Systems Infrastructure for Data Science

Web Science Group

Uni Freiburg

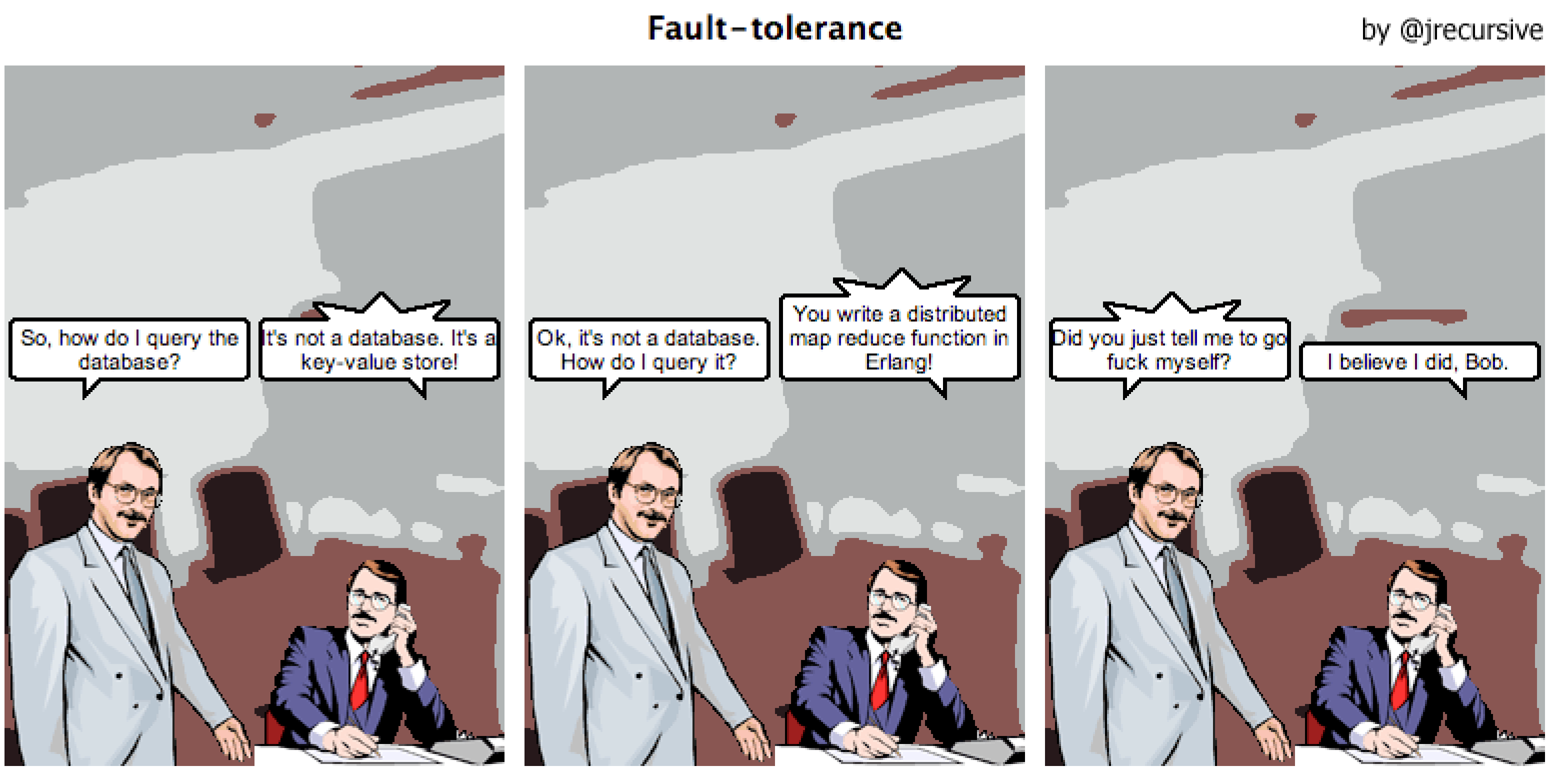
WS 2012/13

Hadoop Ecosystem

Overview of this Lecture Module

- Background
- Google MapReduce
- The Hadoop Ecosystem
  - Core components:
    - Hadoop MapReduce
    - Hadoop Distributed File System (HDFS)
  - Other selected Hadoop projects:
    - Pig
    - Hive
    - Hbase (separate lecture)

Not everybody is content with Map/Reduce



http://pig.apache.org/

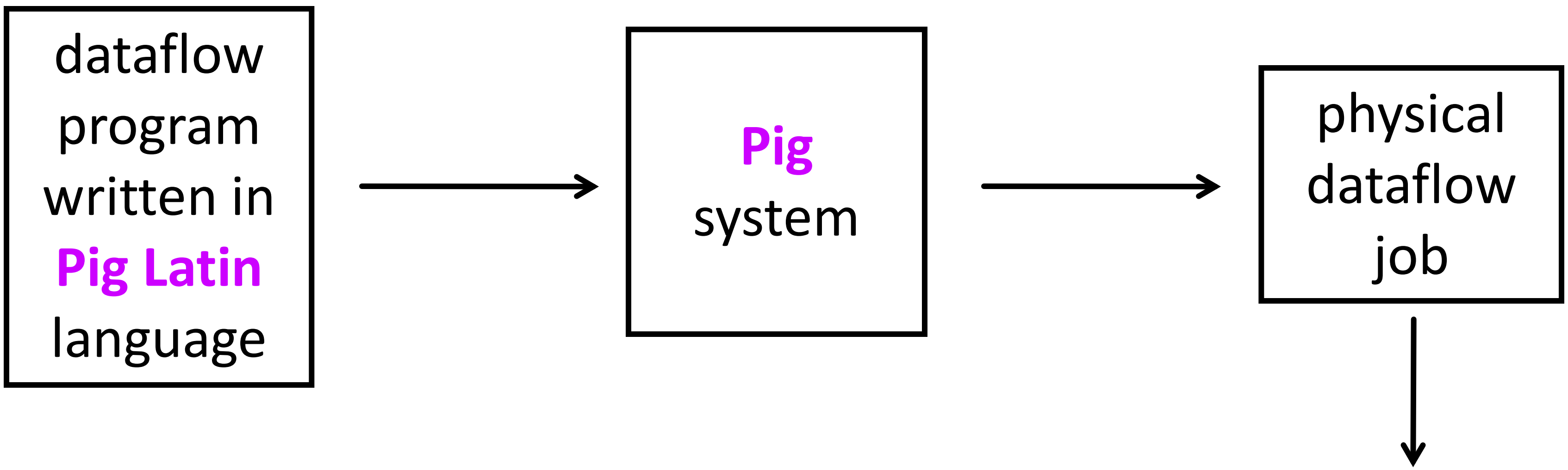
Pig & Pig Latin

- MapReduce model is too low-level and rigid
  - one-input, two-stage data flow
- Custom code even for common operations
  - hard to maintain and reuse

➢ Pig Latin: high-level data flow language

➢ Pig: a system that compiles Pig Latin into physical MapReduce plans that are executed over Hadoop

Pig & Pig Latin



A high-level language provides:
- more transparent program structure
- easier program development and maintenance
- automatic optimization opportunities

Example

Find the top 10 most visited pages in each category.

Visits

User	Url	Time
Amy	cnn.com	8:00
Amy	bbc.com	10:00
Amy	flickr.com	10:05
Fred	cnn.com	12:00

⋮

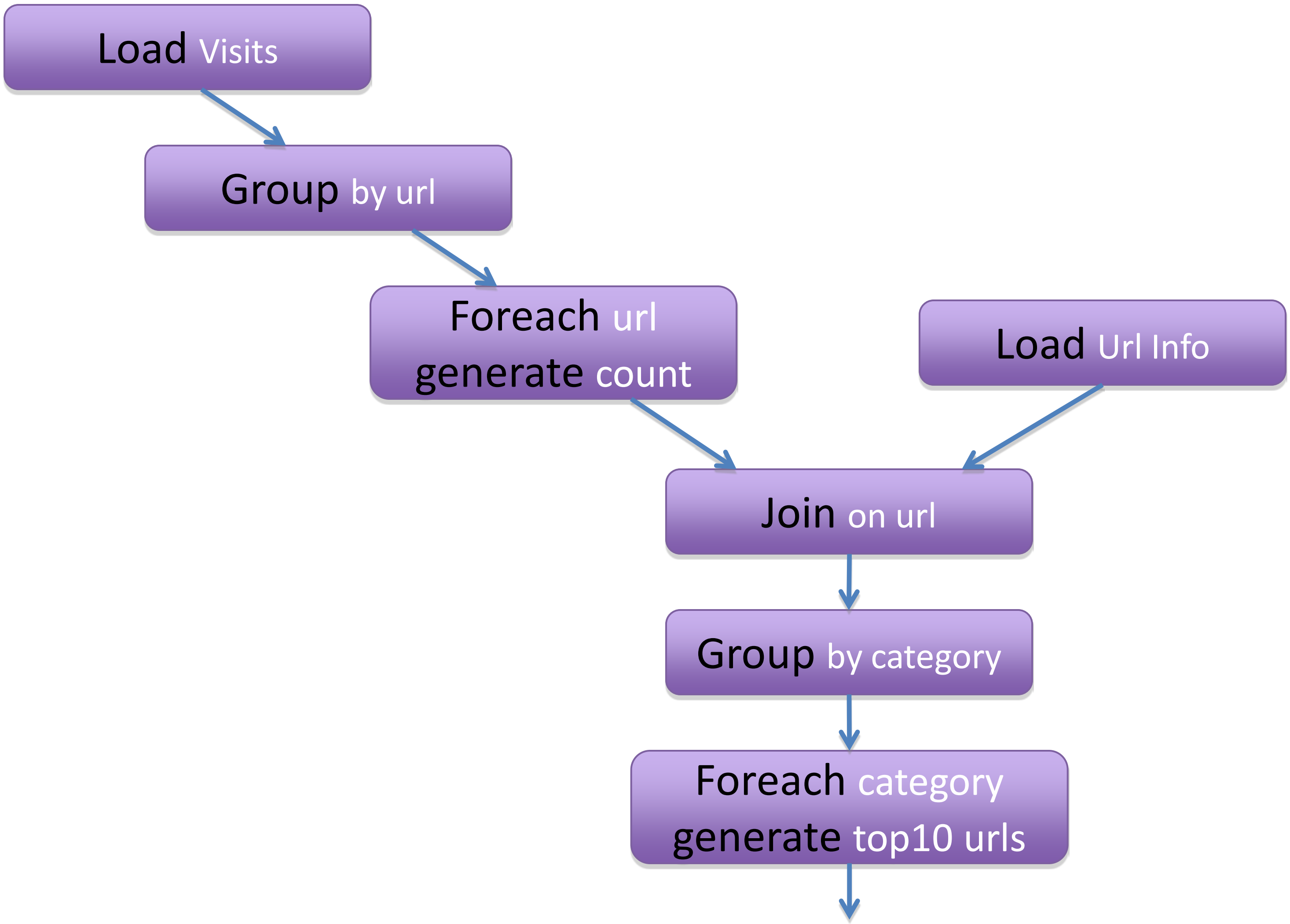
Url Info

Url	Category	PageRank
cnn.com	News	0.9
bbc.com	News	0.8
flickr.com	Photos	0.7
espn.com	Sports	0.9

⋮

Example

Data Flow Diagram



Example in Pig Latin

```
visits        = load '/data/visits' as (user, url, time);
gVisits       = group visits by url;
visitCounts   = foreach gVisits generate url, count(visits);

urlInfo       = load '/data/urlInfo' as (url, category, pRank);
visitCounts   = join visitCounts by url, urlInfo by url;

gCategories   = group visitCounts by category;
topUrls       = foreach gCategories generate top(visitCounts,10);

store topUrls into '/data/topUrls';
```

Quick Start and Interoperability

```
visits        = load '/data/visits' as (user, url, time);
gVisits       = group visits by url;
visitCounts   = foreach gVisits generate url, count(visits);

urlInfo       = load '/data/urlInfo' as (url, category, pRank);
visitCounts   = join visitCounts by url, urlInfo by url;

gCategorie
topUrls                                                   op(visitCounts,10);

store topUrls into '/data/topUrls';
```

Operates directly over files.

Quick Start and Interoperability

```
visits          = load '/data/visits' as (user, url, time);
gVisits         = group visits by url;
visitCounts     = foreach gVisits generate url, count(visits);

urlInfo         = load '/data/urlInfo' as (url, category, pRank);
visitCounts     = join visitCounts by url, urlInfo by url;

gCategorie
topUrls                                                  (visitCounts,10);

store topUrls into '/data/topUrls';
```

Schemas are optional;
can be assigned dynamically.

User-Code as a First-Class Citizen

User-Defined Functions (UDFs) can be used in every construct

- Load, Store
- Group, Filter, Foreach

```
visits          , time);
gVisits
visitCo         ount(visits);

urlInfo         tegory, pRank);
visitCounts    = join visitCounts by url, urlInfo by url;

gCategories    = group visitCounts by category;
topUrls        = foreach gCategories generate top(visitCounts,10);

store topUrls into '/data/topUrls';
```

Nested Data Model

- Pig Latin has a **fully nested data model** with four types:
  - **Atom:** simple atomic value (int, long, float, double, chararray, bytearray)
    - Example: **`'alice'`**
  - **Tuple:** sequence of fields, each of which can be of **any type**
    - Example: **`('alice', 'lakers')`**
  - **Bag:** collection of tuples, possibly with **duplicates**
    - Example: $\left\{ \begin{array}{c} (\texttt{'alice'}, \texttt{'lakers'}) \\ (\texttt{'alice'}, (\texttt{'iPod'}, \texttt{'apple'})) \end{array} \right\}$
  - **Map:** collection of data items, where each item can be looked up through a key
    - Example: $\left[\begin{array}{c} \texttt{'fan of'} \to \left\{ \begin{array}{c} (\texttt{'lakers'}) \\ (\texttt{'iPod'}) \end{array} \right\} \\ \texttt{'age'} \to 20 \end{array} \right]$

Expressions in Pig Latin

$$t = \left(\texttt{'alice'}, \left\{ \begin{array}{c} (\texttt{'lakers', 1}) \\ (\texttt{'iPod', 2}) \end{array} \right\}, \left[\texttt{'age'} \rightarrow \texttt{20} \right] \right)$$

Let fields of tuple `t` be called `f1, f2, f3`

Expression Type	Example	Value for `t`
Constant	'bob'	Independent of `t`
Field by position	$0	'alice'
Field by name	f3	['age' → 20]
Projection	f2.$0	{ ('lakers') ('iPod') }
Map Lookup	f3#'age'	20
Function Evaluation	SUM(f2.$1)	1 + 2 = 3
Conditional Expression	f3#'age'>18? 'adult':'minor'	'adult'
Flattening	FLATTEN(f2)	'lakers', 1 'iPod', 2

Commands in Pig Latin

Command	Description
LOAD	Read data from file system.
STORE	Write data to file system.
FOREACH .. GENERATE	Apply an expression to each record and output one or more records.
FILTER	Apply a predicate and remove records that do not return true.
GROUP/COGROUP	Collect records with the same key from one or more inputs.
JOIN	Join two or more inputs based on a key.
CROSS	Cross product two or more inputs.

Commands in Pig Latin (cont'd)

Command	Description
UNION	Merge two or more data sets.
SPLIT	Split data into two or more sets, based on filter conditions.
ORDER	Sort records based on a key.
DISTINCT	Remove duplicate tuples.
STREAM	Send all records through a user provided binary.
DUMP	Write output to stdout.
LIMIT	Limit the number of records.

LOAD

```
queries  =  LOAD 'query_log.txt'
            USING myLoad()
            AS (userId, queryString, timestamp);
```

file as a bag of tuples

optional deserializer

logical bag handle

optional tuple schema

STORE

a bag of tuples in Pig

output file

```
STORE query_revenues INTO 'myoutput'
      USING myStore();
```

optional serializer

- STORE command triggers the actual input reading and processing in Pig.

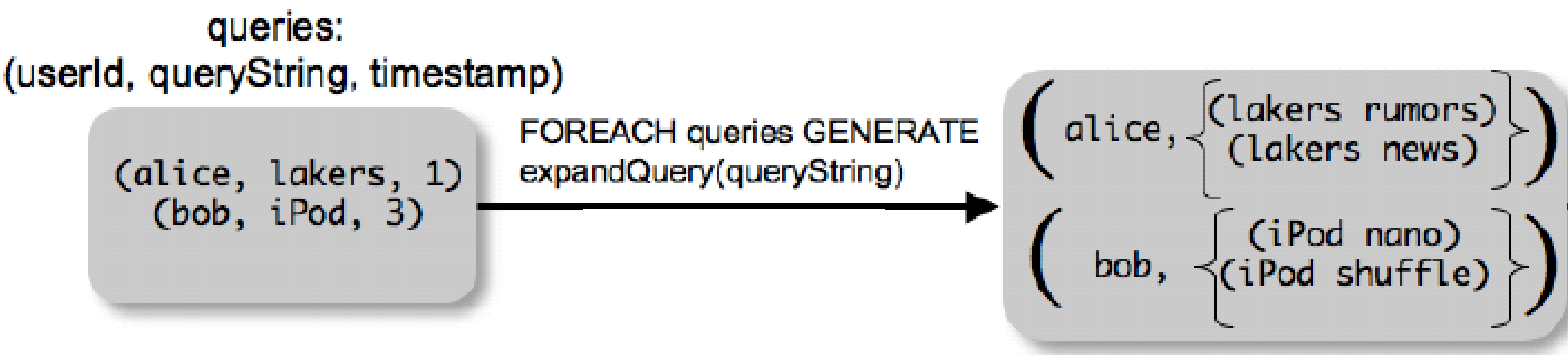
FOREACH .. GENERATE

a bag of tuples

UDF

```
expanded_queries = FOREACH queries GENERATE
                   userId, expandQuery(queryString);
```

output tuple with two fields



FILTER

a bag of tuples

```
real_queries = FILTER queries BY userId neq ‘bot’;
```

filtering condition
(comparison)

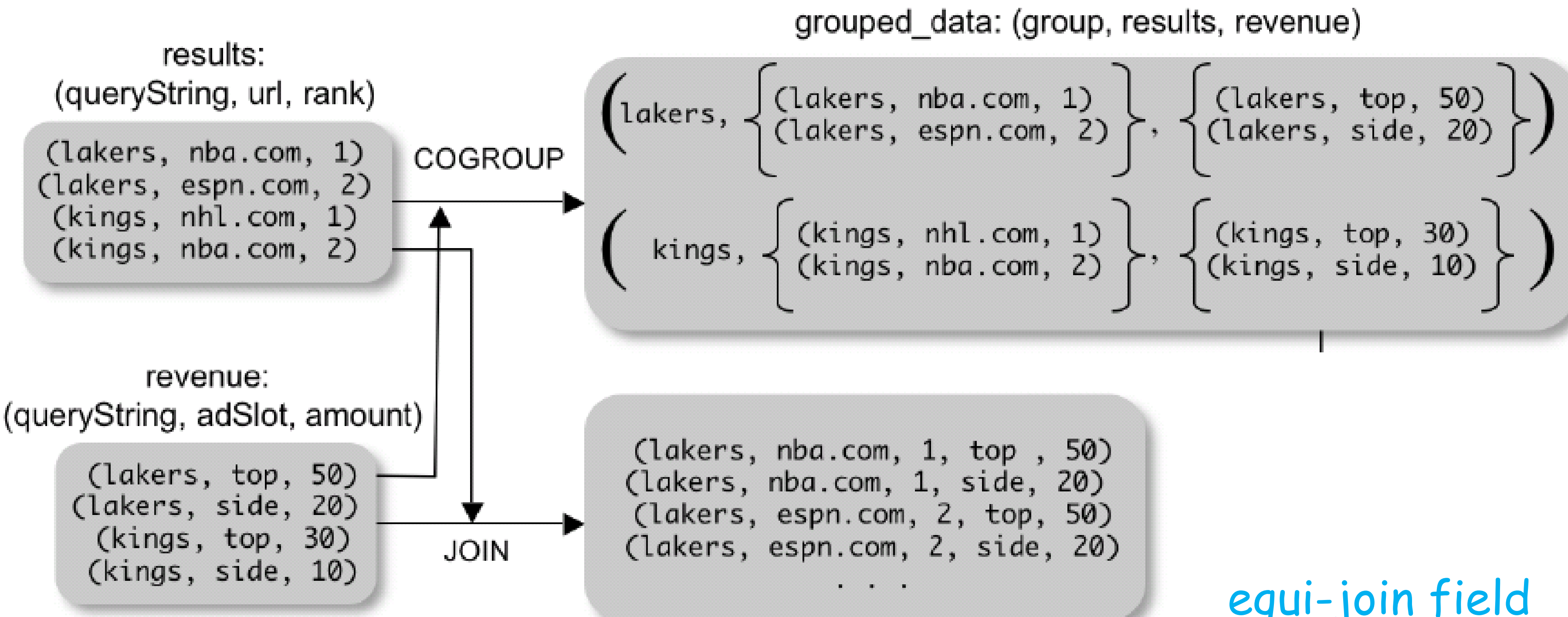
```
real_queries = FILTER queries BY NOT isBot(userId);
```

filtering condition
(UDF)

COGROUP vs. JOIN

group identifier

```
grouped_data  =  COGROUP results BY queryString,
                         revenue BY queryString;
```



equi-join field

```
join_result  =  JOIN results BY queryString,
                revenue BY queryString;
```

COGROUP vs. JOIN

- JOIN ~ COGROUP + FLATTEN

```
join_result  =  JOIN results BY queryString,
                revenue BY queryString;
```

```
   temp_var  =  COGROUP results BY queryString,
                revenue BY queryString;
join_result  =  FOREACH temp_var GENERATE
                FLATTEN(results), FLATTEN(revenue);
```

COGROUP vs. GROUP

- GROUP ~ COGROUP with only one input data set
- Example: group-by-aggregate

```
grouped_revenue = GROUP revenue BY queryString;
query_revenues = FOREACH grouped_revenue GENERATE
                 queryString,
                 SUM(revenue.amount) AS totalRevenue;
```

Nested Operations in Pig Latin

- FILTER, ORDER, and DISTINCT can be nested within a FOREACH command.

```
grouped_revenue = GROUP revenue BY queryString;
query_revenues = FOREACH grouped_revenue{
                     top_slot = FILTER revenue BY
                                adSlot eq 'top';
                     GENERATE queryString,
                              SUM(top_slot.amount),
                              SUM(revenue.amount);
                 };
```

MapReduce in Pig Latin

- A MapReduce program can be expressed in Pig Latin.

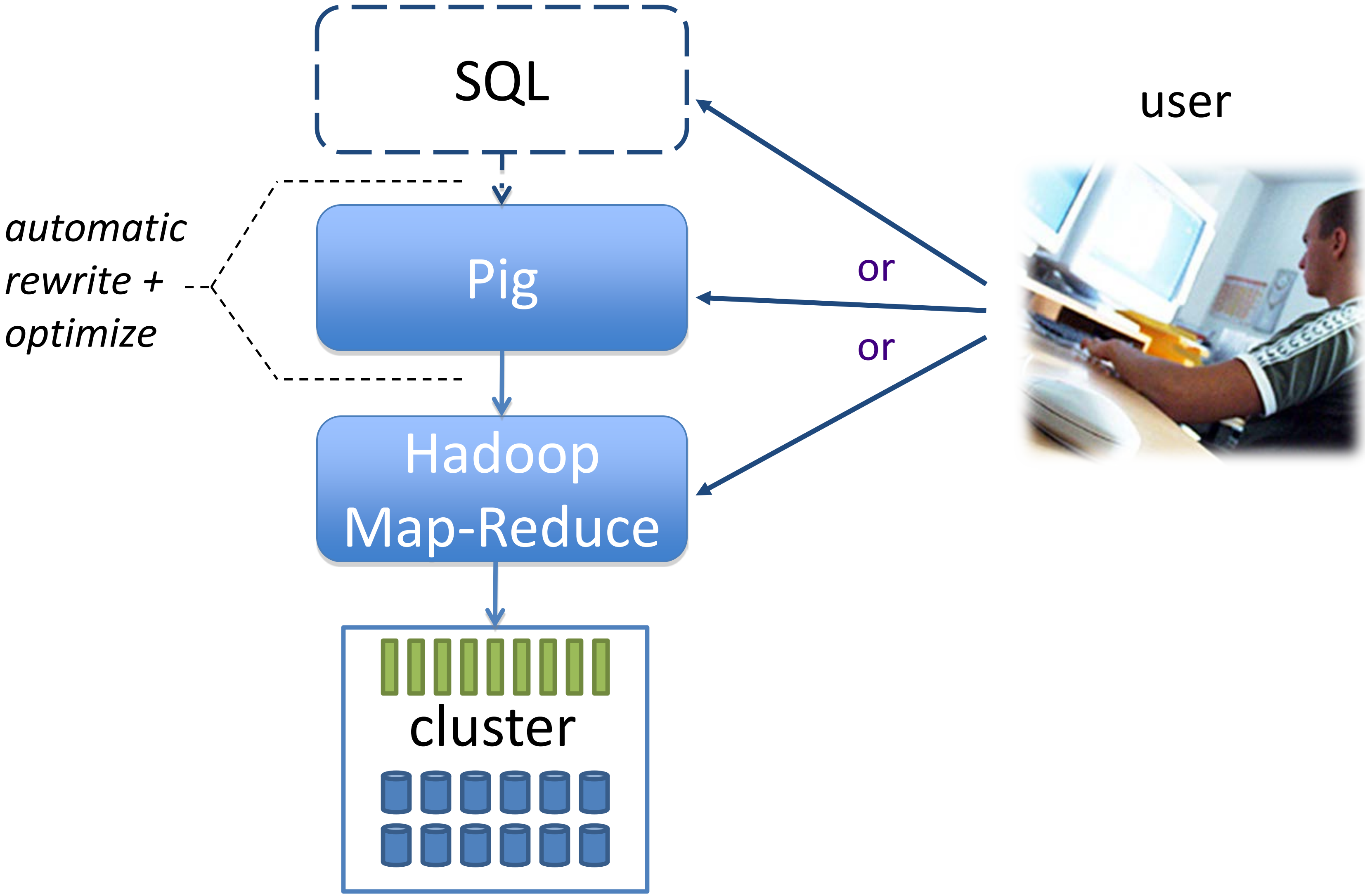
map UDF produces
a bag of key-value pairs

```
map_result = FOREACH input GENERATE FLATTEN(map(*));
key_groups = GROUP map_result BY $0;
    output = FOREACH key_groups GENERATE reduce(*);
```

key is the first field

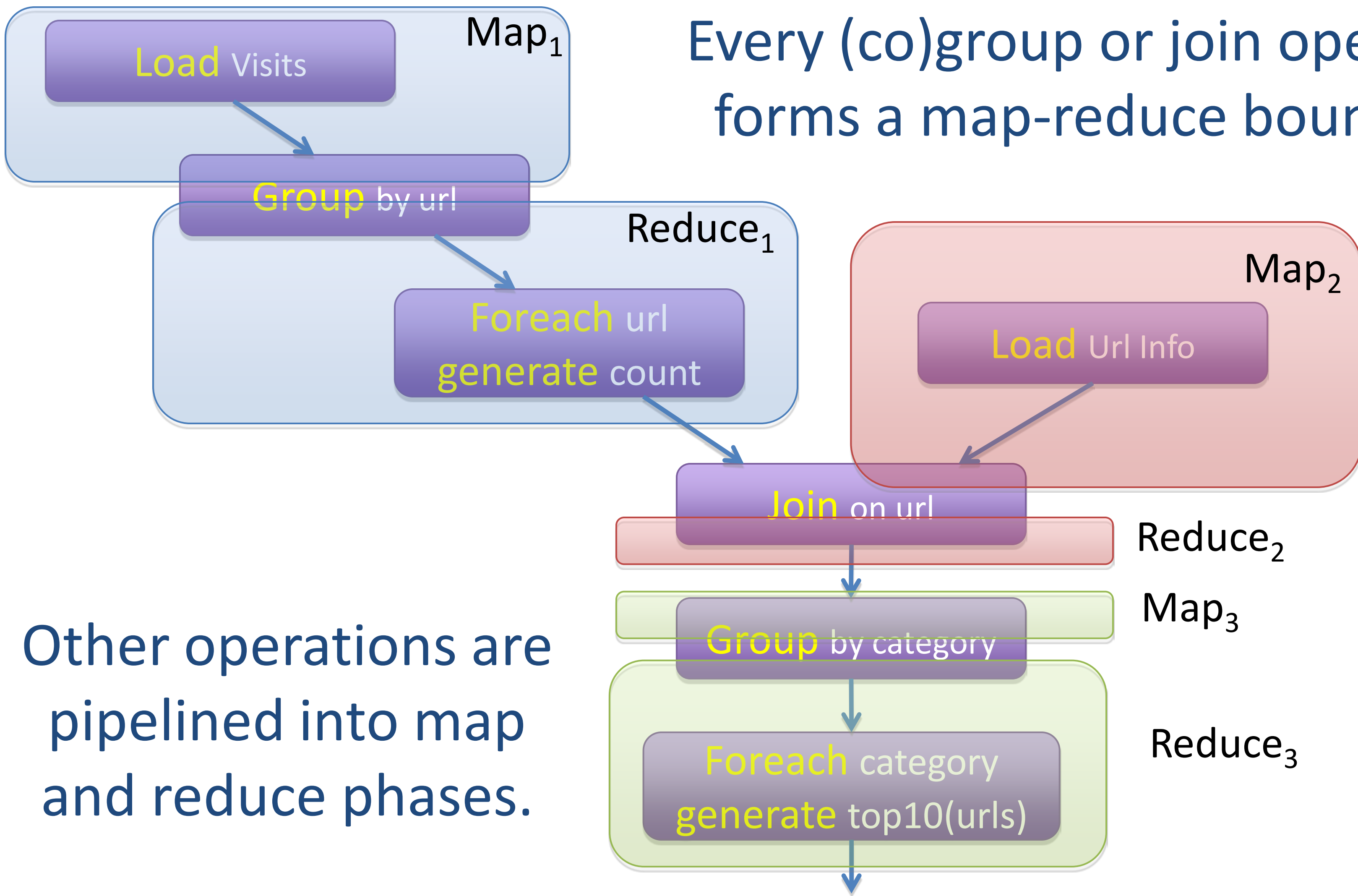
reduce UDF

Pig System Overview



Compilation into MapReduce

Every (co)group or join operation forms a map-reduce boundary.



Other operations are pipelined into map and reduce phases.

Pig vs. MapReduce

- MapReduce welds together 3 primitives:
  process records $\rightarrow$ create groups $\rightarrow$ process groups
- In Pig, these primitives are:
  - explicit
  - independent
  - fully composable
- Pig adds primitives for common operations:
  - filtering data sets
  - projecting data sets
  - combining 2 or more data sets

Pig vs. DBMS

	DBMS	Pig
workload	Bulk and random reads & writes; indexes, transactions	Bulk reads & writes only; no indexes or transactions
data representation	System controls data format Must pre-declare schema (flat data model, 1NF)	Pigs eat anything (nested data model)
programming style	System of constraints (declarative)	Sequence of steps (procedural)
customizable processing	Custom functions second-class to logic expressions	Easy to incorporate custom functions

http://hive.apache.org/

Hive – What?

- A system for managing and querying structured data
  - is built on top of Hadoop
  - uses MapReduce for execution
  - uses HDFS for storage
  - maintains structural metadata in a system catalog

- Key building principles:
  - SQL-like declarative query language (HiveQL)
  - support for nested data types
  - extensibility (types, functions, formats, scripts)
  - performance

Hive – Why?

- Big data
  - Facebook: 100s of TBs of new data every day
- Traditional data warehousing systems have limitations
  - proprietary, expensive, limited availability and scalability
- Hadoop removes these limitations, but it has a low-level programming model
  - custom programs
  - hard to maintain and reuse

➢ Hive brings traditional warehousing tools and techniques to the Hadoop eco system.

➢ Hive puts **structure** on top of the data in Hadoop + provides an **SQL-like language** to query that data.

Example: HiveQL vs. Hadoop MapReduce

```
$ hive> select key, count(1)
        from kv1
        where key > 100
        group by key;
```

instead of:

```
$ cat > /tmp/reducer.sh
uniq -c | awk '{print $2"\t"$1}‘
$ cat > /tmp/map.sh
awk -F '\001' '{if($1 > 100) print $1}‘
$ bin/hadoop jar contrib/hadoop-0.19.2-dev-streaming.jar
-input /user/hive/warehouse/kv1 -file /tmp/map.sh -file /tmp/reducer.sh
-mapper map.sh -reducer reducer.sh -output /tmp/largekey
-numReduceTasks 1
$ bin/hadoop dfs -cat /tmp/largekey/part*
```

Hive Data Model and Organization
Tables

- Data is logically organized into tables.
- Each table has a corresponding directory under a particular warehouse directory in HDFS.
- The data in a table is serialized and stored in files under that directory.
- The serialization format of each table is stored in the system catalog, called “Metastore”.
- Table schema is checked during querying, not during loading (“schema on read” vs. “schema on write”).

Hive Data Model and Organization

Partitions

- Each table can be further split into partitions, based on the values of one or more of its columns.
- Data for each partition is stored under a subdirectory of the table directory.
- Example:
  - Table T under: /user/hive/warehouse/T/
  - Partition T on columns A and B
  - Data for A=a and B=b will be stored in files under: /user/hive/warehouse/T/A=a/B=b/

Hive Data Model and Organization
Buckets

- Data in each partition can be further divided into buckets, based on the hash of a column in the table.
- Each bucket is stored as a file in the partition directory.
- Example:
  - If bucketing on column C (hash on C):

    /user/hive/warehouse/T/A=a/B=b/part-0000

    …

    /user/hive/warehouse/T/A=a/B=b/part-1000

Hive Column Types

- Primitive types
  - integers (tinyint, smallint, int, bigint)
  - floating point numbers (float, double)
  - boolean
  - string
  - timestamp
- Complex types
  - array<any-type>
  - map<primitive-type, any-type>
  - struct<field-name: any-type, ..>
- Arbitrary level of nesting

Hive Query Model

- DDL: data definition statements to create tables with specific serialization formats, partitioning/ bucketing columns
  - CREATE TABLE …
- DML: data manipulation statements to load and insert data (no updates or deletes)
  - LOAD ..
  - INSERT OVERWRITE ..
- HiveQL: SQL-like querying statements
  - SELECT .. FROM .. WHERE .. (subset of SQL)

Example

- Status updates table:

  CREATE TABLE status_updates (userid int, status string, ds string)

  ROW FORMAT DELIMITED FIELDS TERMINATED BY `\t`;

- Load the data daily from log files:

  LOAD DATA LOCAL INPATH ‘/logs/status_updates’

  INTO TABLE status_updates PARTITION (ds=’2009-03-20’)

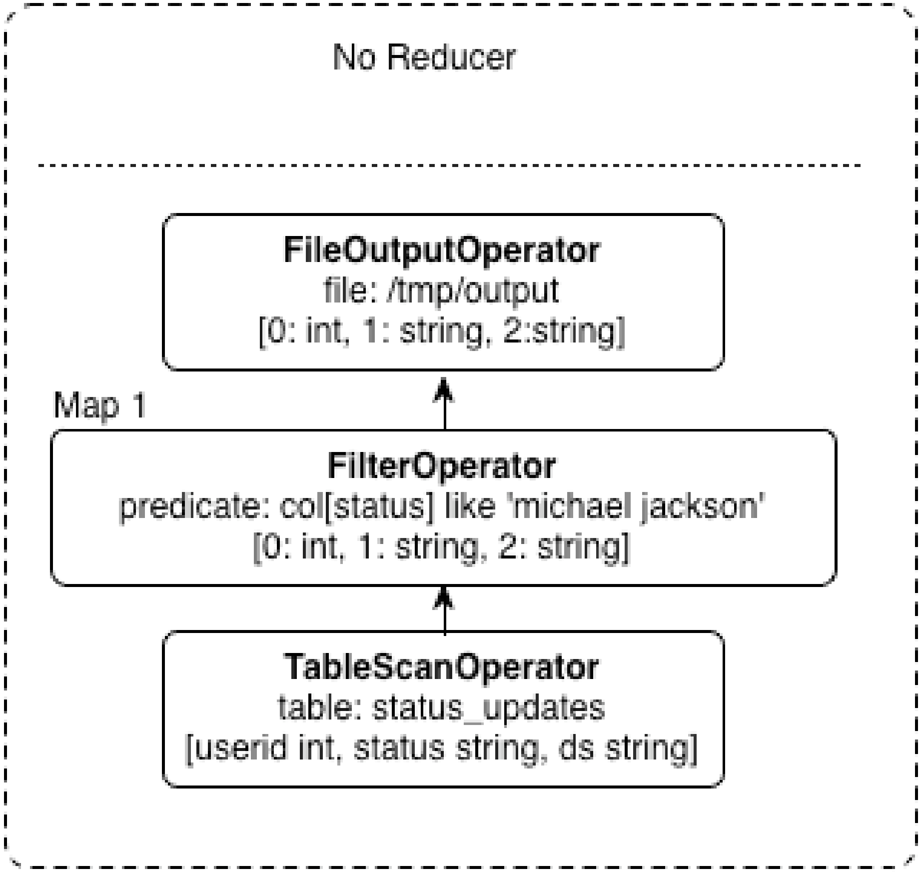
Example Query (Filter)

- Filter status updates containing ‘michael jackson’.

SELECT *

FROM status_updates

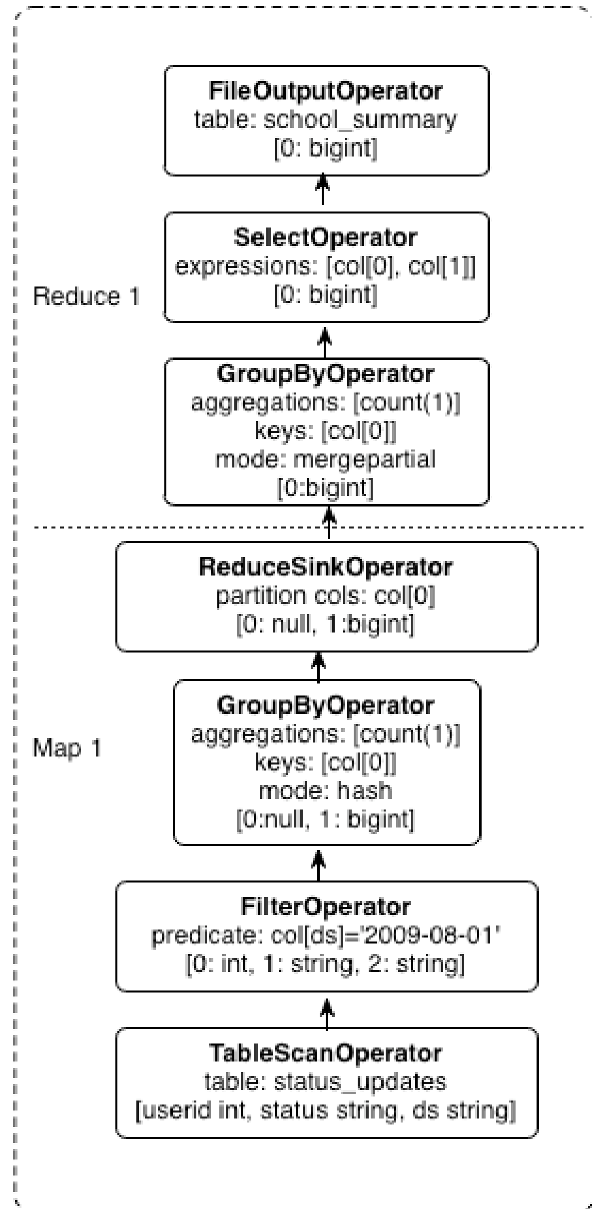
WHERE status LIKE ‘michael jackson’



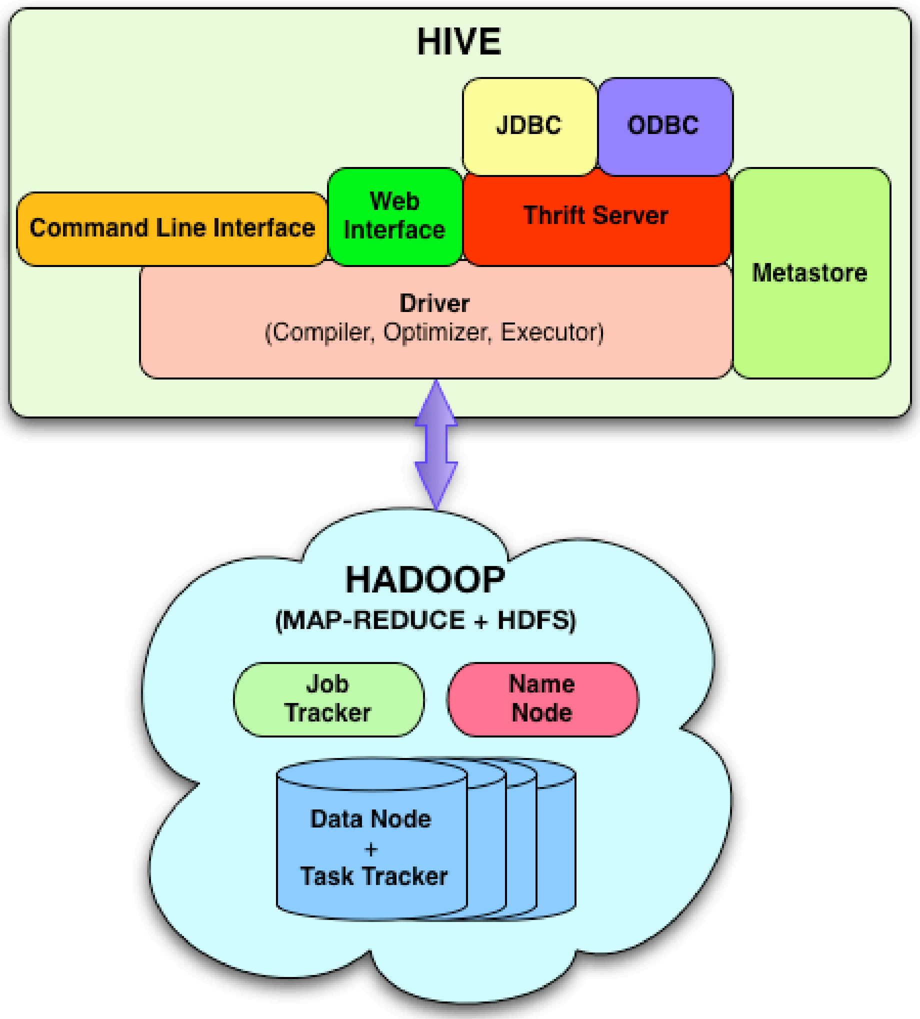
Example Query (Aggregation)

- Find the total number of status_updates in a given day.

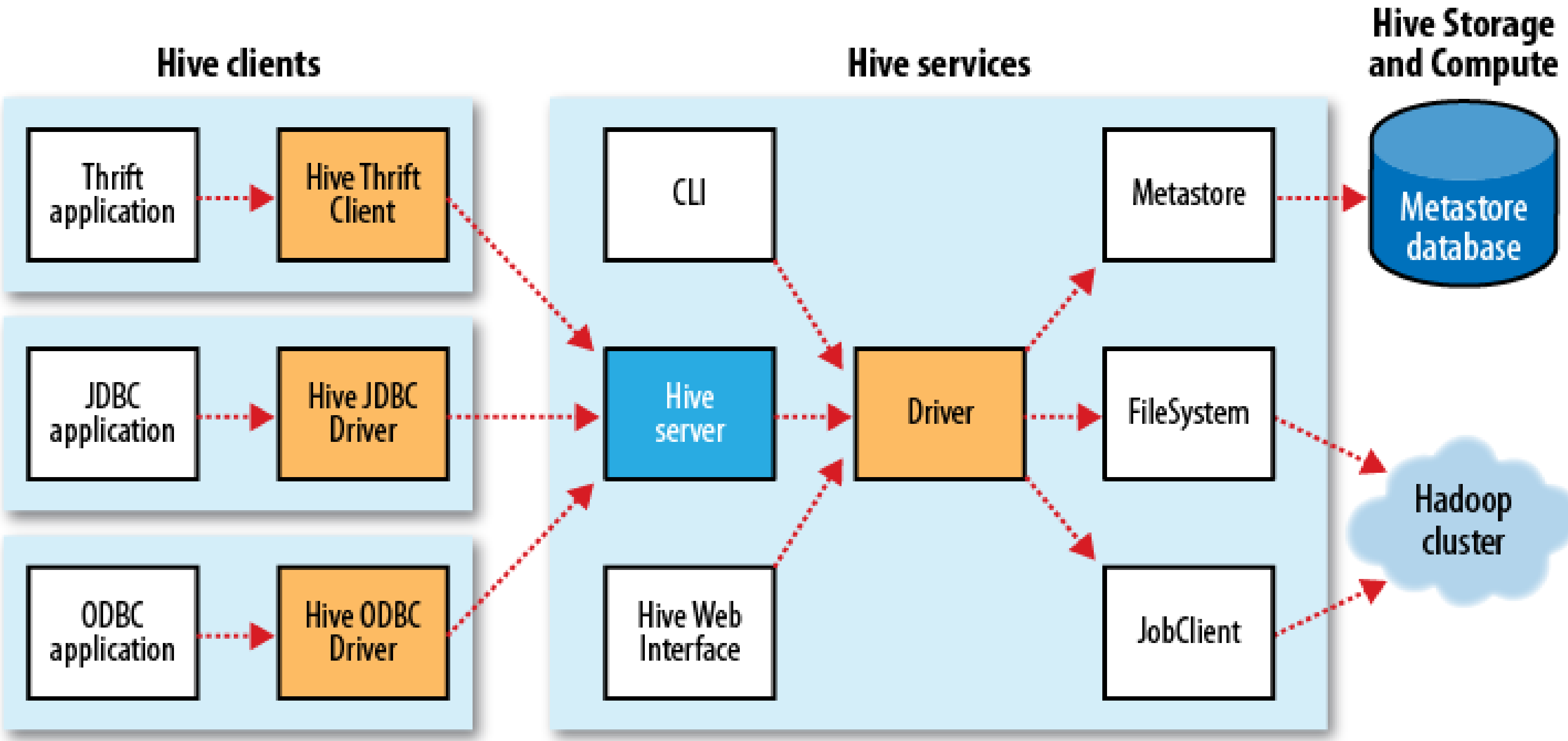
SELECT COUNT(1)
FROM status_updates
WHERE ds = ’2009-08-01’



Hive Architecture



Hive Architecture



Metastore

- System catalog that contains metadata about Hive tables
  - namespace
  - list of columns and their types; owner, storage, and serialization information
  - partition and bucketing information
  - statistics
- Not stored in HDFS
  - should be optimized for online transactions with random accesses and updates
  - use a traditional relational database (e.g., MySQL)
- Hive manages the consistency between metadata and data explicitly.

Query Compiler

- Converts query language strings into plans:
  - DDL -> metadata operations
  - DML/LOAD -> HDFS operations
  - DML/INSERT and HiveQL -> DAG of MapReduce jobs

- Consists of several steps:
  - Parsing
  - Semantic analysis
  - Logical plan generation
  - Query optimization and rewriting
  - Physical plan generation

Example Optimizations

- Column pruning
- Predicate pushdown
- Partition pruning
- Combine multiple joins with the same join key into a single multi-way join, which can be handled by a single MapReduce job
- Add repartition operators for join and group-by operators to mark the boundary between map and reduce phases

Hive Extensibility

- Define new column types.
- Define new functions written in Java:
  - UDF: user-defined functions
  - UDA: user-defined aggregation functions
- Add support for new data formats by defining custom serialize/de-serialize methods ("SerDe").
- Embed custom map/reduce scripts written in any language using a simple streaming interface.

References

- **"Pig Latin: A Not-So-Foreign Language for Data Processing"**, C. Olston et al, SIGMOD 2008.
- **"Building a High-Level Dataflow System on top of Map-Reduce: The Pig Experience"**, A. F. Gates et al, VLDB 2009.
- **"Hive: A Warehousing Solution Over a Map-Reduce Framework"**, A. Thusoo et al, VLDB 2009.
- **"Hive: A Petabyte Scale Data Warehouse Using Hadoop"**, A. Thusoo et al, ICDE 2010.
- **"BigTable: A Distributed Storage System for Structured Data"**, F. Chang et al, OSDI 2006.