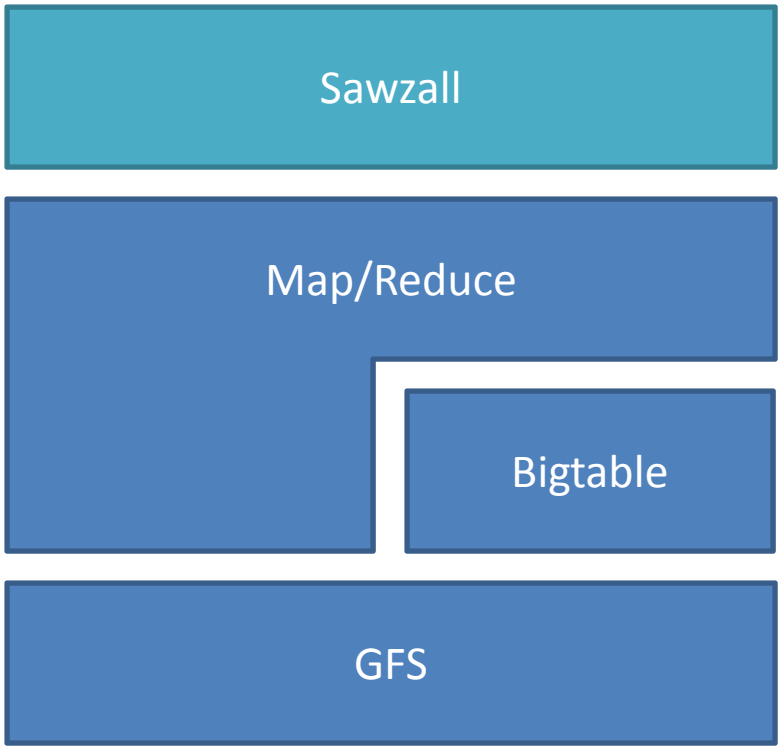
Data Management in the Cloud

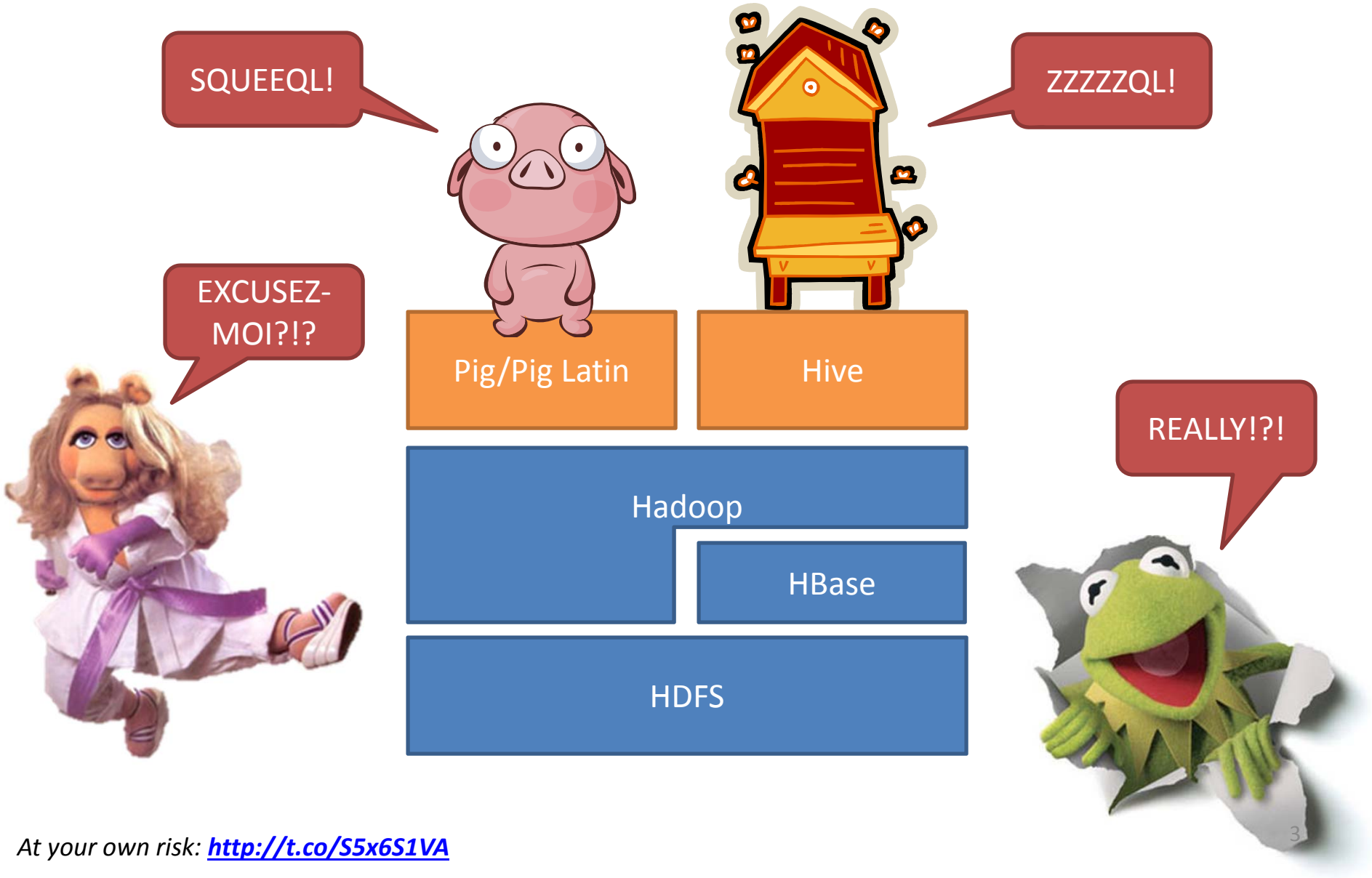
PIG LATIN AND HIVE

THANKS TO M. GROSSNIKLAUS

The Google Stack



The Hadoop Stack



At your own risk: ***http://t.co/S5x6S1VA***

Motivation for Pig Latin

- Disadvantages of parallel database products
  - prohibitively expensive at Web scale
  - programmers like to write scripts to analyze data (but see Facebook)
  - SQL is "unnatural" and overly restrictive in this context
- Limitations of Map/Reduce
  - one-input two-stage data flow is extremely rigid
  - custom code has to be written for common operations such as projection and filtering
  - opaque nature of map and reduce function impedes ability of system to perform optimization
- Pig Latin combines "best of both worlds"
  - high-level declarative querying in SQL
  - low-level procedural programming of Map/Reduce

A First Example

- Find average rank in large categories of high-rank pages
- SQL (tuple calculus)

```
SELECT category, AVG(pagerank)
FROM urls WHERE pagerank > 0.2
GROUP BY category HAVING COUNT(*) > 10^6
```

- Pig Latin (nested relational algebra)

```
good_urls = FILTER urls BY pagerank > 0.2;
groups = GROUP good_urls BY category;
big_groups =
   FILTER groups BY COUNT(good_urls) > 10^6;
output =
   FOREACH big_groups
   GENERATE category, AVG(good_urls.pagerank);
```

Pig Latin Programs

- Embody “best of both worlds” approach
- Sequence of steps
  - similar to imperative language
  - each step carries out a single data transformation
  - appealing to many developers
- High-level transformations
  - bags in, bag out
  - high-level operations render low-level manipulations unnecessary
  - potential for optimization
- Similar to specifying a query execution plan
  - “automatic query optimization has its limits, especially with uncataloged data, prevalent user-defined functions, and parallel execution”

Pig Latin Features

- "Unconventional features that are important for [..] casual ad-hoc data analysis by programmers"
  - flexible, fully nested data model
  - extensive support for user-defined functions
  - ability to operate over plain input files without any schema
  - debugging environment to deal with enormous data sets
- Pig Latin programs are executed using Pig
  - compiled into (ensembles of) map-reduce jobs
  - executed using Hadoop
  - could map to other execution environments
- Pig is an open-source project in the Apache incubator

Dataflow Language

- "While the SQL approach is good for non-programmers and/or small data sets, experienced programmers who must manipulate large data sets [..] prefer the Pig Latin approach."
  - "I much prefer writing in Pig [Latin] versus SQL. The step-by-step method of creating a program in Pig [Latin] is much cleaner and simpler to use than the single block method of SQL. It is easier to keep track of what your variables are, and where you are in the process of analyzing your data." – Jasmine Novak, Engineer, Yahoo!

Optimizations

- Pig Latin programs supply explicit sequence of operations, but are not necessarily executed in that order
- High-level relational-algebra-style operations enable traditional database optimization
- Example

```
spam_urls = FILTER urls BY isSpam(url);
culprit_urls = FILTER spam_urls BY pagerank > 0.8;
```

  - if **`isSpam`** is an expensive function and the **`FILTER`** condition is selective, it is more efficient to execute the second statement first

Optional Schemas

- Traditional database systems require importing data into system-managed tables
  - transactional consistency guarantees
  - efficient point lookups (physical tuple identifiers)
  - curate data on behalf of the user: schema enables other users to make sense of the data
- Pig only supports read-only data analysis of data sets that are often temporary
  - stored schemas are strictly optional
  - no need for time-consuming data import
  - user-provided function converts input into tuples (and vice-versa)

Nested Data Model

- Motivation
  - programmers often think in nested data models
    `term = Map<documentId, Set<positions>>`
  - in a traditional database, data must be normalized into flat table
    `term(termId, termString, …)`
    `term_position(termId, documentId, position)`
- Pig Latin has a flexible, fully nested data model
  - closer to how programmers think
  - data is often already stored in nested fashion in source files on disk
  - expressing processing tasks as sequences of steps where each step performs a single transformation requires a nested data model, e.g. **GROUP** returns a non-atomic result
  - user-defined functions are more easily written

User-Defined Functions

- Pig Latin has extensive support for user-defined functions (UDFs) for custom processing
  - analysis of search logs
  - crawl data
  - click streams
  - …
- Input and output of UDF follow flexible, nested data model
  - non-atomic input and output
  - only one type of UDF that can be used in all constructs
- UDFs are implemented in Java

Parallelism

- Pig Latin is geared towards Web-scale data
  - requires parallelism
  - does not make sense to consider non-parallel evaluation
- Pig Latin includes a small set of carefully chosen primitives that can easily be parallelized
  - "language primitives that do not lend themselves to efficient parallel evaluation have been deliberately excluded"
  - no non-equi-joins
  - no correlated sub-queries
- Backdoor
  - UDFs can be written to carry out tasks that require this functionality
  - this approach makes user aware of how efficient their programs will be and which parts will be parallelized

Data Model

- **Atom**: contains a simple atomic value
  - string, number, …
- **Tuple**: sequence of fields
  - each field can be any of the data types
- **Bag**: collection of tuple with possible duplicates
  - schema of constituent tuples is flexible
- **Map**: collection of data items, where each data item has a key
  - data items can be looked up by key
  - schema of constituent tuples is flexible
  - useful to model data sets where schemas change over time, i.e. attribute names are modeled as keys and attribute values as values

Data Model

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

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

Figure Credit: "Pig Latin: A Not-So-Foreign Language for Data Processing" by C. Olston et al., 2008

Data Loading

- First step in a Pig Latin program
  - what are the data files?
  - how are file contents deserialized, i.e. converted into Pig's data model
  - data files are assumed to contain a bag of tuples
- Example

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

  - **query_log.txt** is the input file
  - file contents are converted into tuples using the custom **myLoad** deserializer
  - the tuples have three attributes named **userId, queryString, timestamp**
  - Note: If you are explaining about queries, don't have your example be about queries (or at least call them "searches").

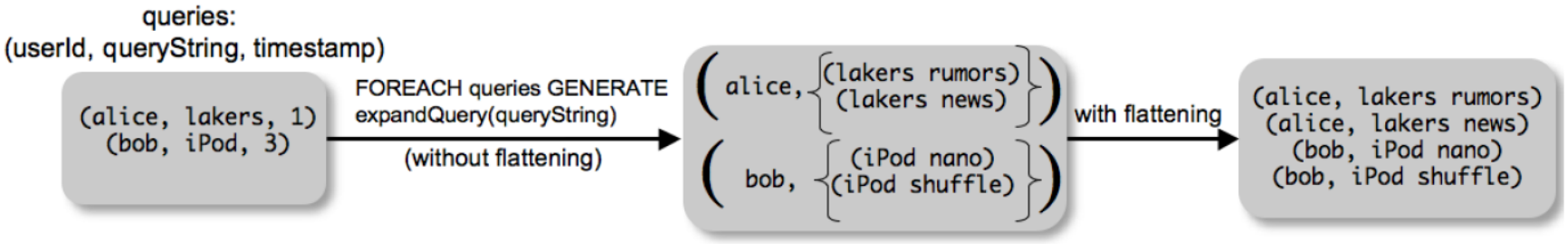
Per-Tuple Processing

- Command **FOREACH** applies some processing to every tuple of the data sets
- Example

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

  - every tuple in the **queries** bag is processed independently
  - attribute **userId** is projected
  - UDF **expandQuery** is applied to the **queryString** attribute
- Since there can be no dependence between the processing of different tuples, **FOREACH** can be easily parallelized

Per-Tuple Processing



- **GENERATE** clause is followed by a list of expressions as supported by Pig Latin's data model
- For example, **FLATTEN** can be used to unnest data

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

Figure Credit: "Pig Latin: A Not-So-Foreign Language for Data Processing" by C. Olston et al., 2008

Selection

- Tuples are selected using the FILTER command
- Example

```
real_queries = FILTER queries BY userId neq 'bot';
```

- Filtering conditions involve a combination of expressions
  - **equality**: **==** (numeric), **eq** (strings)
  - **inequality**: **!=** (numeric), **neq** (strings)
  - **logical connectors**: **AND**, **OR**, and **NOT**
  - **user-defined functions**
- Example

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

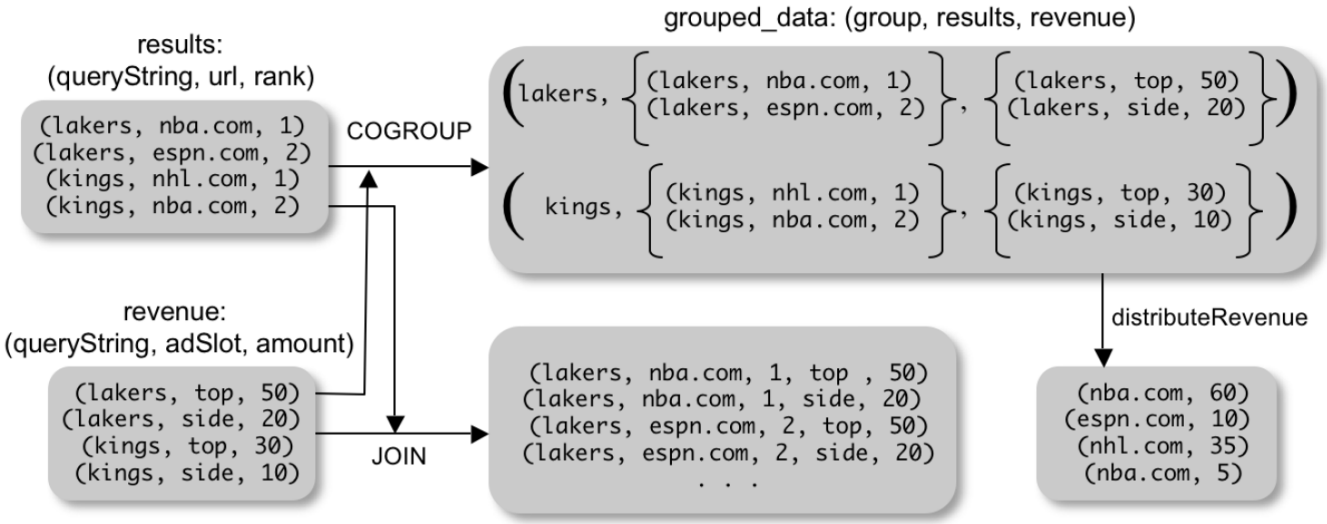
Grouping

- Command **COGROUP** groups tuples from one or more data sets
- Example

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

  - assume **(queryString, url, rank)** for **results**
  - assume **(queryString, adSlot, amount)** for **revenue**
  - **gouped_data** will be **(queryString, results, revenue)**
- Difference from **JOIN**
  - **JOIN** is equivalent to **COGROUP** followed by taking the cross-product of the tuples in the nested bags
  - **COGROUP** gives access to "intermediate result" (example on next slide)
- Nested data model enables **COGROUP** as independent operation

Grouping versus Joining



- Example

```
url_revenues = FOREACH grouped_data GENERATE
FLATTEN(distributeRevenue(results, revenue));
```

  - **distributeRevenue** attributes revenue from top slot entirely to first result, while revenue from side slot is attributed to all results
  - Since this processing task is difficult to express in SQL, **COGROUP** is a key difference between Pig Latin and SQL

Figure Credit: "Pig Latin: A Not-So-Foreign Language for Data Processing" by C. Olston et al., 2008

Syntactic Sugar

- Special case of **COGROUP** is when only one data set is involved
  - can use more intuitive keyword **GROUP**
  - similar to typical group-by/aggregate queries
- Example

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

  - **revenue.amount** refers to a projection of the nested bag in the tuples of **grouped_revenue**

More Syntactic Sugar

- Pig Latin provides a **JOIN** key word for equi-joins
- Example

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

is equivalent to

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

We Gotta Have Map/Reduce!

- Based on **FOREACH**, **GROUP**, and UDFs, map-reduce programs can be expressed
- Example

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

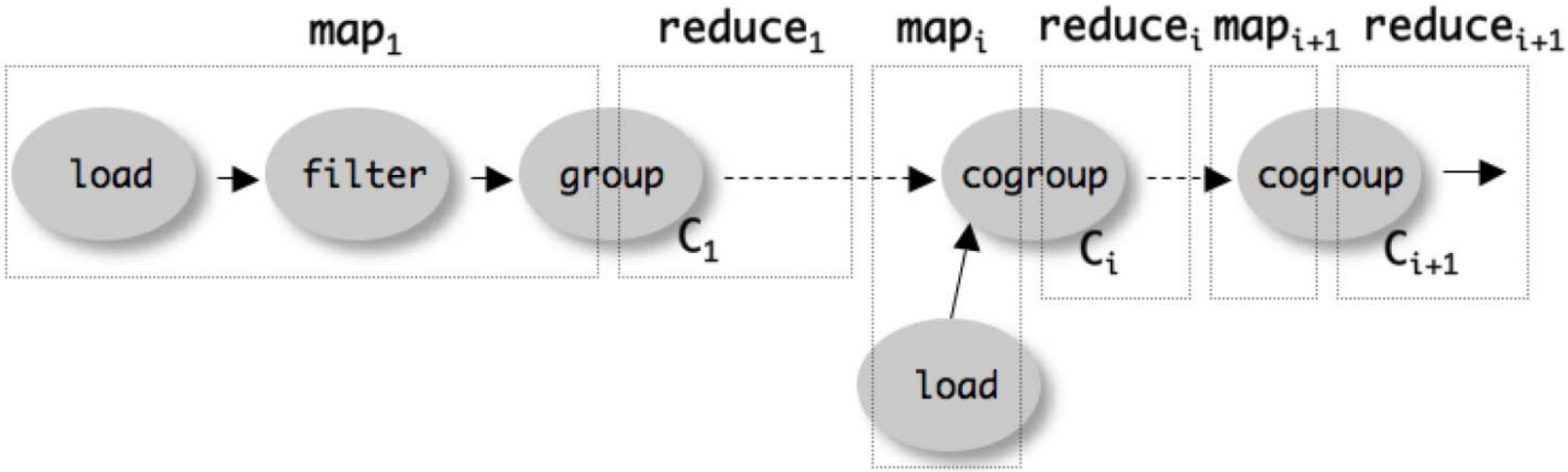
More Pig Latin Commands

- Pig Latin commands that are similar to SQL counterparts
  - UNION: returns the union of two or more bags
  - CROSS: returns the cross-product of two or more bags
  - ORDER: orders a bag by the specified fields
  - DISTINCT: eliminates duplicate tuples in the bag (syntactic sugar for grouping the bag by all fields and projecting out the groups)
- Nested operations
  - process nested bags within tuples
  - **`FILTER`**, **`ORDER`**, and **`DISTINCT`** can be nested within **`FOREACH`**
- Output
  - command **`STORE`** materializes results to a file
  - as in **`LOAD`**, default serializer can be replaced in the **`USING`** clause

Implementation

- Pig is the execution platform of Pig Latin
  - different systems can be plugged in as data processing backend
  - currently implemented using Hadoop
- Lazy execution
  - processing is only triggered when **STORE** command is invoked
  - enables in-memory pipelining and filter reordering across multiple Pig Latin commands
- Logical query plan builder
  - checks that input files and bags being referred to are valid
  - builds a plan for every bag the user defines
  - is independent of data processing backend
- Physical query plan compiler
  - compiles a Pig Latin program into map-reduce jobs (see next slide)

Mapping Pig Latin to Map/Reduce



- Each (**CO**)**GROUP** command is converted into a separate map-reduce job, i.e. a dedicated map and reduce function
- Commands between (**CO**)**GROUP** commands are appended to the preceding reduce function
- For (**CO**)**GROUP** commands with more than one data set, the map function adds an extra attribute to identify the data set

Figure Credit: "Pig Latin: A Not-So-Foreign Language for Data Processing" by C. Olston et al., 2008

More Nuts and Bolts

- Two map-reduce jobs are required for the **ORDER** command
  - first job samples input to determine statistics of sort key
  - map of second job range partitions input according to statistics
  - reduce of second job performs the sort
- Parallelism
  - **LOAD**: parallelism due to data residing in HDFS
  - **FILTER** and **FOREACH**: automatic parallelism due to Hadoop
  - (**CO**)**GROUP**: output from multiple map instances is repartitioned in parallel to multiple reduce instances

Hadoop as a Data Processing Backend

- Pros: Hadoop comes with free
  - parallelism
  - load-balancing
  - fault-tolerance
- Cons: Map-reduce model introduces overheads
  - data needs to be materialized and replicated between successive jobs
  - additional attributes need to be inserted to identify multiple data sets
- Conclusion
  - overhead is often acceptable, given the Pig Latin productivity gains
  - Pig does not preclude use of an alternative data-processing backend

Debugging Environment

- Using an iterative development and debugging cycle is not efficient in the context of long-running data processing tasks
- Pig Pen
  - interactive Pig Latin development
  - sandbox data set visualizes result of each step
- Sandbox data set
  - must meet objectives of realism, conciseness, and completeness
  - generated by random sampling, synthesizing “missing” data, and pruning redundant tuples

Debugging Environment

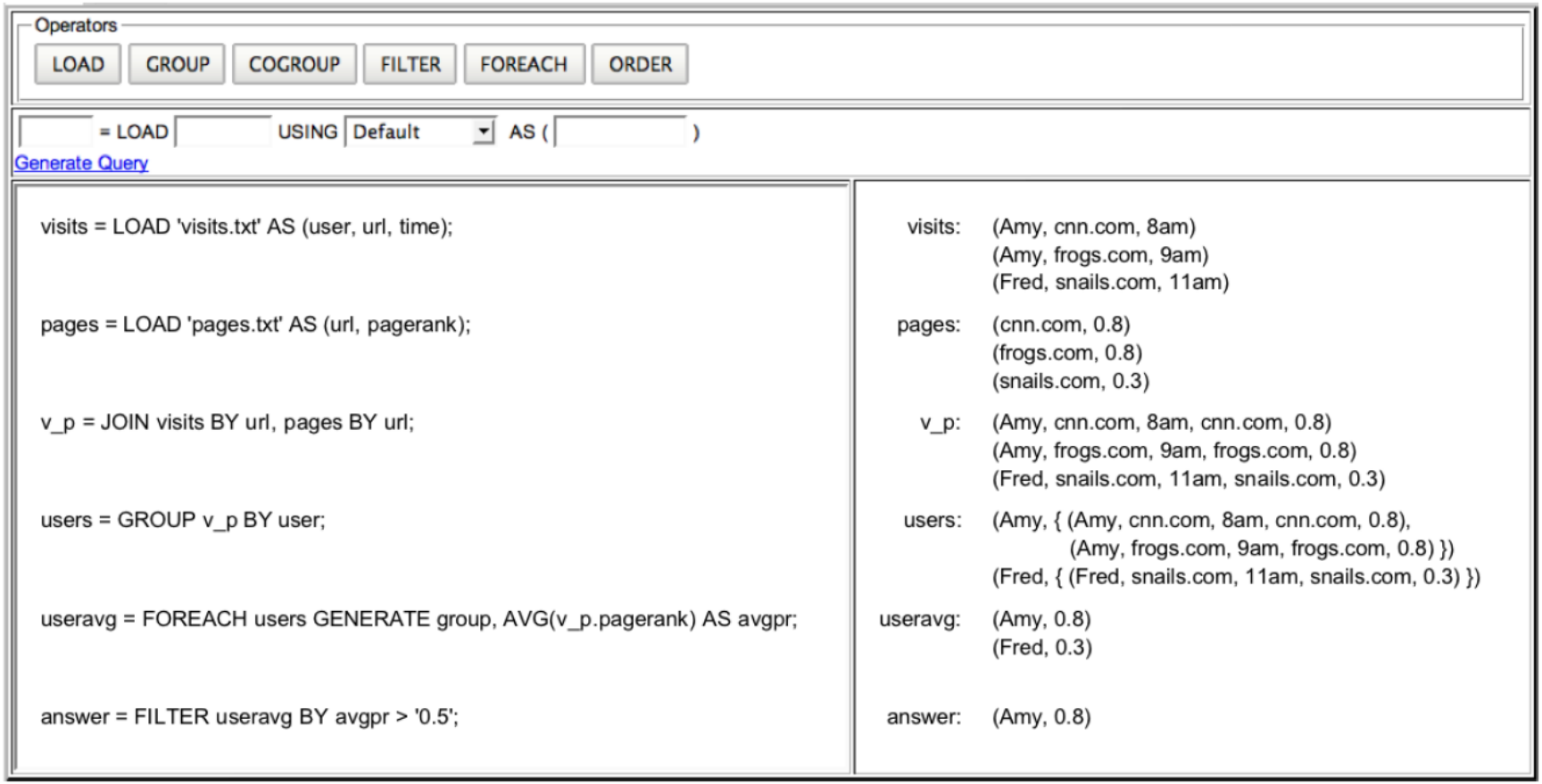


Figure Credit: "Pig Latin: A Not-So-Foreign Language for Data Processing" by C. Olston et al., 2008

Use Cases at Yahoo!

- Rollup aggregates
  - frequency of search terms aggregated over days, week, or months, and also geographical location
  - number of searches per user and average number of searches per user
  - **Pig Point**: data is too big and transient to justify curation in database
- Temporal analysis
  - how do search query distributions change over time?
  - **Pig Point**: good use case for the **COGROUP** command
- Session analysis
  - how long is the average user session?
  - how many links does a user click before leaving a page?
  - how do click patterns vary over time?
  - **Pig Point**: sessions are easily expressed in the nested data model

Use Cases Elsewhere

- AOL
  - Hadoop is used by MapQuest, Ad, Search, Truveo, and Media groups.
  - Jobs are written in Pig or native map reduce
- LinkedIn
  - Hadoop and Pig used for discovering People You May Know and other fun facts
- Salesforce.com
  - Pig used for log processing and Search, and to generate usage reports for several products and features at SFDC
  - goal is to allow Hadoop/Pig to be used across Data Warehouse, Analytics and other teams making it easier for folks outside engineering to use data
- Twitter
  - Pig used extensively to process usage logs, mine tweet data, and more
  - Twitter maintains an extension Pig function library (elephant-bird)

Motivation for Hive

- Growth of the Facebook data warehouse
  - 2007: 15TB of net data
  - 2010: 700TB of net data
  - 2011: >30PB of net data
  - 2012: >100PB of net data
- Scalable data analysis used across the company
  - ad hoc analysis
  - business intelligence
  - Insights for the Facebook Ad Network
  - analytics for page owners
  - system monitoring

Motivation for Hive (continued)

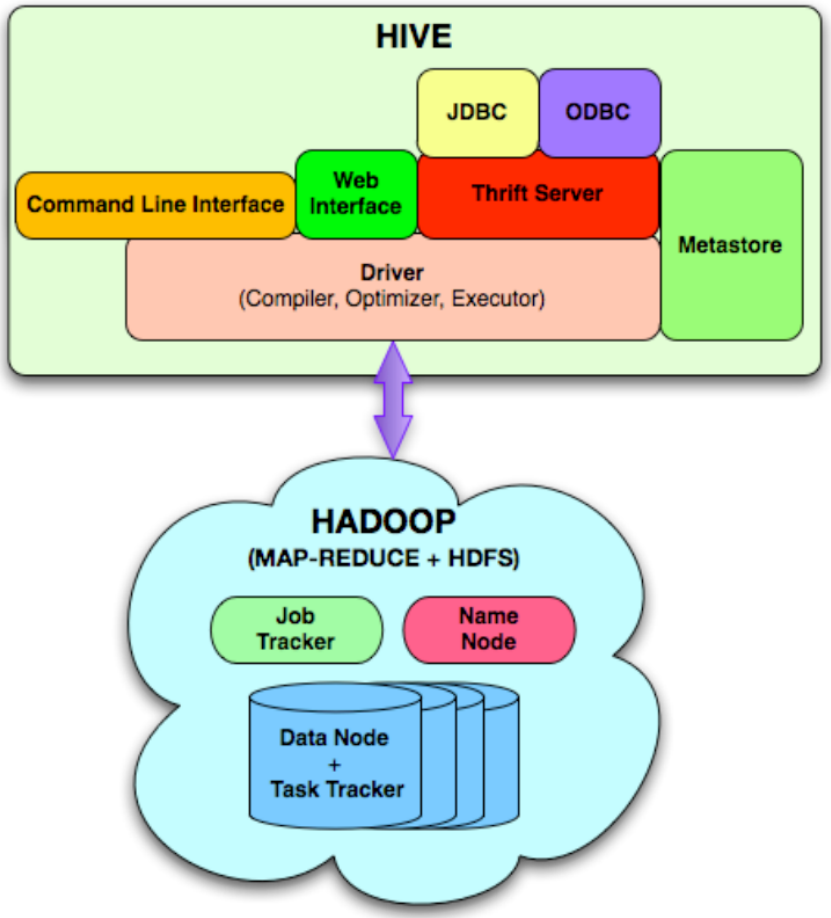
- Original Facebook data processing infrastructure
  - built using a commercial RDBMS prior to 2008
  - became inadequate as daily data processing jobs took longer than a day
- Hadoop was selected as a replacement
  - pros: petabyte scale and use of commodity hardware
  - cons: using it was not easy for end user not familiar with map-reduce
  - “Hadoop lacked the expressiveness of [..] query languages like SQL and users ended up spending hours (if not days) to write programs for even simple analysis.”

Motivation for Hive (continued)

- Hive is intended to address this problem by bridging the gap between RDBMS and Hadoop
  - "Our vision was to bring the familiar concepts of tables, columns, partitions and a subset of SQL to the unstructured world of Hadoop"

- Hive provides:
  - tools to enable easy data extract/transform/load (ETL)
  - a mechanism to impose structure on a variety of data formats
  - access to files stored either directly in HDFS or in other data storage systems such as Hbase, Cassandra, MongoDB, and Google Spreadsheets
  - a simple SQL-like query language (compare to Pig Latin)
  - query execution via MapReduce

Hive Architecture



- **Clients** use command line interface (CLI), Web UI, or JDBC/ODBC driver
- **HiveServer** provides Thrift and JDBC/ODBC interfaces
- **Metastore** stores system catalogue and metadata about tables, columns, partitions etc.
- **Driver** manages lifecycle of HiveQL statement as it moves through Hive

Figure Credit: "Hive – A Petabyte Scale Data Warehouse Using Hadoop" by A. Thusoo et al., 2010

Data Model

- Unlike Pig Latin, schemas are **not** optional in Hive
- Hive structures data into well-understood database concepts like tables, columns, rows, and partitions
- Primitive types
  - **Integers**: bigint (8 bytes), int (4 bytes), smallint (2 bytes), tinyint (1 byte)
  - **Floating point**: float (single precision), double (double precision)
  - **String**

Complex Types

- Complex types
  - **Associative arrays**: map<key-type, value-type>
  - **Lists**: list<element-type>
  - **Structs**: struct<field-name: field-type, …>
- Complex types are templated and can be composed to create types of arbitrary complexity
  - **`li list<map<string, struct<p1:int, p2:int>>`**

Complex Types

- Complex types
  - **Associative arrays**: map<key-type, value-type>
  - **Lists**: list<element-type>
  - **Structs**: struct<field-name: field-type, …>
- Accessors
  - **Associative arrays:** m[‘key’]
  - **Lists:** li[0]
  - **Structs:** s.field-name
- Example:
  - **`li list<map<string, struct<p1:int, p2:int>>`**
  - **`t1.li[0][‘key’].p1`** gives the **`p1`** field of the struct associated with the **`key`** of the first array of the list **`li`**

Query Language

- HiveQL is a subset of SQL plus some extensions
  - from clause sub-queries
  - various types of joins: inner, left outer, right outer and outer joins
  - Cartesian products
  - group by and aggregation
  - union all
  - create table as select
- Limitations
  - only equality joins
  - joins need to be written using ANSI join syntax (not in **WHERE** clause)
  - no support for inserts in existing table or data partition
  - all inserts overwrite existing data

Query Language

- Hive supports user defined functions written in java
- Three types of UDFs
  - UDF: user defined function
    - Input: single row
    - Output: single row
  - UDAF: user defined aggregate function
    - Input: multiple rows
    - Output: single row
  - UDTF: user defined table function
    - Input: single row
    - Output: multiple rows (table)

Creating Tables

- Tables are created using the **CREATE TABLE** DDL statement
- Example:

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

- Tables may be partitioned or non-partitioned (we'll see more about this later)
- Partitioned tables are created using the **PARTITIONED BY** statement

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

Inserting Data

- Example

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

  - **OVERWRITE** (instead of **INTO**) keyword to make semantics of insert statement explicit
- Lack of **INSERT INTO**, **UPDATE**, and **DELETE** enable simple mechanisms to deal with reader and writer concurrency
- At Facebook, these restrictions have not been a problem
  - data is loaded into warehouse daily or hourly
  - each batch is loaded into a new partition of the table that corresponds to that day or hour

Inserting Data

- Hive supports inserting data into HDFS, local directories, or directly into partitions (more on that later)
- Inserting into HDFS

```
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;
```

- Inserting into local directory

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

Inserting Data

- Hive supports inserting data into multiple tables/files from a single source given multiple transformations
- Example (corrected from paper):

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

    INSERT OVERWRITE DIRECTORY '/output_dir'
    SELECT t1.c2, AVG(t1.c1)
    WHERE t1.c1 > 20 AND t1.c1 <= 30
    GROUP BY t1.c2;

    INSERT OVERWRITE LOCAL DIRECTORY '/home/dir'
    SELECT t1.c2, SUM(t1.c1)
    WHERE t1.c1 > 30
    GROUP BY t1.c2;
```

Loading Data

- Hive also supports syntax that can load the data from a file in the local files system directly into a Hive table where the input data format is the same as the table format
- Example:
  - Assume we have previously issued a **`CREATE TABLE`** statement for **`page_view`**

```
LOAD DATA INPATH '/user/data/pv_2008-06-08_us.txt'
INTO TABLE page_view
```

- Alternatively we can create a table directly from the file (as we will see a little bit later)

We Gotta Have Map/Reduce!

- HiveQL has extensions to express map-reduce programs
- Example

```
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';
```

  - **MAP** clause indicates how the input columns are transformed by the mapper UDF (and supplies schema)
  - **CLUSTER BY** clause specifies output columns that are hashed and distributed to reducers
  - **REDUCE** clause specifies the UDF to be used by the reducers

We Gotta Have Map/Reduce!

- Distribution criteria between mappers and reducers can be fine tuned using **`DISTRIBUTE BY`** and **`SORT BY`**
- Example

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

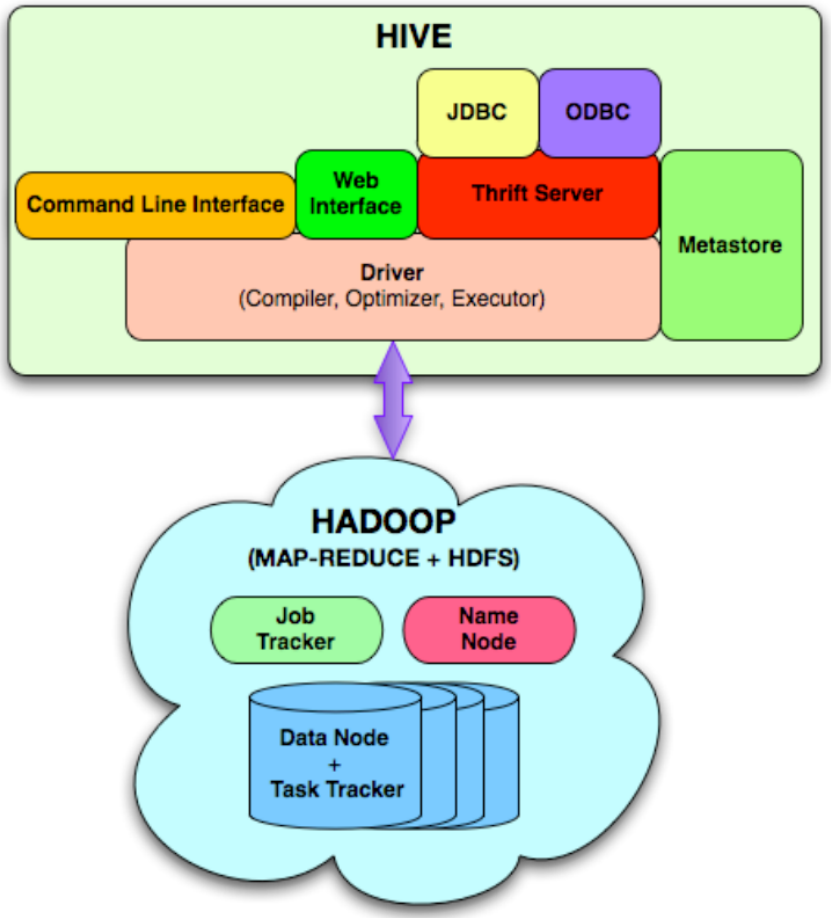
- If no transformation is necessary in the mapper or reducer the UDF can be omitted
- **`CLUSTER BY`** = **`DISTRIBUTE BY`** + **`SORT BY`** on same key

We Gotta Have Map/Reduce!

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

- Users can interchange the order of the **FROM** and **SELECT/MAP/REDUCE** clauses within a given subquery
- Mappers and reducers can be written in numerous languages

Hive Architecture



- **Clients** use command line interface, Web UI, or JDBC/ODBC driver
- **HiveServer** provides Thrift and JDBC/ODBC interfaces
- **Metastore** stores system catalogue and metadata about tables, columns, partitions etc.
- **Driver** manages lifecycle of HiveQL statement as it moves through Hive

Figure Credit: "Hive – A Petabyte Scale Data Warehouse Using Hadoop" by A. Thusoo et al., 2010

Metastore

- Stores system catalog and metadata about tables, columns, partitions, etc.
- Uses a traditional RDBMS “as this information needs to be served fast”
- Backed up regularly (since everything depends on this)
- Needs to be able to scale with the number of submitted queries (we don’t won’t thousands of Hadoop workers hitting this DB for every task)
- Only Query Compiler talks to Metastore (metadata is then sent to Hadoop workers in XML plans at runtime)

Data Storage

- Table metadata associates data in a table to HDFS directories
  - **tables**: represented by a top-level directory in HDFS
  - **table partitions**: stored as a sub-directory of the table directory
  - **buckets**: stores the actual data and resides in the sub-directory that corresponds to the bucket's partition, or in the top-level directory if there are no partitions

- Tables are stored under the Hive root directory

```
CREATE TABLE test_table (…);
```

  - Creates a directory like

    <warehouse_root_directory>/test_table

    where <warehouse_root_directory> is determined by the Hive configuration

Partitions

- Partitioned tables are created using the **PARTITIONED BY** clause in the **CREATE TABLE** statement

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

- Note that partitioning columns are not part of the table data
- New partitions can be created through an **INSERT** statement or an **ALTER** statement that adds a partition to a table

Partition Example

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

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

- Each of these statements creates a new directory
  - /…/test_part/ds=2009-01-01/hr=12
  - /…/test_part/ds=2009-02-02/hr=11
- HiveQL compiler uses this information to prune directories that need to be scanned to evaluate a query

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

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

Buckets

- A bucket is a file in the leaf level directory of a table or partition
- Users specify number of buckets and column on which to bucket data using the **`CLUSTERED BY`** clause

```
CREATE TABLE test_part(c1 string, c2 int)
PARTITIONED BY (ds string, hr int)
CLUSTERED BY (c1) INTO 32 BUCKETS;
```

Buckets

- Bucket information is then used to prune data in the case the user runs queries on a sample of data
- Example:

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

  - This query will only use 1/32 of the data as a sample from the second bucket in each partition

Serialization/Deserialization (SerDe)

- Tables are serialized and deserialized using serializers and deserializers provided by Hive or as user defined functions
- Default Hive SerDe is called the LazySerDe
  - Data stored in files
  - Rows delimited by newlines
  - Columns delimited by ctrl-A (ascii code 13)
  - Deserializes columns lazily only when a column is used in a query expression
  - Alternate delimiters can be used

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

Additional SerDes

- Facebook maintains additional SerDes including the RegexSerDe for regular expressions
- RegexSerDe can be used to 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‘
);
```

Custom SerDes

- Legacy data or data from other applications is supported through custom serializers and deserializers
  - SerDe framework
  - **`ObjectInspector`** interface
- Example

```
ADD JAR /jars/myformat.jar
CREATE TABLE t2
ROW FORMAT SERDE 'com.myformat.MySerDe';
```

File Formats

- Hadoop can store files in different formats (text, binary, column-oriented, …)
- Different formats can provide performance improvements
- Users can specify file formats in Hive using the **STORED AS** clause
  - Example:

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

- File format classes can be added as jar files in the same fashion as custom SerDes

External Tables

- Hive also supports using data that does not reside in the HDFS directories of the warehouse using the **EXTERNAL** statement
  - Example:

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

- If no custom SerDes the data in the 'mydata' file is assumed to be Hive's internal format
- Difference between external and normal tables occurs when **DROP** commands are performed
  - Normal table: metadata is dropped from Hive catalogue and data is dropped as well
  - External table: only metadata is dropped from Hive catalogue, no data is deleted

Custom Storage Handlers

- Hive supports using storage handlers besides HDFS
  - e.g. HBase, Cassandra, MongoDB, …
- A storage handler builds on existing features
  - Input formats
  - Output formats
  - SerDe libraries
- Additionally storage handlers must implement a metadata interface so that the Hive metastore and the custom storage catalogs are maintained simultaneously and consistently

Custom Storage Handlers

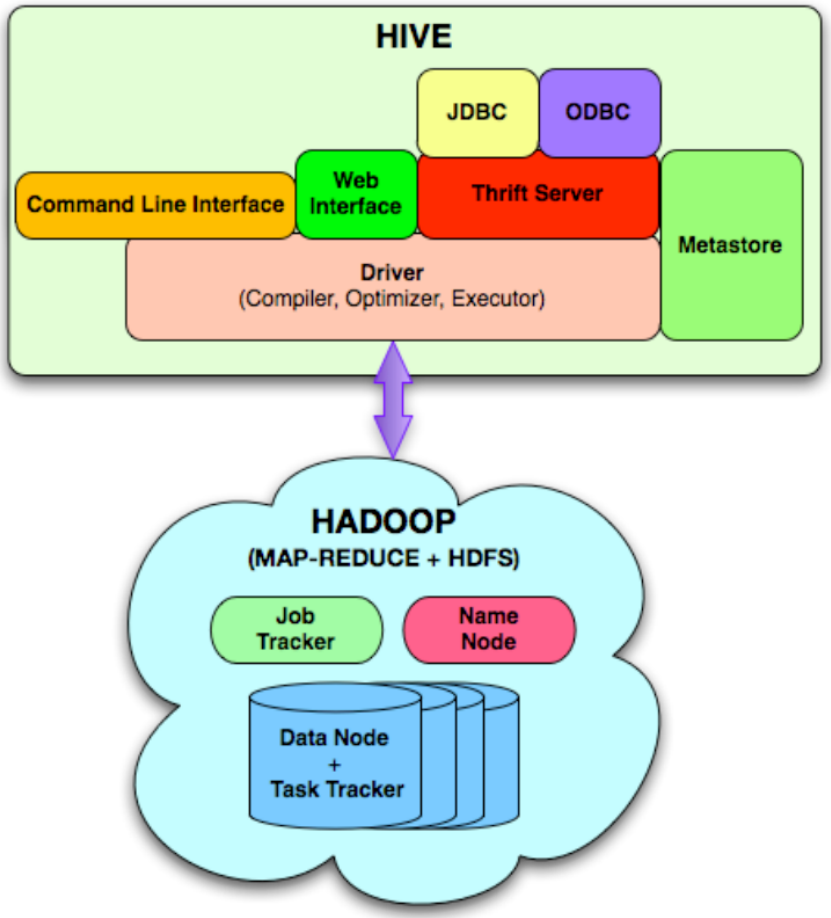
- Hive supports using custom storage and HDFS storage simultaneously
- Tables stored in custom storage are created using the **STORED BY** statement
  - Example:

```
CREATE TABLE hbase_table_1(key int, value string)
STORED BY
'org.apache.hadoop.hive.hbase.HBaseStorageHandler‘;
```

Custom Storage Handlers

- As we saw earlier Hive has *normal (managed)* and *external* tables
- Now we have *native* (stored in HDFS) and *non-native* (stored in custom storage) tables
- non-native may also use external tables
- Four possibilities for base tables
  - managed native: **`CREATE TABLE …`**
  - external native: **`CREATE EXTERNAL TABLE …`**
  - managed non-native: **`CREATE TABLE … STORED BY …`**
  - external non-native: **`CREATE EXTERNAL TABLE … STORED BY …`**

Hive Architecture



- **Clients** use command line interface, Web UI, or JDBC/ODBC driver
- **HiveServer** provides Thrift and JDBC/ODBC interfaces
- **Metastore** stores system catalogue and metadata about tables, columns, partitions etc.
- **Driver** manages lifecycle of HiveQL statement as it moves through Hive

Figure Credit: "Hive – A Petabyte Scale Data Warehouse Using Hadoop" by A. Thusoo et al., 2010

Query Compiler

- Parses HiveQL using Antlr to generate an abstract syntax tree
- Type checks and performs semantic analysis based on Metastore information
- Naïve rule-based optimizations
- Compiles HiveQL into a directed acyclic graph of MapReduce tasks

Optimizations

- Column Pruning
  - Ensures that only columns needed in query expressions are deserialized and used by the execution plan
- Predicate Pushdown
  - Filters out rows in the first scan if possible
- Partition Pruning
  - Ensures that only partitions needed by the query plan are used

Optimizations

- Map side joins
  - If one table in a join is very small it can be replicated in all of the mappers and joined with other tables
  - User must know ahead of time which are the small tables and provide hints to Hive

```
SELECT /*+ MAPJOIN(t2) */ t1.c1, t2.c1
FROM t1 JOIN t2 ON(t1.c2 = t2.c2);
```

- Join reordering
  - Smaller tables are kept in memory and larger tables are streamed in reducers ensuring that the join does not exceed memory limits

Optimizations

- GROUP BY repartitioning
  - If data is skewed in GROUP BY columns the user can specify hints like MAPJOIN

```
set hive.groupby.skewindata=true;
SELECT t1.c1, sum(t1.c2)
FROM t1
GROUP BY t1;
```

- Hashed based partial aggregations in mappers
  - Hive enables users to control the amount of memory used on mappers to hold rows in a hash table
  - As soon as that amount of memory is used, partial aggregates are sent to reducers.

MapReduce Generation

- Example:

```
FROM (SELECT a.status, b.school, b.gender
  FROM status_updates a JOIN profiles b
  ON (a.userid = b.userid
  AND a.ds='2009-03-20')) subq1

  INSERT OVERWRITE TABLE gender_summary
  PARTITION(ds='2009-03-20')
  SELECT subq1.gender, COUNT(1)
  GROUP BY subq1.gender

  INSERT OVERWRITE TABLE school_summary
  PARTITION(ds='2009-03-20')
  SELECT subq1.school, COUNT(1)
  GROUP BY subq1.school
```

MapReduce Generation

```
SELECT a.status, b.school, b.gender
  FROM status_updates a JOIN profiles b
  ON (a.userid = b.userid
  AND a.ds='2009-03-20')
```

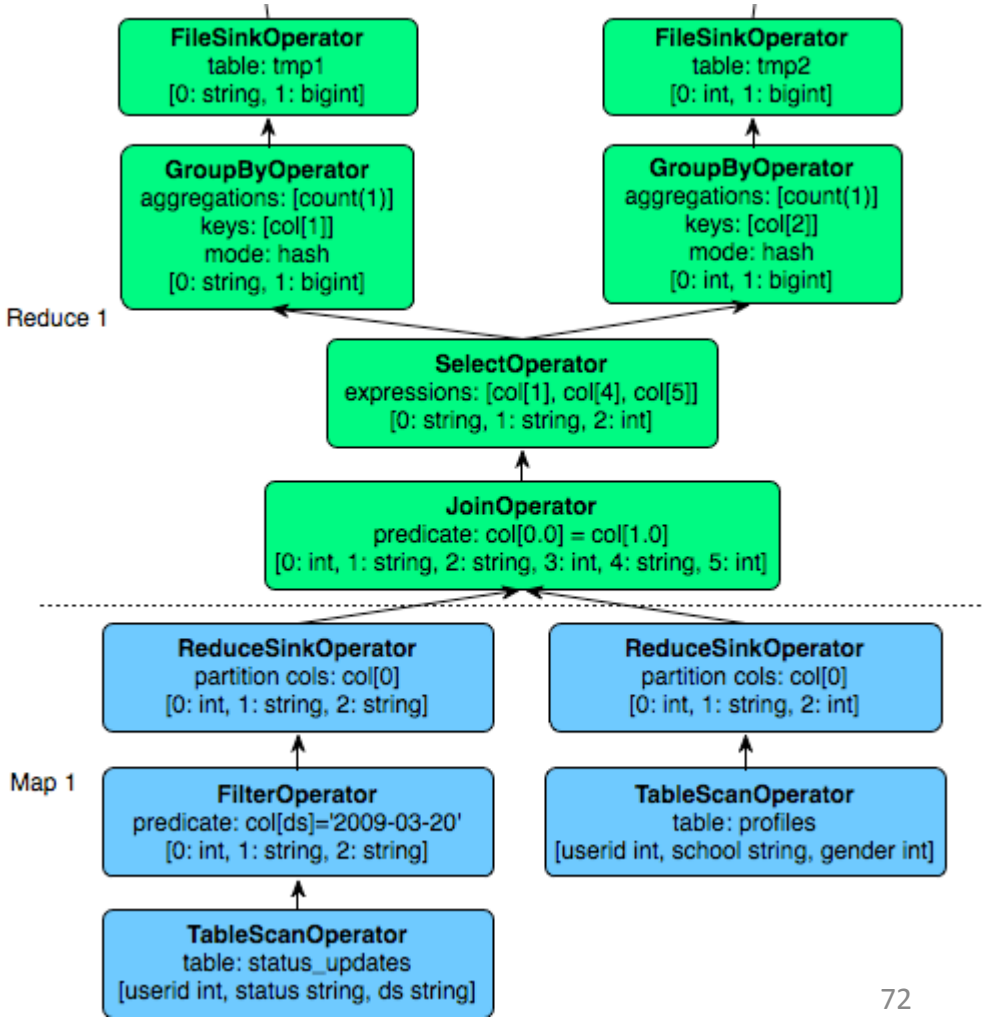


Figure Credit: "Hive – A Petabyte Scale Data Warehouse Using Hadoop" by A. Thusoo et al., 2010

MapReduce Generation

```
SELECT a.status, b.school, b.gender
  FROM status_updates a JOIN profiles b
  ON (a.userid = b.userid
  AND a.ds='2009-03-20')
```

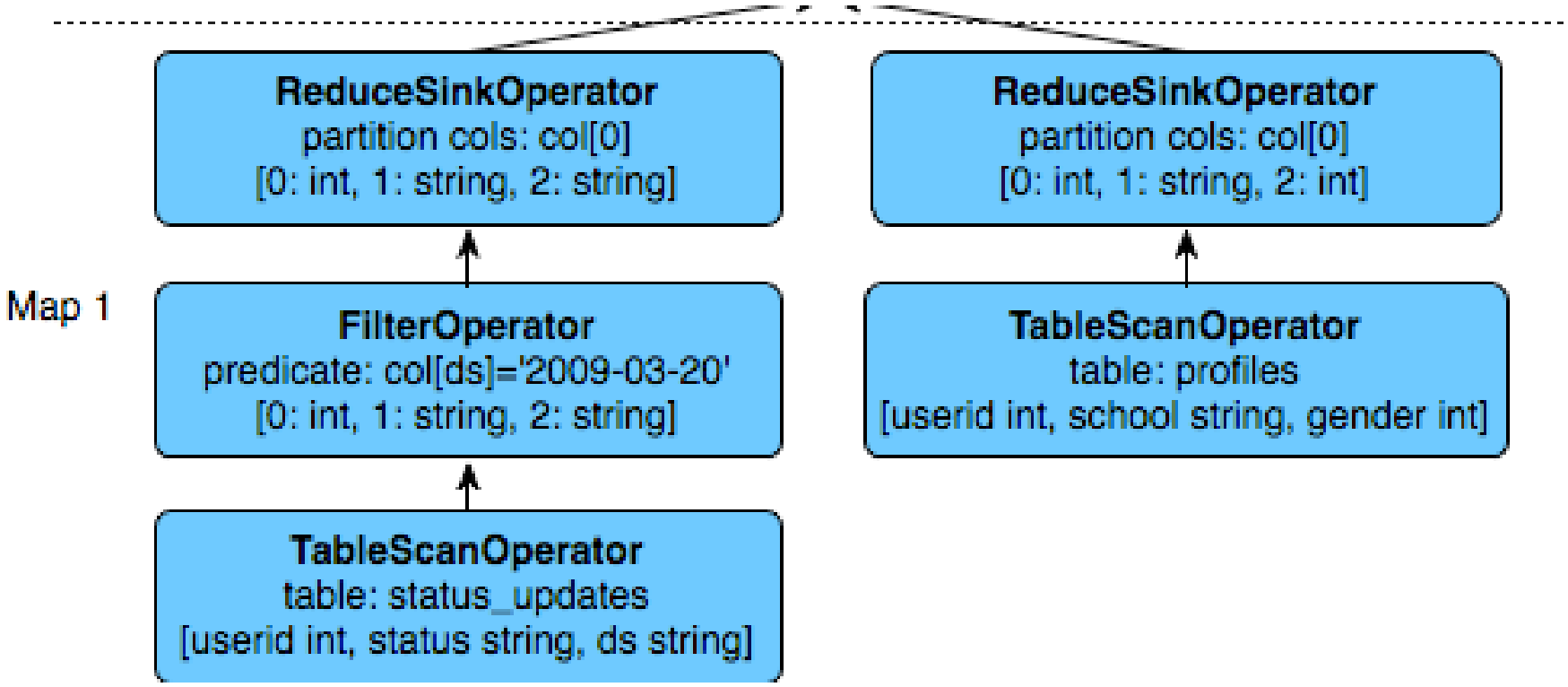


Figure Credit: "Hive – A Petabyte Scale Data Warehouse Using Hadoop" by A. Thusoo et al., 2010

MapReduce Generation

```
SELECT a.status, b.school, b.gender
  FROM status_updates a JOIN profiles b
  ON (a.userid = b.userid
  AND a.ds='2009-03-20')
```

- Note that we've already started doing some of the processing for the following **INSERT** statements

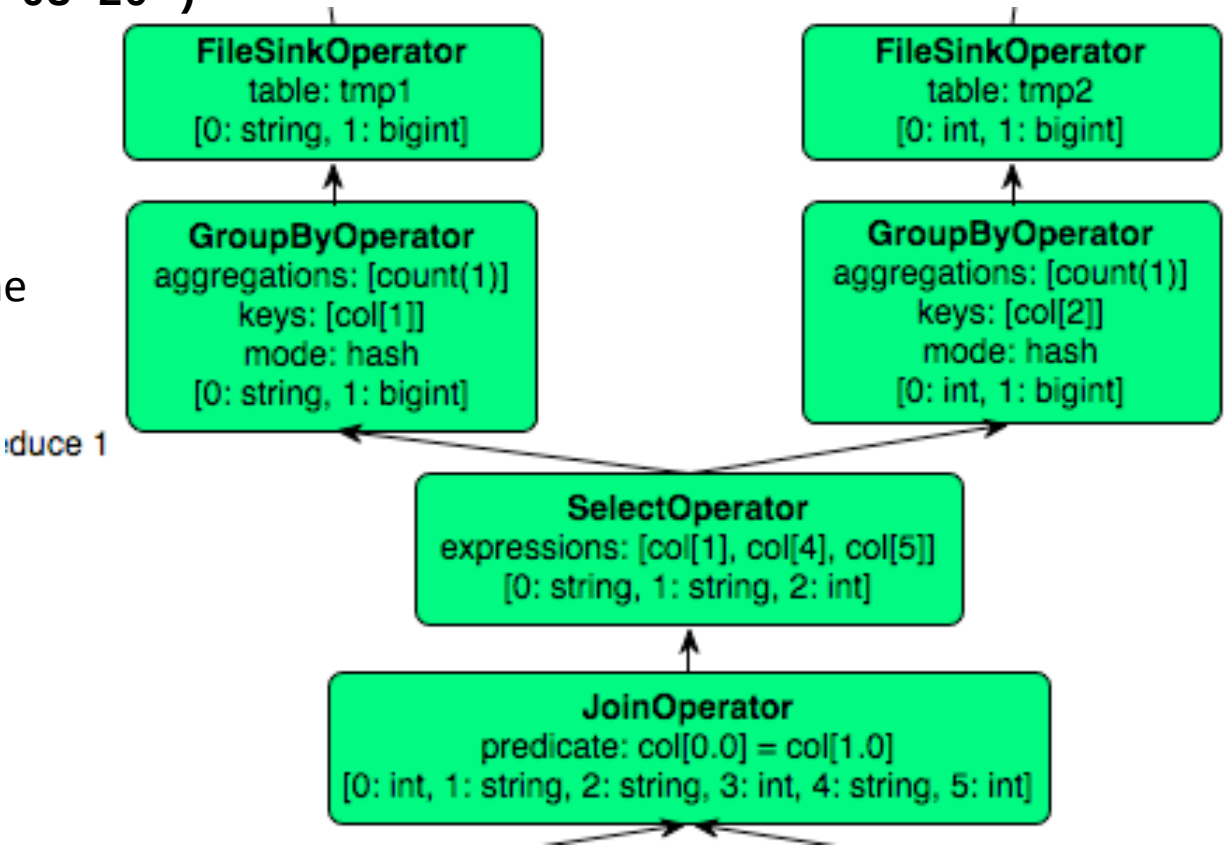


Figure Credit: "Hive – A Petabyte Scale Data Warehouse Using Hadoop" by A. Thusoo et al., 2010

MapReduce Generation

```
INSERT OVERWRITE TABLE school_summary
PARTITION(ds='2009-03-20')
SELECT subq1.school, COUNT(1)
GROUP BY subq1.school
```

```
INSERT OVERWRITE TABLE gender_summary
PARTITION(ds='2009-03-20')
SELECT subq1.gender, COUNT(1)
GROUP BY subq1.gender
```

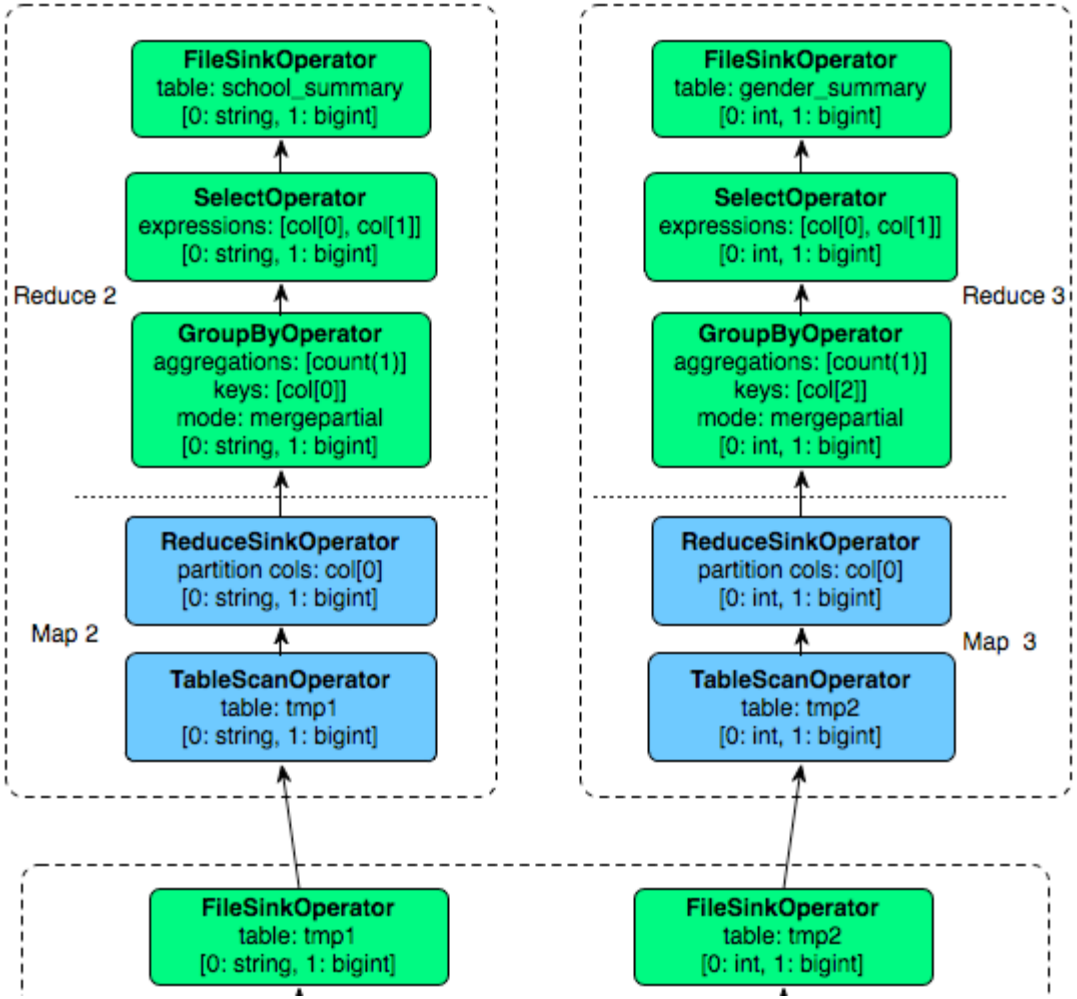


Figure Credit: "Hive – A Petabyte Scale Data Warehouse Using Hadoop" by A. Thusoo et al., 2010

MapReduce Generation

```
INSERT OVERWRITE TABLE school_summary
PARTITION(ds='2009-03-20')
SELECT subq1.school, COUNT(1)
GROUP BY subq1.school
```

```
INSERT OVERWRITE TABLE gender_summary
PARTITION(ds='2009-03-20')
SELECT subq1.gender, COUNT(1)
GROUP BY subq1.gender
```

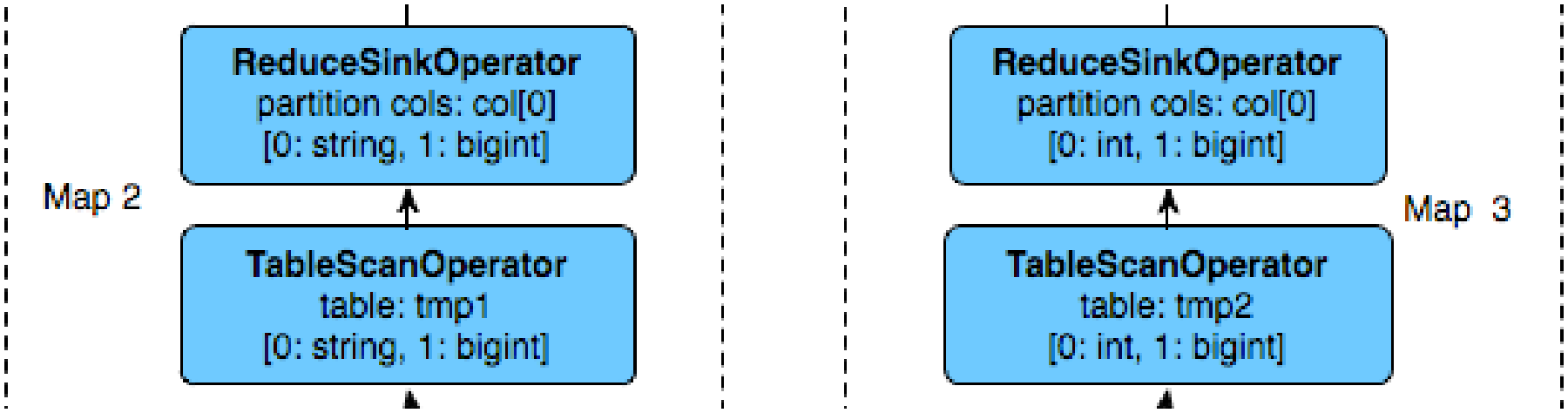


Figure Credit: "Hive – A Petabyte Scale Data Warehouse Using Hadoop" by A. Thusoo et al., 2010

MapReduce Generation

```
INSERT OVERWRITE TABLE school_summary
PARTITION(ds='2009-03-20')
SELECT subq1.school, COUNT(1)
GROUP BY subq1.school
```

```
INSERT OVERWRITE TABLE gender_summary
PARTITION(ds='2009-03-20')
SELECT subq1.gender, COUNT(1)
GROUP BY subq1.gender
```

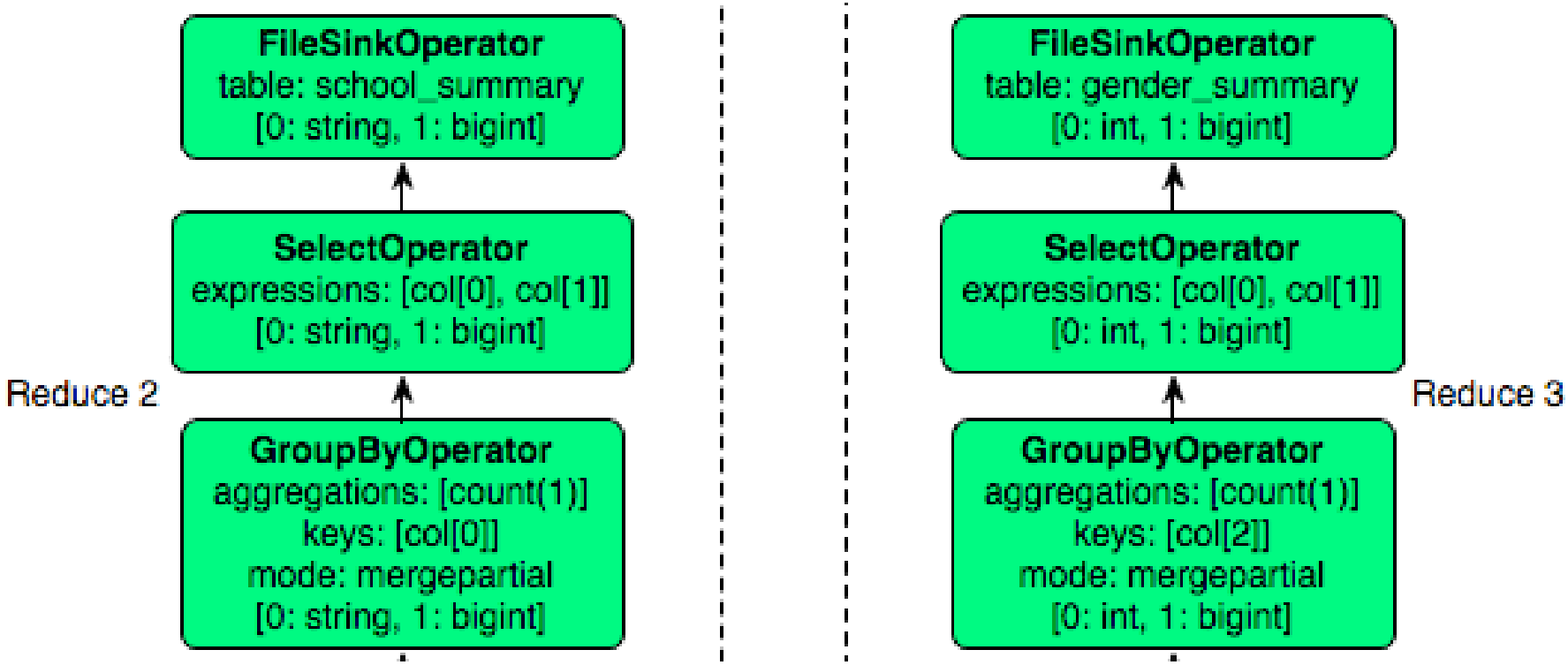


Figure Credit: "Hive – A Petabyte Scale Data Warehouse Using Hadoop" by A. Thusoo et al., 2010

Execution Engine

- MapReduce tasks are executed in the order of their dependencies
- Independent tasks can be executed in parallel

Hive Usage at Facebook

- Data processing task
  - more than 50% of the workload are ad-hoc queries
  - remaining workload produces data for reporting dashboards
  - range from simple summarization tasks to generate rollups and cubes to more advanced machine learning algorithms
- Hive is used by novice and expert users
- Types of Applications:
  - Summarization
    - Eg: Daily/Weekly aggregations of impression/click counts
  - Ad hoc Analysis
    - Eg: how many group admins broken down by state/country
  - Data Mining (Assembling training data)
    - Eg: User Engagement as a function of user attributes

Hive Usage Elsewhere

- CNET
  - Hive used for data mining, internal log analysis and ad hoc queries
- eHarmony
  - Hive used as a source for reporting/analytics and machine learning
- Grooveshark
  - Hive used for user analytics, dataset cleaning, and machine learning R&D
- Last.fm
  - Hive used for various ad hoc queries
- Scribd
  - Hive used for machine learning, data mining, ad-hoc querying, and both internal and user-facing analytics

Hive and Pig Latin

Feature	Hive	Pig
Language	SQL-like	PigLatin
Schemas/Types	Yes (explicit)	Yes (implicit)
Partitions	Yes	No
Server	Optional (Thrift)	No
User Defined Functions (UDF)	Yes (Java)	Yes (Java)
Custom Serializer/Deserializer	Yes	Yes
DFS Direct Access	Yes (implicit)	Yes (explicit)
Join/Order/Sort	Yes	Yes
Shell	Yes	Yes
Streaming	Yes	Yes
Web Interface	Yes	No
JDBC/ODBC	Yes (limited)	No

Source: Lars George (http://www.larsgeorge.com)

References

- C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins: **Pig Latin: A Not-So-Foreign Language for Data Processing.** *Proc. Intl. Conf. on Management of Data (SIGMOD), pp. 1099-1110, 2008.*
- A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Anthony, H. Liu, R. Murthy: **Hive – A Petabyte Scale Data Warehouse Using Hadoop.** *Proc. Intl. Conf. on Data Engineering (ICDE), pp. 996-1005, 2010.*