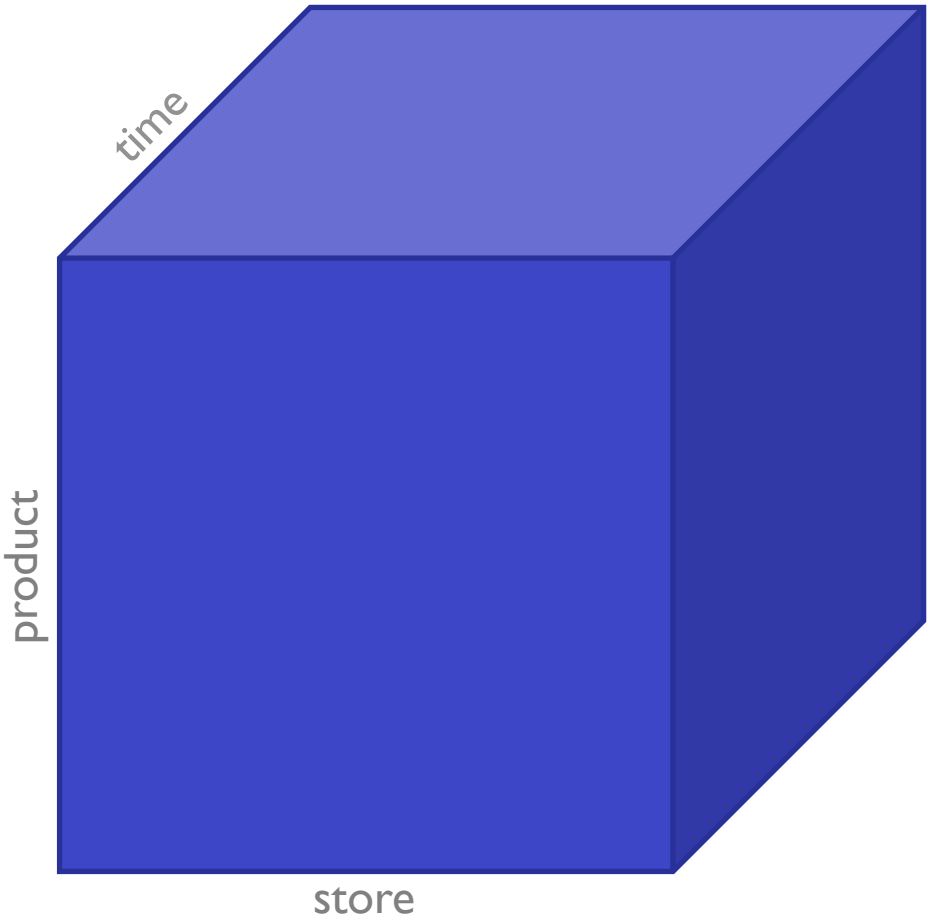
Structure of Data Warehouses




```
SELECT  P.Brand, S.Country,
        SUM(F.Units_Sold)
FROM Fact_Sales F
INNER JOIN Dim_Date D    ON F.Date_Id = D.Id
INNER JOIN Dim_Store S   ON F.Store_Id = S.Id
INNER JOIN Dim_Product P ON F.Product_Id = P.Id
WHERE D.YEAR = 1997 AND P.Product_Category = 'tv'
GROUP BY P.Brand, S.Country;
```

Source: Wikipedia (Star Schema)

OLAP Cubes



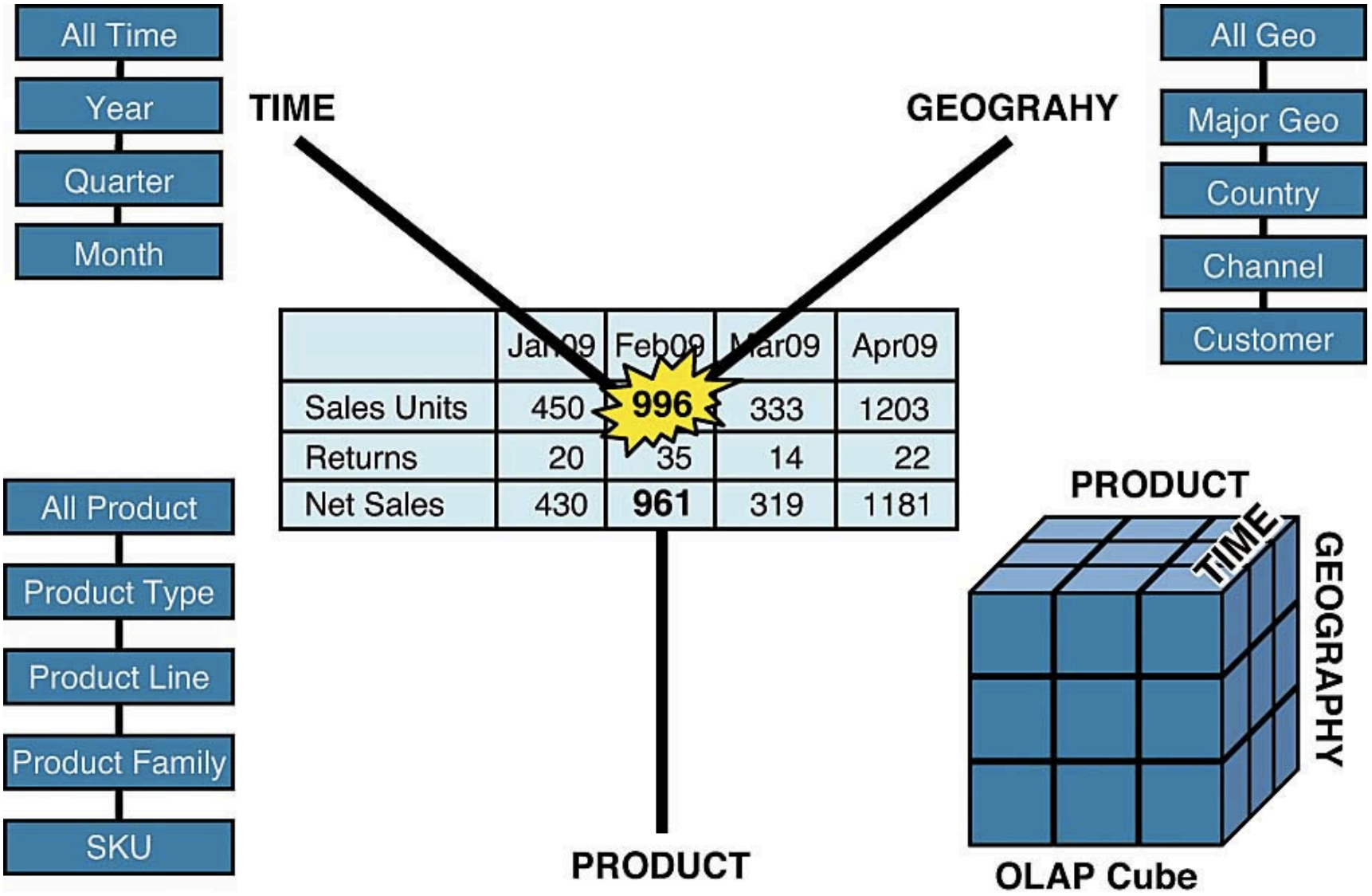
Common operations

slice and dice

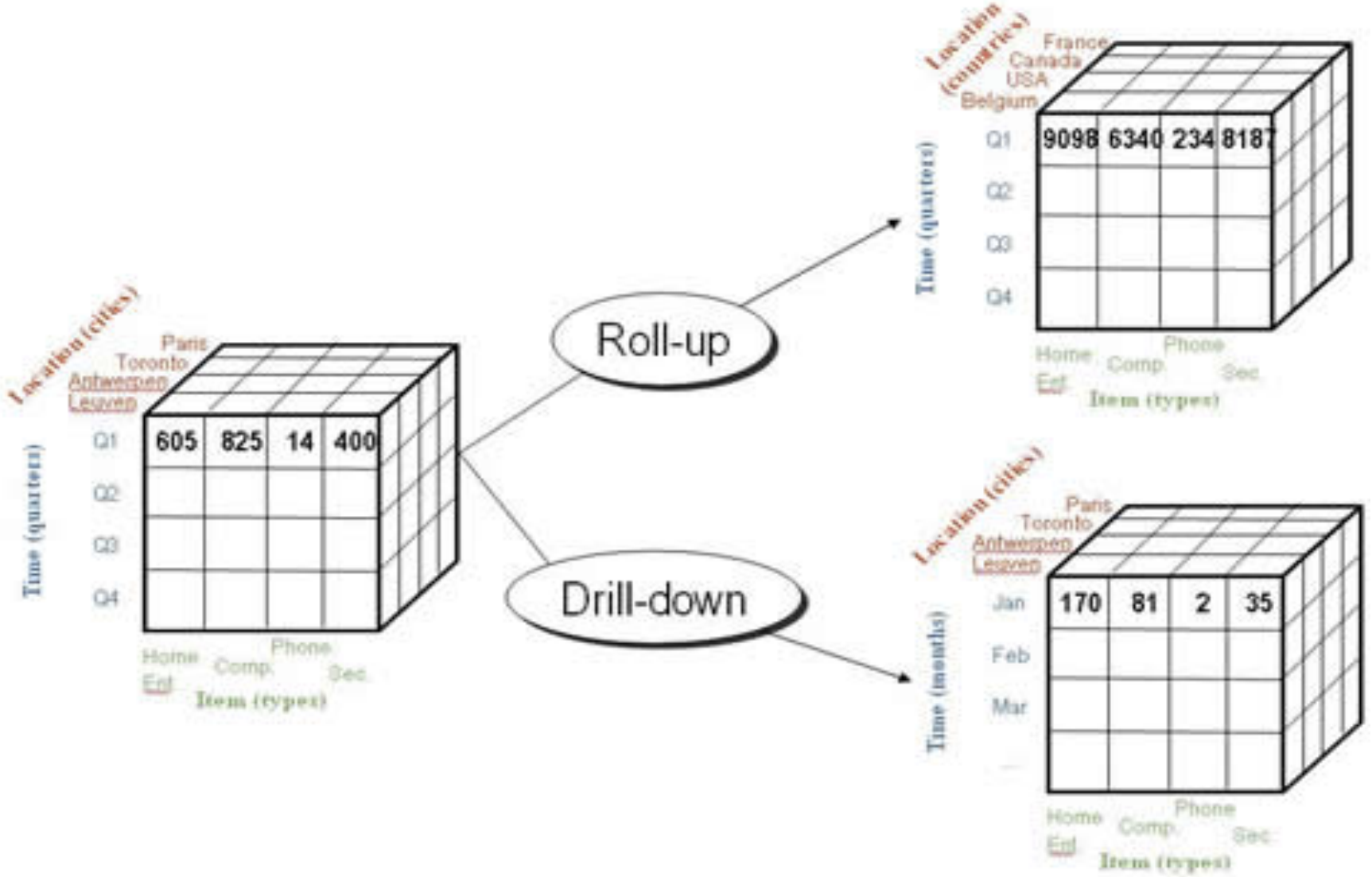
roll up/drill down

pivot

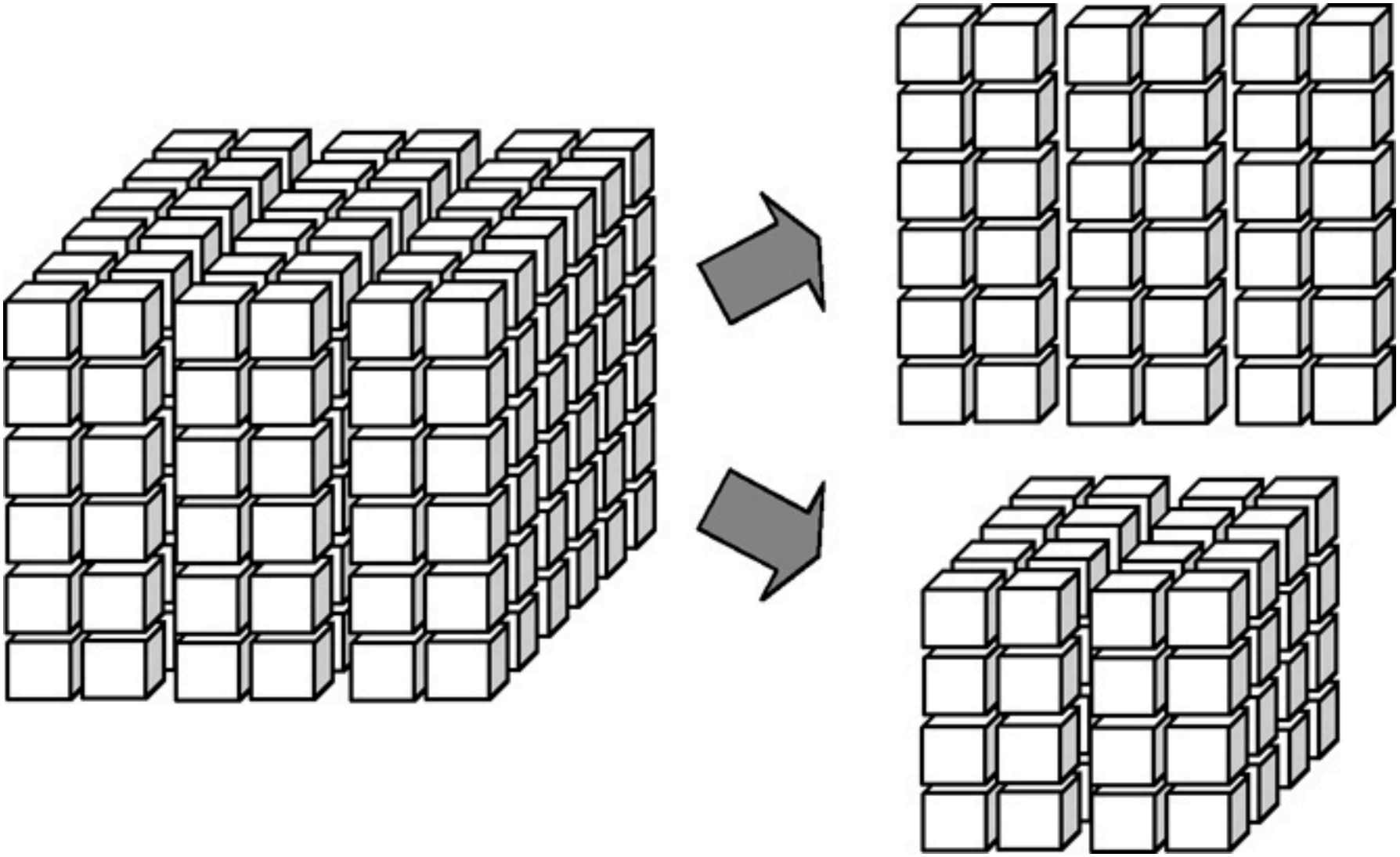
OLAP Cube Example



Roll-Up/Drill-Down



Slice and Dice



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."

What's changed?

- Dropping cost of disks
  - Cheaper to store everything than to figure out what to throw away
- Rise of social media and user-generated content
  - Large increase in data volume
- Growing maturity of data mining techniques
  - Demonstrates value of data analytics

ETL Bottleneck

- ETL is typically a nightly task:
  - What happens if processing 24 hours of data takes longer than 24 hours?
- Hadoop is perfect:
  - Ingest is limited by speed of HDFS
  - Scales out with more nodes
  - Massively parallel
  - Ability to use any processing tool
  - Much cheaper than parallel databases
  - ETL is a batch process anyway!

Mapreduce Algorithms for Processing Relational Data

Design Pattern: Secondary Sorting

- MapReduce sorts input to reducers by key
  - Values are arbitrarily ordered
- What if want to sort value also?
  - E.g., $k \rightarrow (v_1, r), (v_3, r), (v_4, r), (v_8, r)\ldots$

Secondary Sorting: Solutions

- Solution 1:
  - Buffer values in memory, then sort
  - Why is this a bad idea?
- Solution 2:
  - "Value-to-key conversion" design pattern: form composite intermediate key, (k, v_1)
  - Let execution framework do the sorting
  - Preserve state across multiple key-value pairs to handle processing

Secondary Sorting: Solution

- Mapreduce sorts intermediate key-value pairs by the keys during the shuffle and sort phase. In addition, if we need to sort by value
  - Google's MapReduce provides built-in functionality for secondary sorting guaranteeing that values arrive in sorted order
  - But Hadoop does not
- Consider example of sensor data with m sensors each taking readings on continuous basis; t is time stamp; r is actual sensor reading

```
(t1;m1; r80521 )
(t1;m2; r14209 )
(t1;m3; r76042 )
…
(t2;m1; r21823 )
(t2;m2; r66508 )
(t2;m3; r98347 )
```

- Suppose we wish to reconstruct the activity at each individual sensor over time
- MR program may map over raw data and emit sensor *id* as intermediate key, with rest of the record as value

  m1 -> (t1 ; r80521)
- This would bring all readings from same sensor together in reducer
- But Hadoop does not guarantee ordering of values, so sensor values will not be in temporal order

Value-to-Key Conversion Design Pattern

- Move part of the value into the intermediate key to form a composite key
  - Hadoop can then handle sorting
- So, instead of emitting *senor id* as key, emit the *sensor id* and *timestamp* as composite key

  (m1; t1) -> (r80521)
- Define intermediate key sort order to first sort by *sensor id* and then by *timestamp*
- Implement custom partitioner so all pairs associated with same sensor are shuffled to same reducer, the key value pairs will then be presented to reducer as sorted.

  (m1; t1) -> [(r80521)]

  (m1; t2) -> [(r21823)]

  (m1; t3) -> [(r146925)]
- Sensor readings split across multiple keys
  - Reducer needs to preserve state and keep track when readings associated with current sensor ends and next starts

Value-to-Key Conversion

Before

$k \rightarrow (v_1, r), (v_4, r), (v_8, r), (v_3, r)\ldots$

Values arrive in arbitrary order…

After

$(k, v_1) \rightarrow (v_1, r)$
$(k, v_3) \rightarrow (v_3, r)$
$(k, v_4) \rightarrow (v_4, r)$
$(k, v_8) \rightarrow (v_8, r)$
…

Values arrive in sorted order…

Process by preserving state across multiple keys

Remember to partition correctly!

Relational Databases

- A relational database is comprised of tables
- Each table represents a relation = collection of tuples (rows)
- Each tuple consists of multiple fields

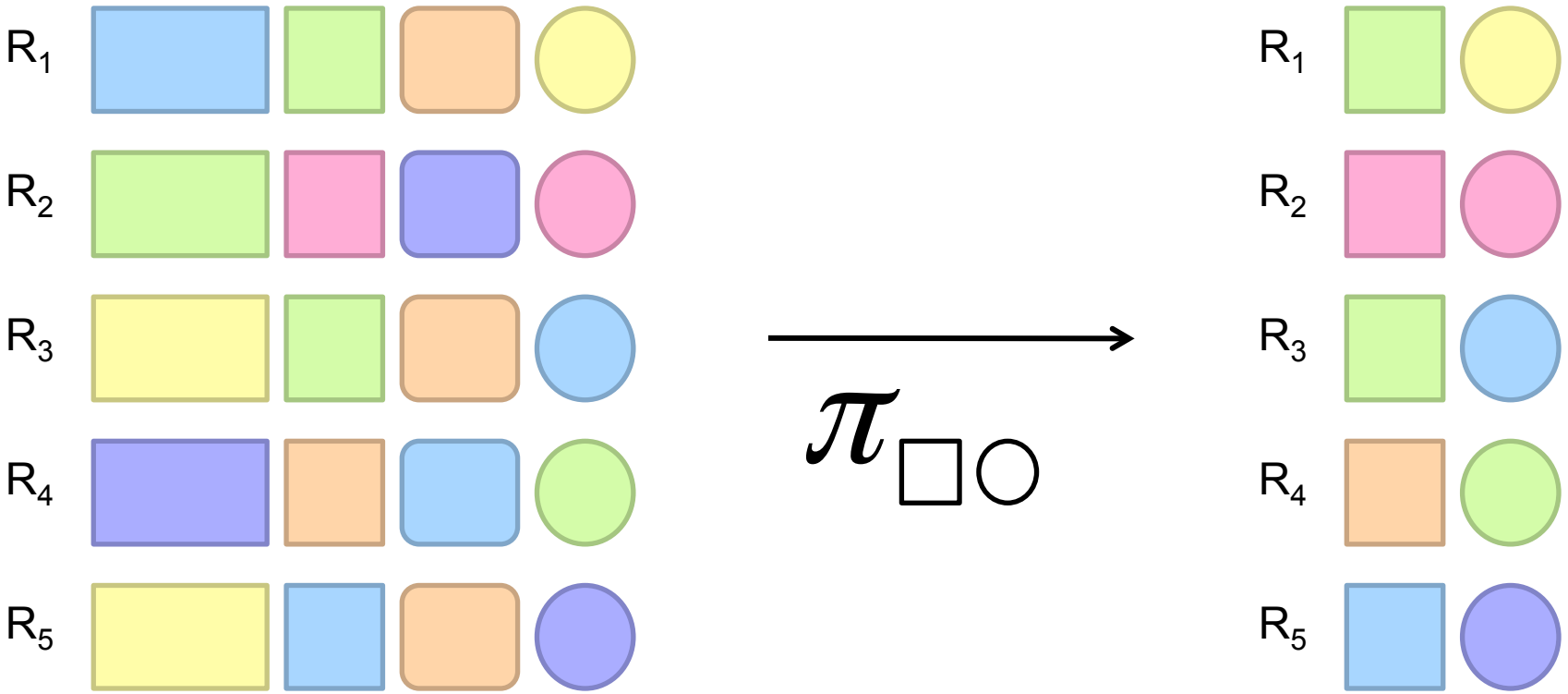
Working Scenario

- Two tables:
  - User demographics (gender, age, income, etc.)
  - User page visits (URL, time spent, etc.)
- Analyses we might want to perform:
  - Statistics on demographic characteristics
  - Statistics on page visits
  - Statistics on page visits by URL
  - Statistics on page visits by demographic characteristic
  - …

Relational Algebra

- Primitives
  - Projection (π)
  - Selection (σ)
  - Cartesian product ($\times$)
  - Set union ($\cup$)
  - Set difference (-)
  - Rename (ρ)
- Other operations
  - Join ($\bowtie$)
  - Group by… aggregation
  - …

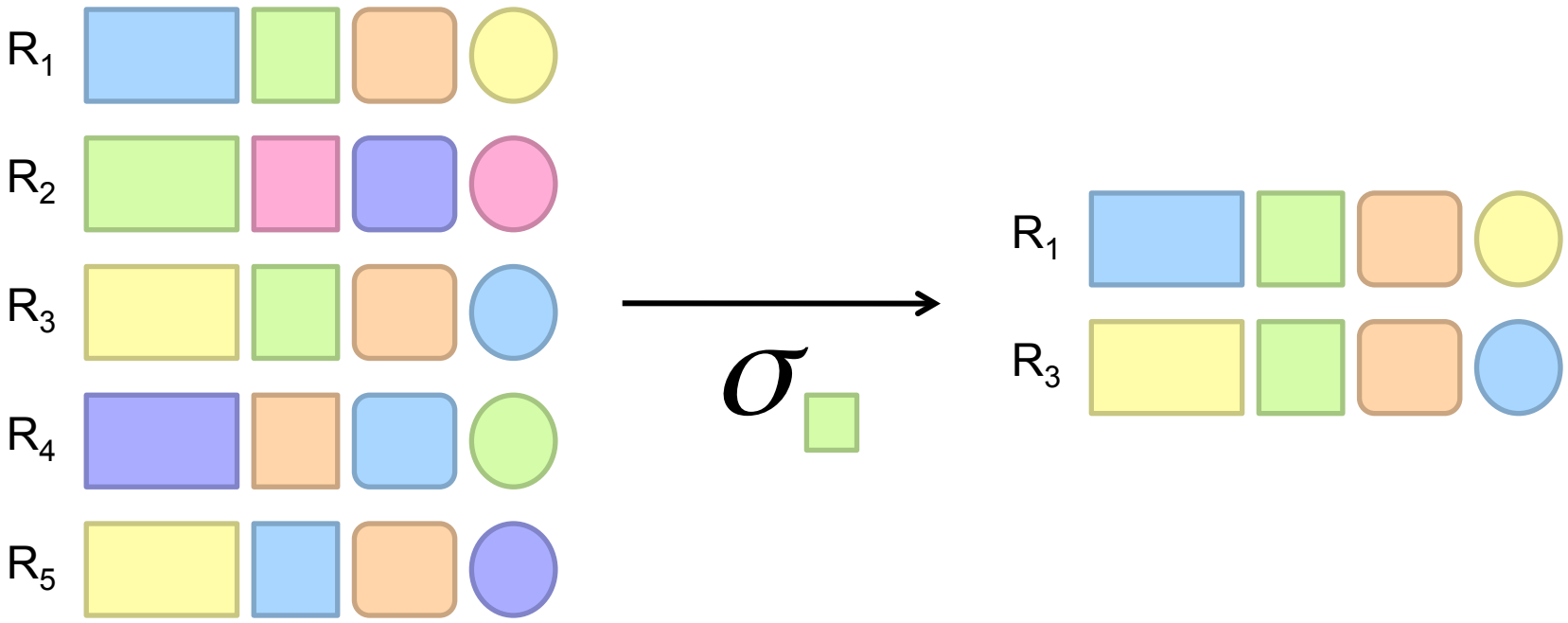
Projection



Projection in MapReduce

- Easy!
  - Map over tuples, emit new tuples with appropriate attributes
  - No reducers, unless for regrouping or resorting tuples
  - Alternatively: perform in reducer, after some other processing
- Basically limited by HDFS streaming speeds
  - Speed of encoding/decoding tuples becomes important
  - Take advantage of compression when available
  - Semistructured data? No problem!

Selection



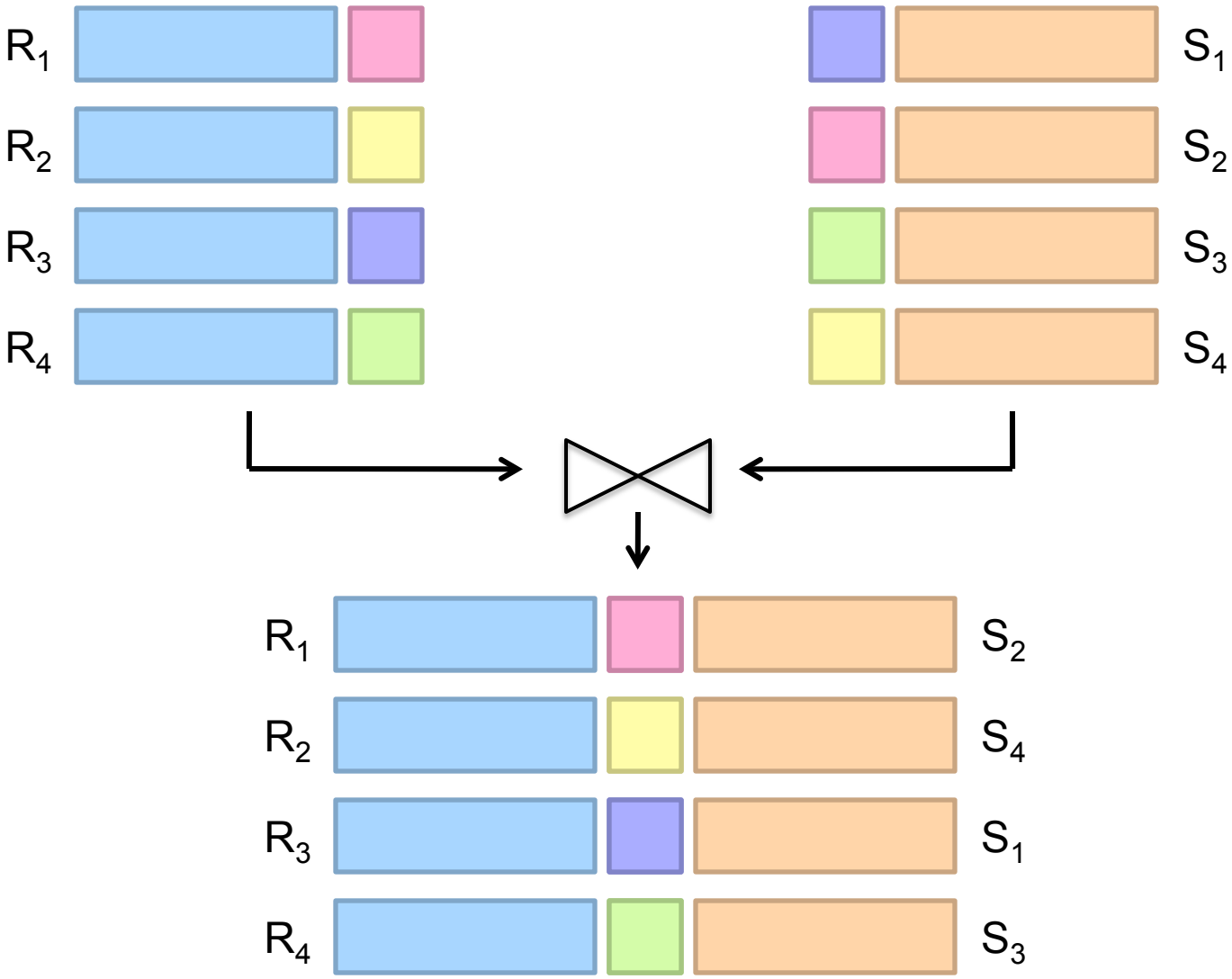
Selection in MapReduce

- Easy!
  - Map over tuples, emit only tuples that meet criteria
  - No reducers, unless for regrouping or resorting tuples
  - Alternatively: perform in reducer, after some other processing
- Basically limited by HDFS streaming speeds
  - Speed of encoding/decoding tuples becomes important
  - Take advantage of compression when available
  - Semistructured data? No problem!

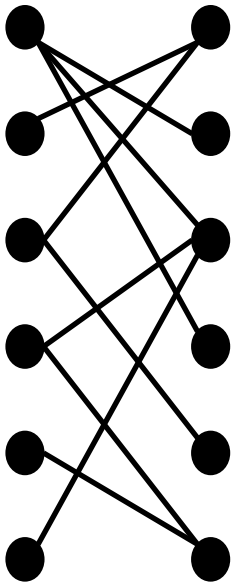
Group by… Aggregation

- Example: What is the average time spent per URL?
- In SQL:
  - SELECT url, AVG(time) FROM visits GROUP BY url
- In MapReduce:
  - Map over tuples, emit time, keyed by url
  - Framework automatically groups values by keys
  - Compute average in reducer
  - Optimize with combiners

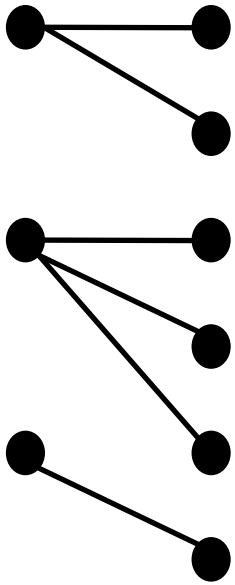
Relational Joins



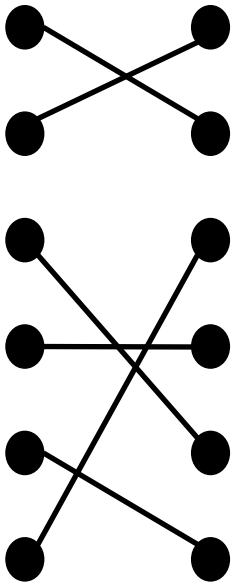
Types of Relationships



Many-to-Many



One-to-Many



One-to-One

Join Algorithms in MapReduce

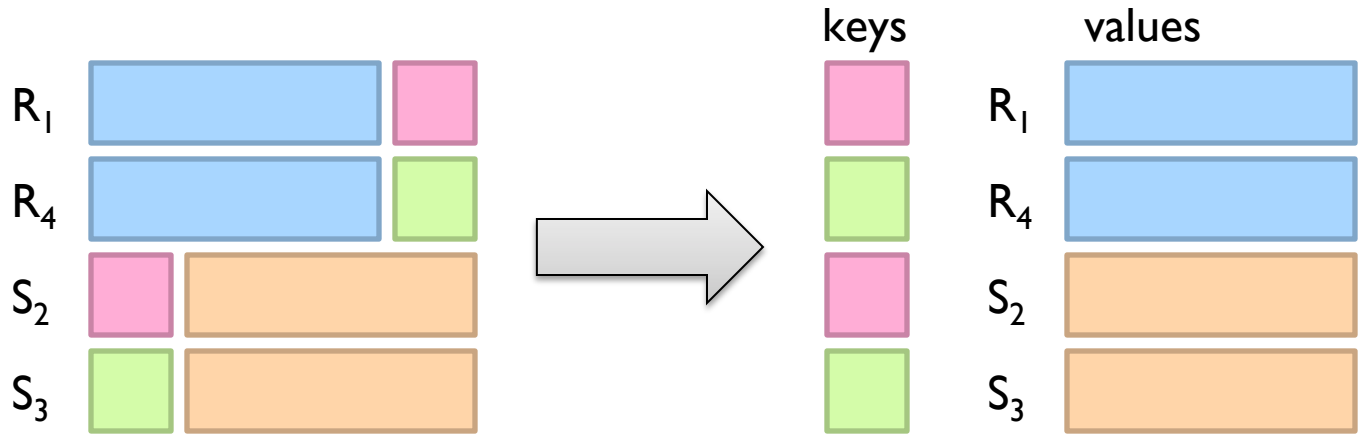
- Reduce-side join
- Map-side join
- In-memory join
  - Striped variant
  - Memcached variant

Reduce-side Join

- Basic idea: group by join key
  - Map over both sets of tuples
  - Emit tuple as value with join key as the intermediate key
  - Execution framework brings together tuples sharing the same key
  - Perform actual join in reducer
  - Similar to a “sort-merge join” in database terminology
- Two variants
  - 1-to-1 joins
  - 1-to-many and many-to-many joins

Reduce-side Join: 1-to-1

Map

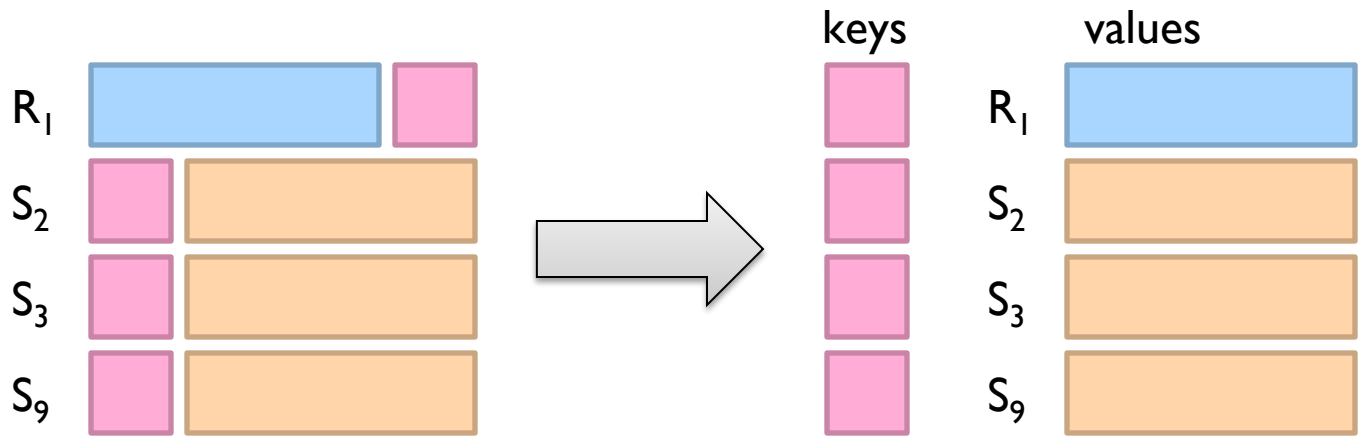


Reduce

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

Reduce-side Join: 1-to-many

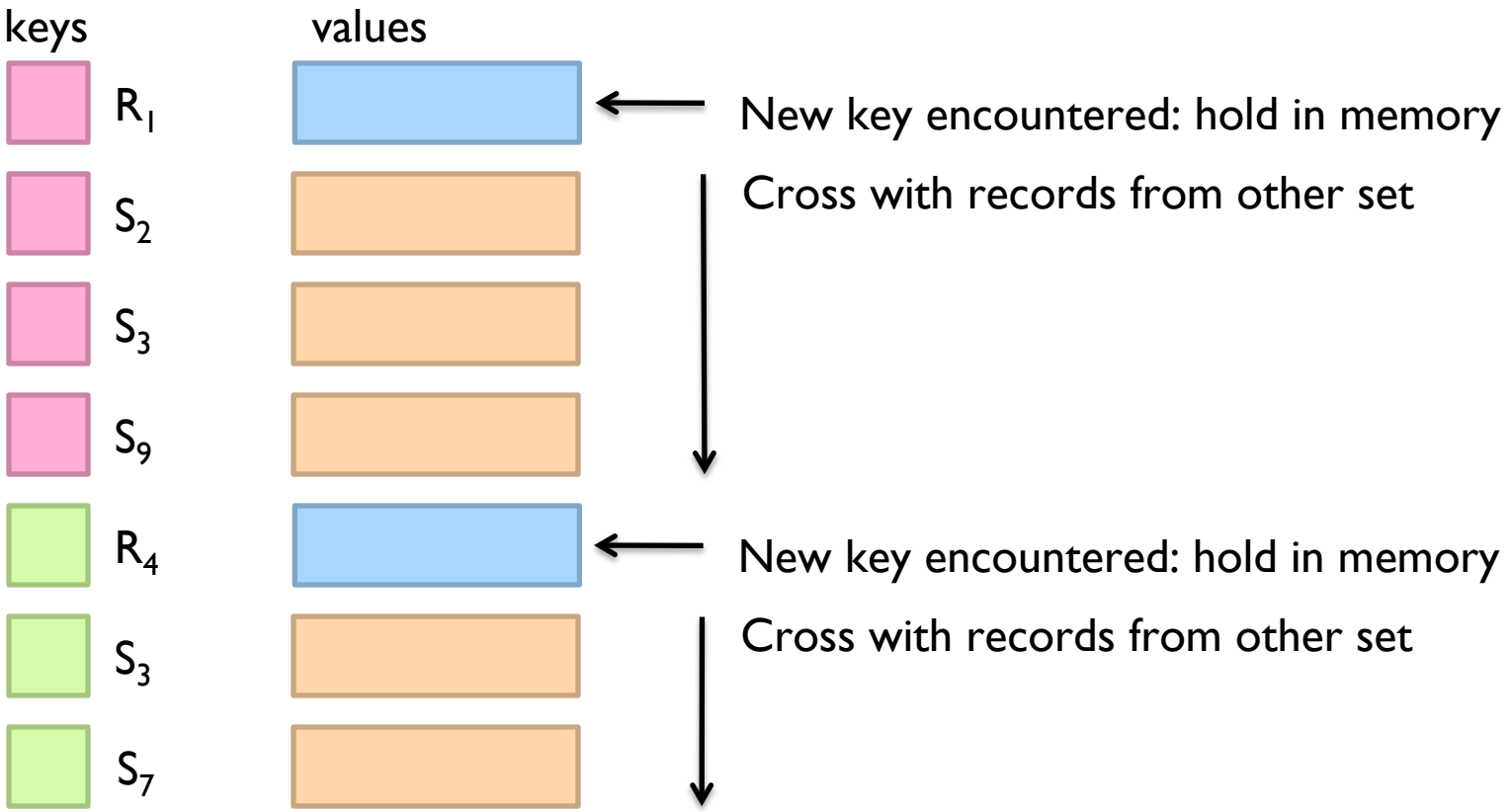
Map



Reduce

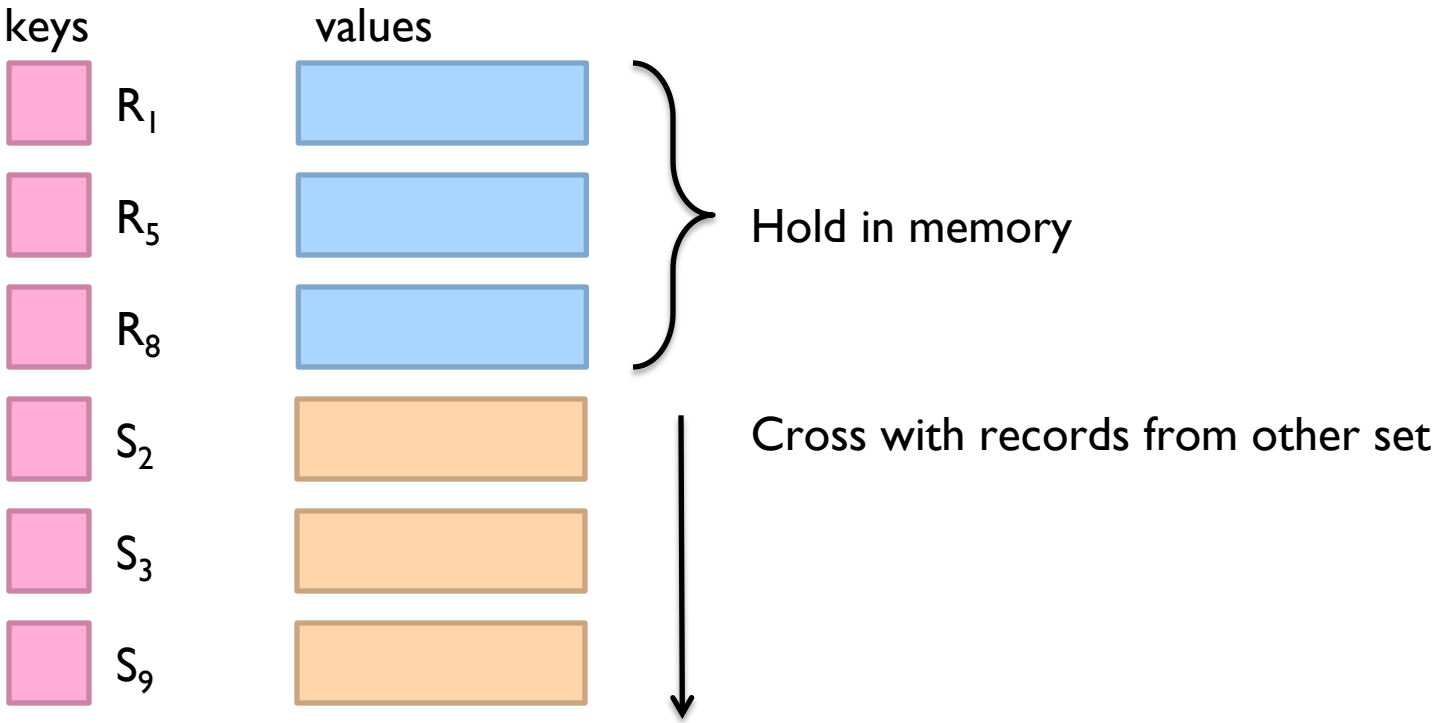
Reduce-side Join: V-to-K Conversion

In reducer…



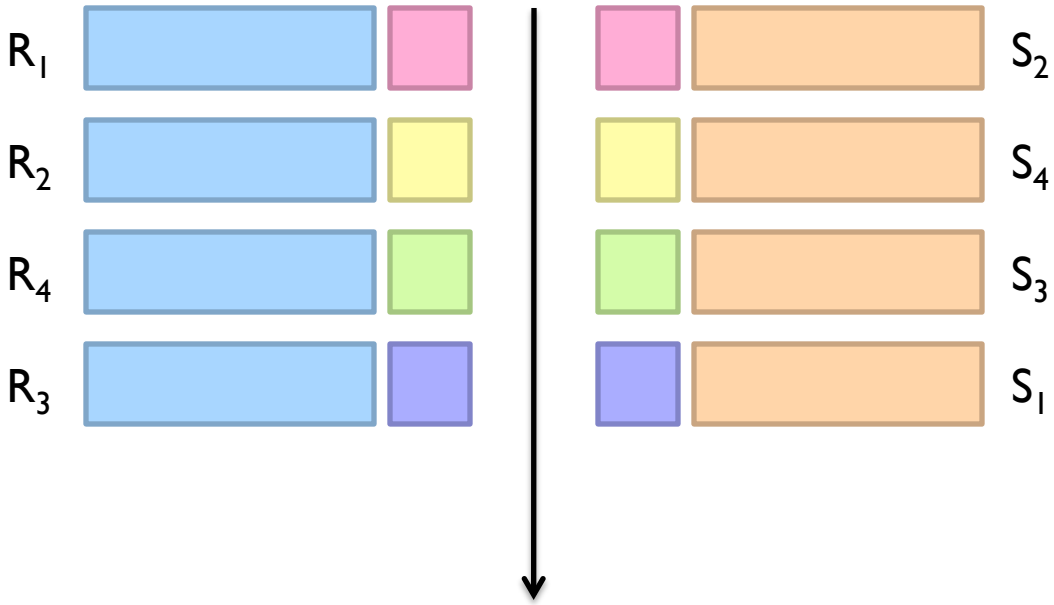
Reduce-side Join: many-to-many

In reducer…



Map-side Join: Basic Idea

Assume two datasets are sorted by the join key:



A sequential scan through both datasets to join
(called a "merge join" in database terminology)

Map-side Join: Parallel Scans

- If datasets are sorted by join key, join can be accomplished by a scan over both datasets
- How can we accomplish this in parallel?
  - Partition and sort both datasets in the same manner
- In MapReduce:
  - Map over one dataset, read from other corresponding partition
  - No reducers necessary (unless to repartition or resort)
- Consistently partitioned datasets: realistic to expect?

In-Memory Join

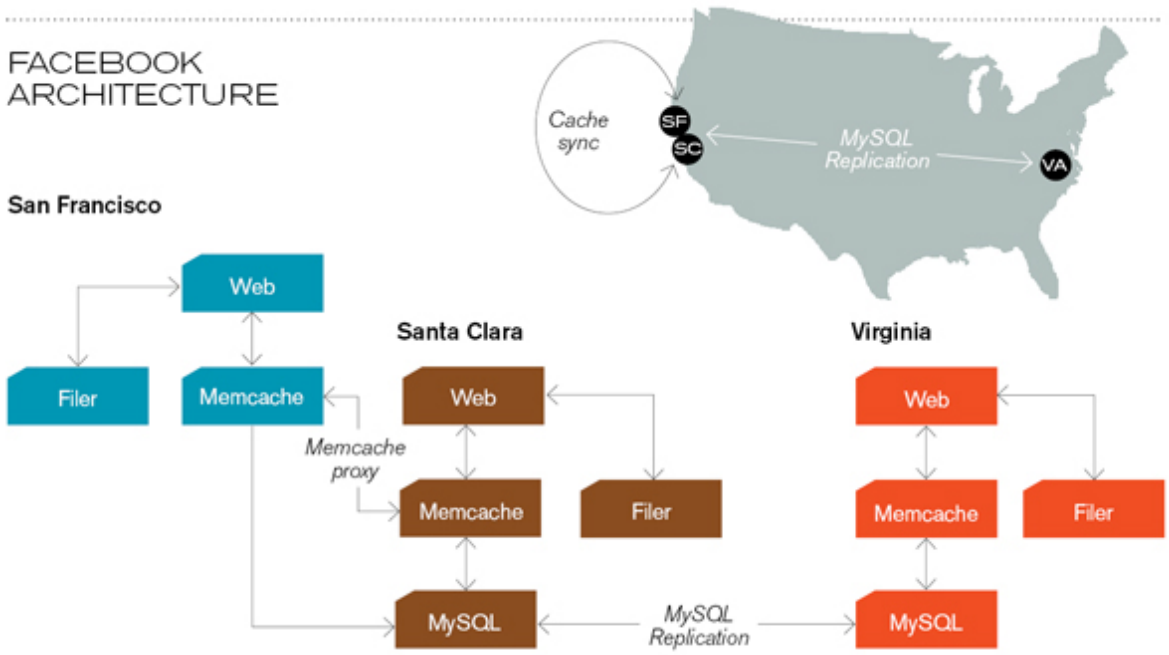
- Basic idea: load one dataset into memory, stream over other dataset
  - Works if R << S and R fits into memory
  - Called a “hash join” in database terminology
- MapReduce implementation
  - Distribute R to all nodes
  - Map over S, each mapper loads R in memory, hashed by join key
  - For every tuple in S, look up join key in R
  - No reducers, unless for regrouping or resorting tuples

In-Memory Join: Variants

- Striped variant:
  - R too big to fit into memory?
  - Divide R into R_1, R_2, R_3, … s.t. each R_n fits into memory
  - Perform in-memory join: $\forall n$, $R_n \bowtie S$
  - Take the union of all join results
- Memcached join:
  - Load R into memcached
  - Replace in-memory hash lookup with memcached lookup

Memcached

Circa 2008 Architecture



Caching servers: 15 million requests per second, 95% handled by memcache (15 TB of RAM)

Database layer: 800 eight-core Linux servers running MySQL (40 TB user data)

Source: Technology Review (July/August, 2008)

Memcached Join

- Memcached join:
  - Load R into memcached
  - Replace in-memory hash lookup with memcached lookup
- Capacity and scalability?
  - Memcached capacity >> RAM of individual node
  - Memcached scales out with cluster
- Latency?
  - Memcached is fast (basically, speed of network)
  - Batch requests to amortize latency costs

Source: See tech report by Lin et al. (2009)

Which join to use?

- In-memory join > map-side join > reduce-side join
  - Why?
- Limitations of each?
  - In-memory join: memory
  - Map-side join: sort order and partitioning
  - Reduce-side join: general purpose

Processing Relational Data: Summary

- MapReduce algorithms for processing relational data:
  - Group by, sorting, partitioning are handled automatically by shuffle/sort in MapReduce
  - Selection, projection, and other computations (e.g., aggregation), are performed either in mapper or reducer
  - Multiple strategies for relational joins
- Complex operations require multiple MapReduce jobs
  - Example: top ten URLs in terms of average time spent
  - Opportunities for automatic optimization

Need for High-Level Languages

- Hadoop is great for large-data processing!
  - But writing Java programs for everything is verbose and slow
  - Data scientists don't want to write Java
- Solution: develop higher-level data processing languages
  - Hive: HQL is like SQL
  - Pig: Pig Latin is a bit like Perl

Hive and Pig

- Hive: data warehousing application in Hadoop
  - Query language is HQL, variant of SQL
  - Tables stored on HDFS with different encodings
  - Developed by Facebook, now open source

- Pig: large-scale data processing system
  - Scripts are written in Pig Latin, a dataflow language
  - Programmer focuses on data transformations
  - Developed by Yahoo!, now open source
- Common idea:
  - Provide higher-level language to facilitate large-data processing
  - Higher-level language "compiles down" to Hadoop jobs

Relational Databases vs. MapReduce

- Relational databases:
  - Multipurpose: analysis and transactions; batch and interactive
  - Data integrity via ACID transactions
  - Lots of tools in software ecosystem (for ingesting, reporting, etc.)
  - Supports SQL (and SQL integration, e.g., JDBC)
  - Automatic SQL query optimization
- MapReduce (Hadoop):
  - Designed for large clusters, fault tolerant
  - Data is accessed in “native format”
  - Supports many query languages
  - Programmers retain control over performance
  - Open source

Source: O'Reilly Blog post by Joseph Hellerstein (11/19/2008)

Design Principles of HIVE?

- A system for managing and querying unstructured data as if it were structured
  - Uses Map-Reduce for execution
  - HDFS for Storage
- Key Building Principles
  - SQL as a familiar data warehousing tool
  - Extensibility (Pluggable map/reduce scripts in the language of your choice, Rich and User Defined Data Types, User Defined Functions)
  - Interoperability (Extensible Framework to support different file and data formats)
  - Performance

Type System

- Primitive types
  - Integers:TINYINT, SMALLINT, INT, BIGINT.
  - Boolean: BOOLEAN.
  - Floating point numbers: FLOAT, DOUBLE .
  - String: STRING.
- Complex types
  - Structs: struct<file-name: field-type, …>
  - Lists: list<element-type>
  - Associative Arrays: map<key-type, value-type>
- List<map<string, struct<p1:int, p2:int>>>
  - List of associative arrays that map strings to structs that in turn contain two integer fields named p1 and p2

Data Model- Tables

- Tables
  - Analogous to tables in relational DBs.
  - Each table has corresponding directory in HDFS.
  - Example
    - Page view table name – cse587
    - HDFS directory is /user/hive/datawarehouse/cse587
- Example: List<map<string, struct<p1:int, p2:int>>>

```
CREATE TABLE t1 (st string, fl float, li
list<map<string,struct<p1:int, p2:int>>>);
```

t1.li[0] gives the first element of the list

t1.li[0]['key'] gives the struct associated with 'key' in that associative array

the p2 field of this struct can be accessed by t1.li[0]['key'].p2.

Data Model

- Tables – A table is stored in a directory in hdfs.
- Partitions – A partition of the table is stored in a subdirectory within a table's directory.
- Buckets – A bucket is stored in a file within the partition's or table's directory depending on whether the table is a partitioned table or not.

- table test_table gets mapped to `<warehouse_root_directory>/test_table` in hdfs.
- The warehouse_root_directory is specified by the `hive.metastore.warehouse.dir` configuration parameter in `hive-site.xml`.
- By default this parameter's value is set to `/user/hive/warehouse`.

HiveQL

- Subset of SQL
- Meta-data queries
- Limited equality and join predicates

```
SELECT t1.a1 as c1, t2.b1 as c2
FROM t1 JOIN t2 ON (t1.a2 = t2.b2);
```

instead of the more traditional

```
SELECT t1.a1 as c1, t2.b1 as c2
FROM t1, t2
WHERE t1.a2 = t2.b2;
```

HiveQL

- Cannot insert into existing table or data partition
- All inserts overwrite existing data

```
INSERT OVERWRITE TABLE t1
SELECT * FROM t2;
```

- Plugged in HiveQL queries

```
FROM (
MAP doctext USING 'python wc_mapper.py' AS (word,cnt)
FROM docs
CLUSTER BY word
) a
REDUCE word, cnt USING 'python wc_reduce.py';
```

HiveQL

- Distribution criteria between the mappers and the reducers
  - provide data to the reducers such that it is sorted on a set of columns that are different from the ones that are used to do the distribution
- All the actions in a session need to be ordered by time

```
FROM (
FROM session_table
SELECT sessionid, tstamp, data
DISTRIBUTE BY sessionid SORT BY tstamp
) a
REDUCE sessionid, tstamp, data USING'session_reducer.sh';
```

HiveQL

```
FROM t1
    INSERT OVERWRITE TABLE t2
    SELECT t3.c2, count(1)
    FROM t3
    WHERE t3.c1 <= 20
    GROUP BY t3.c2

    INSERT OVERWRITE DIRECTORY '/output_dir'
    SELECT t3.c2, avg(t3.c1)
    FROM t3
    WHERE t3.c1 > 20 AND t3.c1 <= 30
    GROUP BY t3.c2

    INSERT OVERWRITE LOCAL DIRECTORY '/home/dir'
    SELECT t3.c2, sum(t3.c1)
    FROM t3
    WHERE t3.c1 > 30
    GROUP BY t3.c2;
```

Partitioning

```
CREATE TABLE test_part(c1 string, c2 int)
PARTITIONED BY (ds string, hr int);
```

- Table partitions will be stored in `/user/hive/warehouse/test_part` hdfs directory
- A partition exists for every distinct value of ds and hr specified by the user

```
INSERT OVERWRITE TABLE
test_part PARTITION(ds='2009-01-01', hr=12)
SELECT * FROM t;
```

- The INSERT statement also populates the partition with data from table t
- /user/hive/warehouse/test_part/ds=2009-01-01/hr=12

```
ALTER TABLE test_part
ADD PARTITION(ds='2009-02-02', hr=11);
```

- Alter table creates an empty partition /user/hive/warehouse/test_part/ds=2009-02-02/hr=11

Partitioning

```
SELECT * FROM test_part WHERE ds='2009-01-01';
```

will only scan all the files within the

`/user/hive/warehouse/test_part/ds=2009-01-01` directory

```
SELECT * FROM test_part
WHERE ds='2009-02-02' AND hr=11;
```

will only scan all the files within the `/user/hive/warehouse/test_part/ds=2009-02-02/hr=11` directory.

Buckets

- Bucket is a file within the leaf level directory of table or partition
- During table creation, user can specify
  - Number of buckets needed
  - Column on which to bucket the data
- Table that is bucketed into 32 buckets can quickly generate a 1/32 sample by choosing to look at the first bucket of data.

```
SELECT * FROM t TABLESAMPLE(2 OUT OF 32);
```

would scan the data present in the second bucket

External Tables

```
CREATE EXTERNAL TABLE test_extern(c1 string, c2 int)
LOCATION '/user/mytables/mydata';
```

- Specify that test_extern is an external table with each row comprising of two columns – c1 and c2
- Data files are stored in the location /user/mytables/mydata in hdfs
- Data is assumed to be in Hive-compatible format – no SerDe defined
- Dropping external table drops only the metadata

Serialization/Deserialization

- Generic (De)Serialzation Interface SerDe
- Uses default LazySerDe
  - rows are delimited by a newline (ascii code 13)
  - columns within a row are delimited by ctrl-A (ascii code 1)
- The SerDes are located in 'hive_contrib.jar';
- Designed to read data separated by different delimiter characters

```
CREATE TABLE test_delimited(c1 string, c2 int)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\002'
LINES TERMINATED BY '\012';
```

- data for table test_delimited uses ctrl-B (ascii code 2) as a column delimiter and uses ctrl-L(ascii code 12) as a row delimiter

Serialization/Deserialization

- RegexSerDe which enables the user to specify a regular expression to parse various columns out from a row, ex. Interpret apache logs

```
add jar 'hive_contrib.jar';
CREATE TABLE apachelog(
        host string,
        identity string,
        user string,
        time string,
        request string,
        status string,
        size string,
        referer string,
        agent string)
ROW FORMAT SERDE
        'org.apache.hadoop.hive.contrib.serde2.RegexSerDe'
WITH SERDEPROPERTIES(
        ' input.regex'  = ' ([^ ]*) ([^ ]*) ([^ ]*) (-|\\[[^\\]]*\\]) ([^
\"]*|\"[^\"]*\") (-|[0-9]*) (-|[0-9]*)(?: ([^ \"]*|\"[^\"]*\") ([^
\"]*|\"[^\"]*\"))?' ,
        'output.format.string' = '%1$s %2$s %3$s %4$s %5$s %6$s
%7$s %8$s %9$s');
```

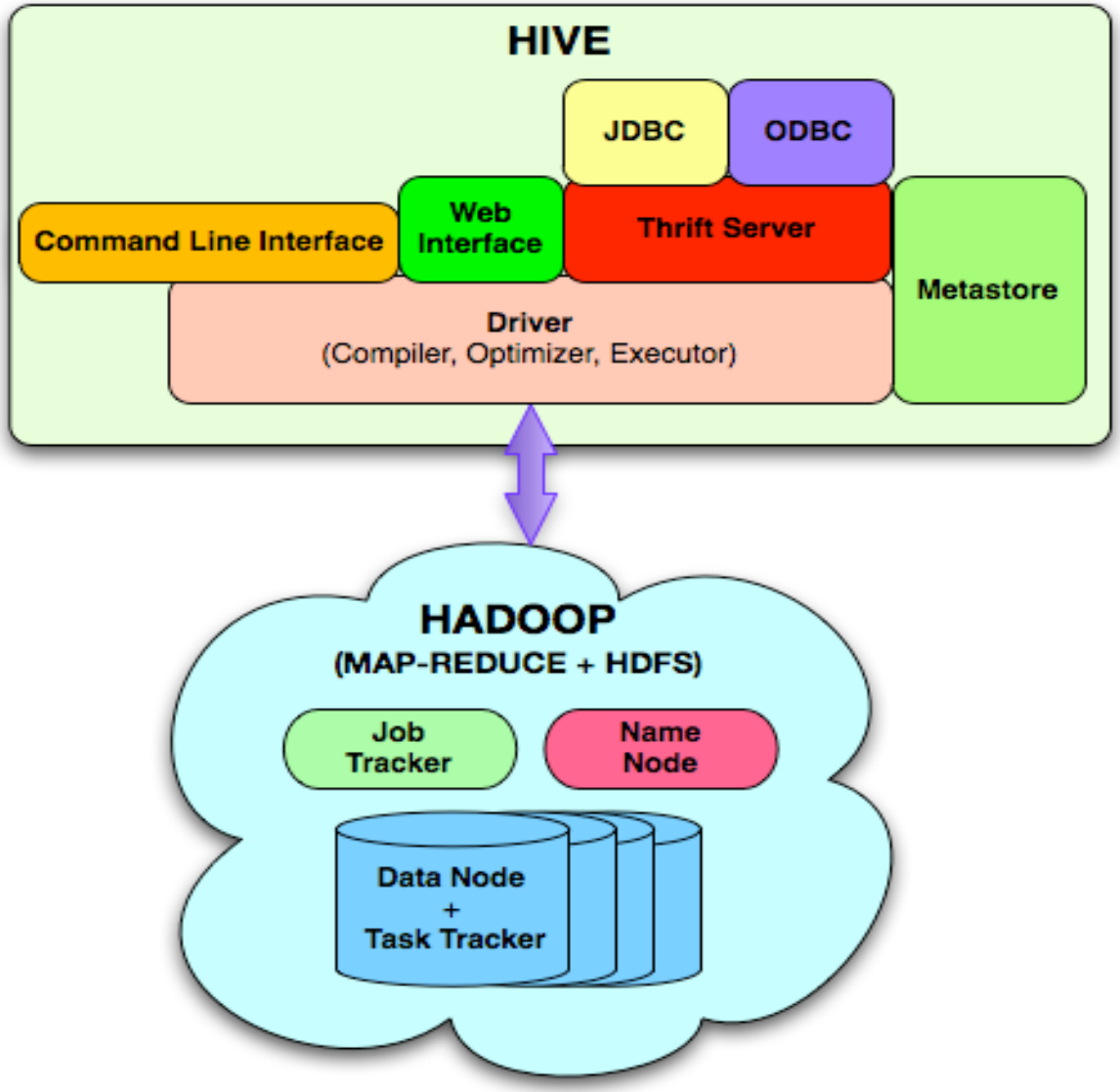
Hive File Formats

- Hive lets users store different file formats
  - `TextInputFormat` – text files
  - `SequenceFileInputFormat` – binary files
  - `RCFileInputFormat` – column oriented to improve performance when all columns not required for query
- Add own file format:

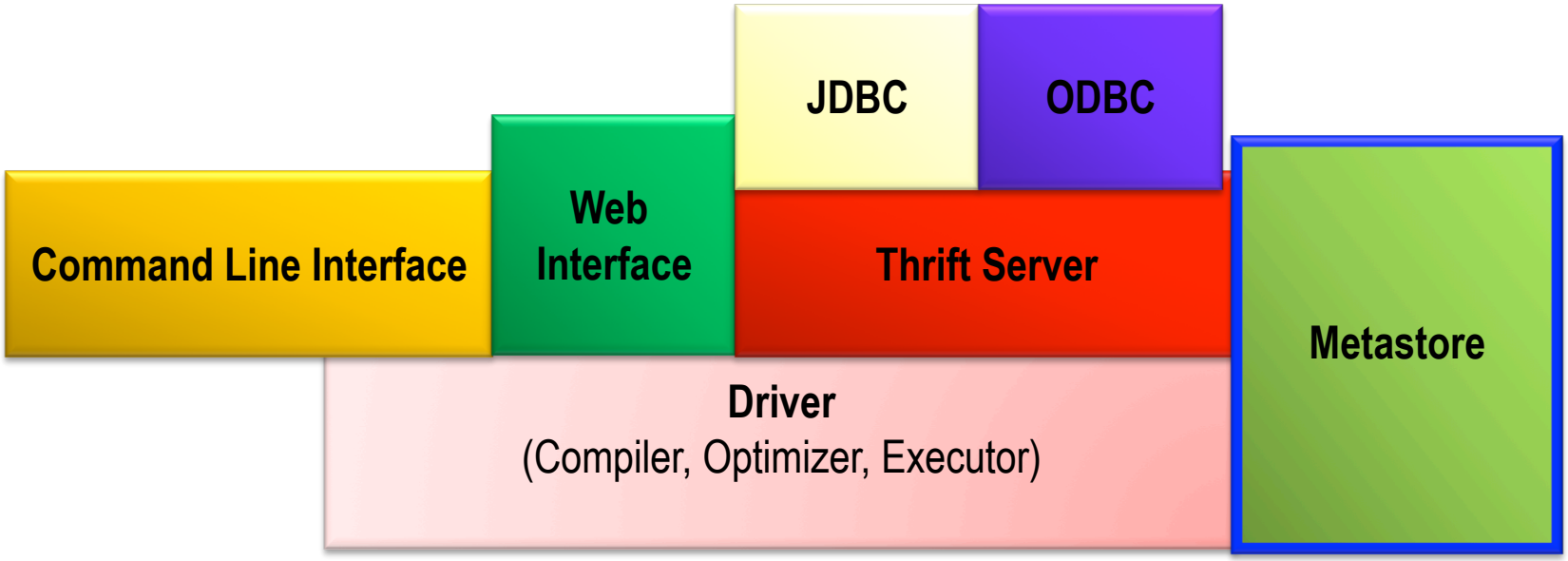
```
CREATE TABLE dest1(key INT, value STRING)
   STORED AS
     INPUTFORMAT
         'org.apache.hadoop.mapred.SequenceFileInputFormat'
     OUTPUTFORMAT
         'org.apache.hadoop.mapred.SequenceFileOutputFormat'
```

- STORED AS clause specifies the classes to be used to determine the input and output formats of the files in the table's or partition's directory.
  - any class that implements the FileInputFormat and FileOutputFormat java interfaces

System Architecture and Components

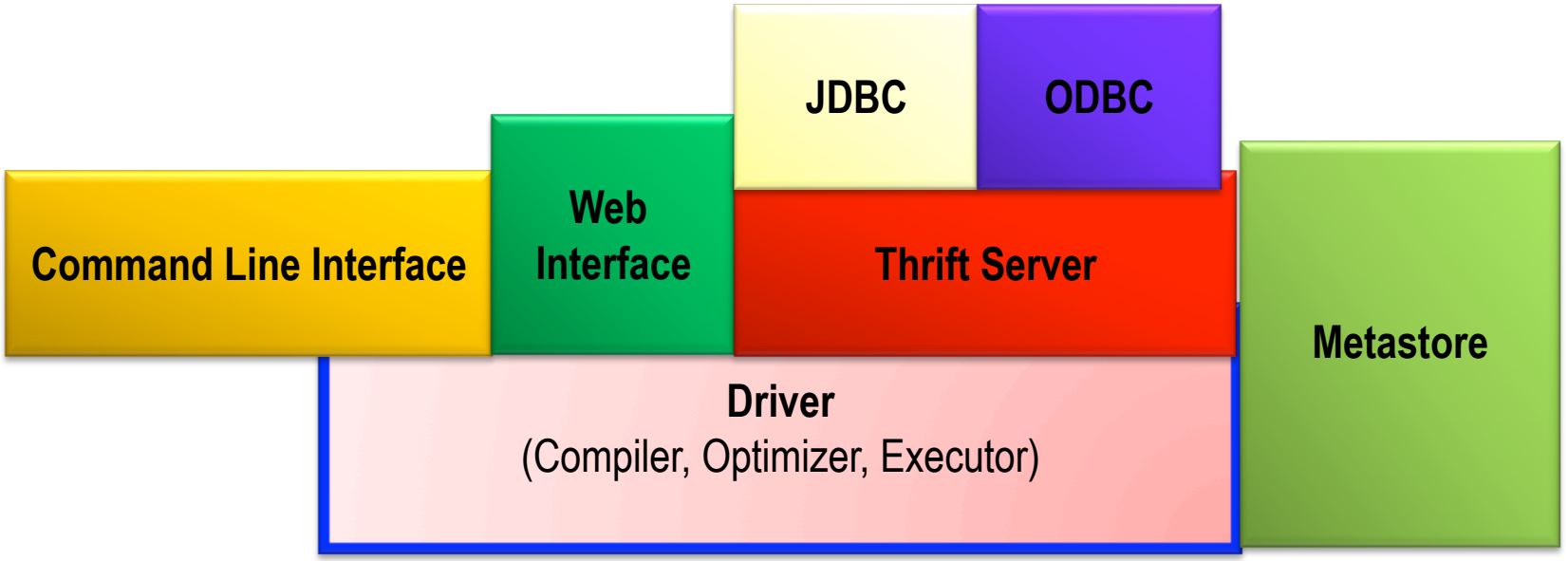


Metastore



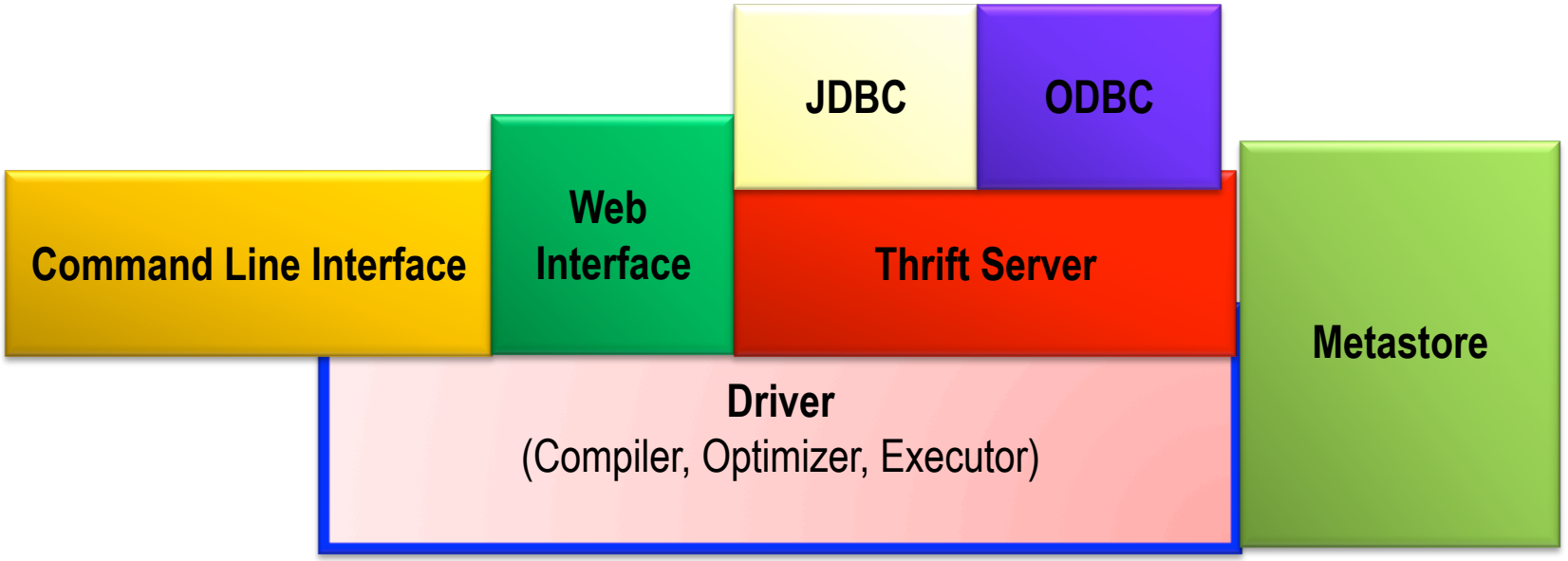
- Stores system catalog and metadata about tables, their partitions, schemas, columns and their types, table locations, etc.
- Stored on a traditional RDBMS mysql (not hdfs) to reduce latency
  - Queried or modified using thrift interface

Driver



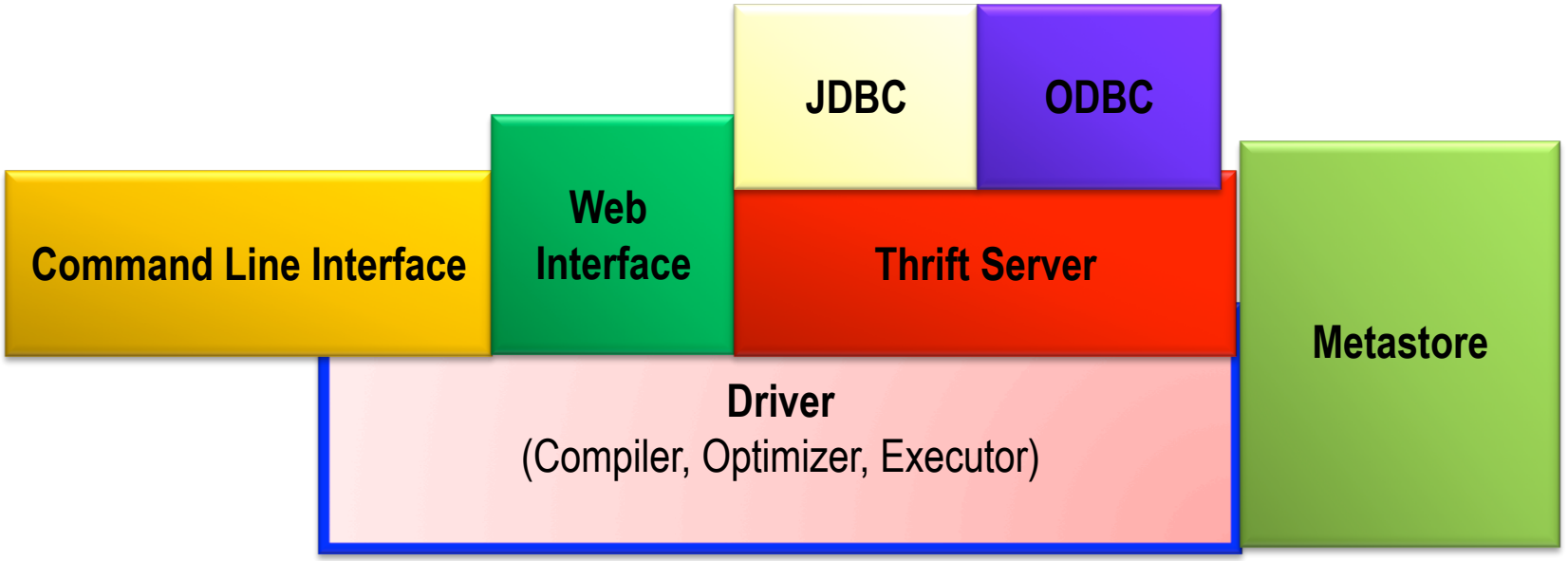
- The component that manages the lifecycle of a HiveQL statement as it moves through Hive.
- The driver also maintains a session handle and any session statistics.

Query Compiler/Optimizer



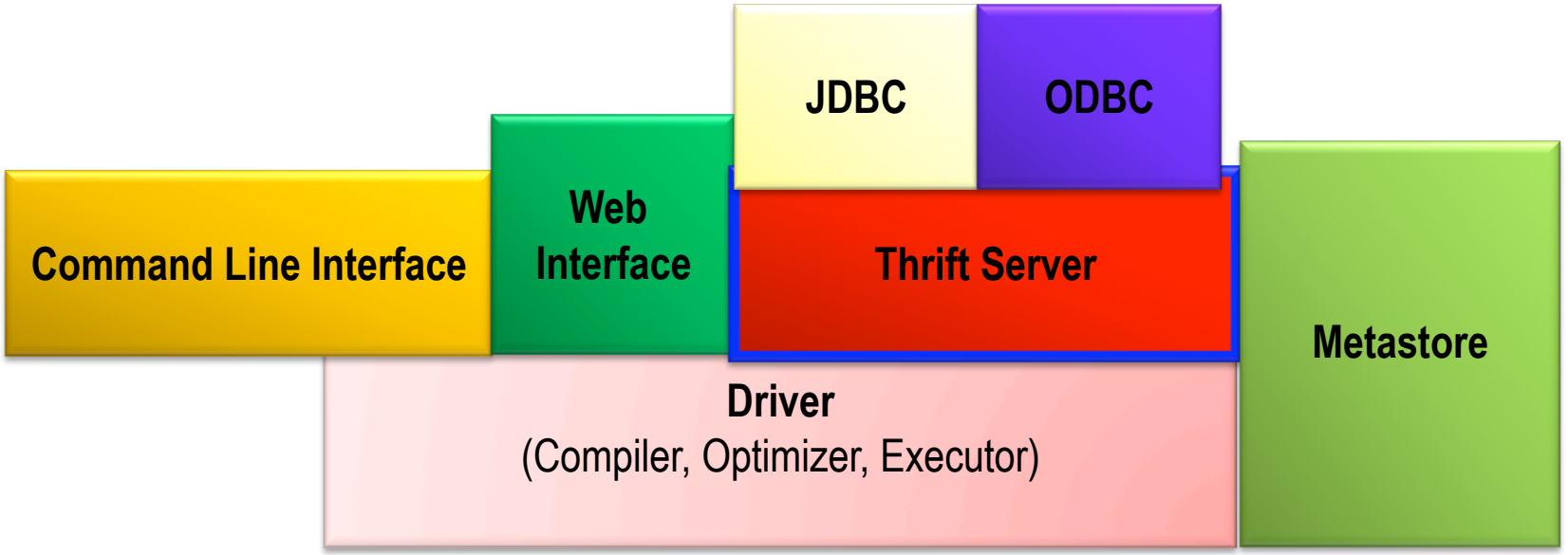
- Compiler
  - The component that compiles HiveQL into a directed acyclic graph of map/reduce tasks.
- Optimizer
  - consists of a chain of transformations such that the operator DAG resulting from one transformation is passed as input to the next transformation
  - Performs tasks like Column Pruning , Partition Pruning, Repartitioning of Data

Execution Engine



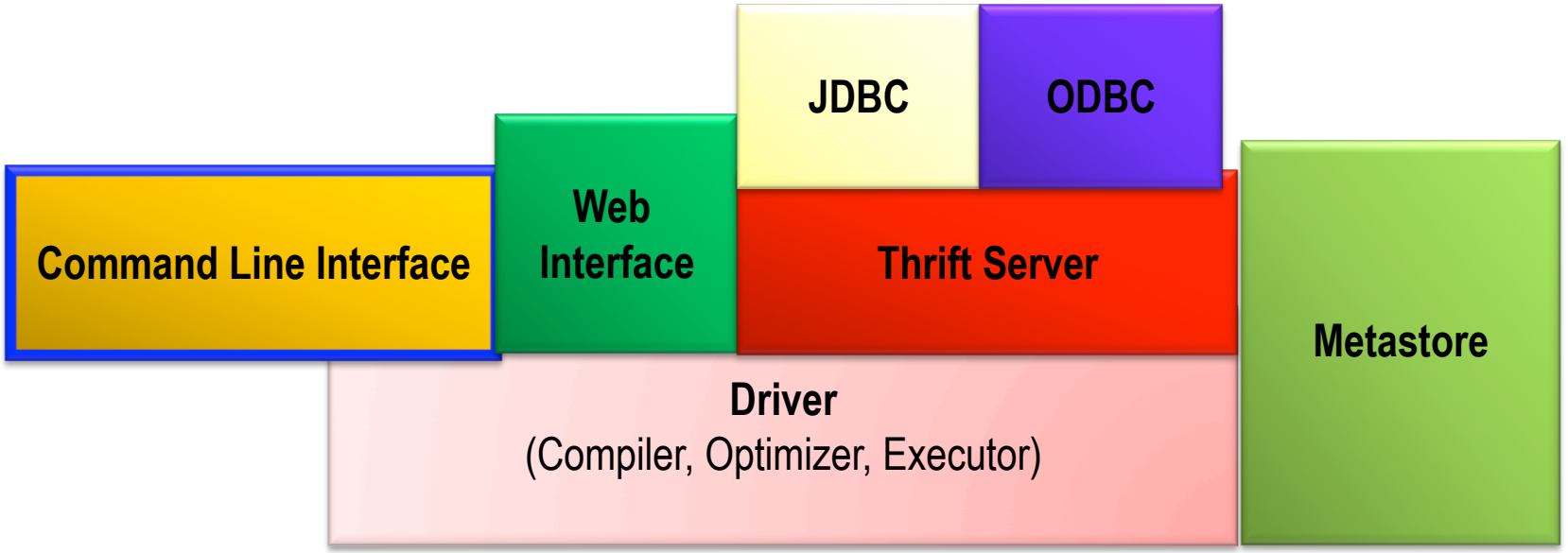
The component that executes the tasks produced by the compiler in proper dependency order. The execution engine interacts with the underlying Hadoop instance.

Hive Server



The component that provides a thrift interface and a JDBC/ODBC server and provides a way of integrating Hive with other applications.

Client Components



Client component like Command Line Interface(CLI), the web UI and JDBC/ODBC driver.

Take Home portion of Midterm#1

- Compute the total cost of building a data center that houses 10 ExaBytes of HDD
  - HDDs, networking, racks, building (real-estate), power, cooling, …
    - You could assume the following: one 4U box can contain 64 disk drives; chassis for a 4U is $20K, a rack is 48U so you can put 10 4U chasses in a rack; you can put network switches (either IB or 40GbE) on other 8U. Price the cost of rack as well.
  - You can get the cost of various parts online. State all your assumptions with reference for pricing, etc. The assumptions should be realistic. You will be evaluated based on the completeness and correctness of your solution.
  - Assume the datacenter is built in Buffalo
  - Due March 11 at 11:59PM on UBLearns.
  - 25% of your grade for midterm

Midterm#1

- Parallel/Distributed File Systems
- Statistical Methods
- Some classification
- Mapreduce programming
- R programming