



# *Programming in Hadoop with Pig and Hive*





# *Hadoop Review*

---

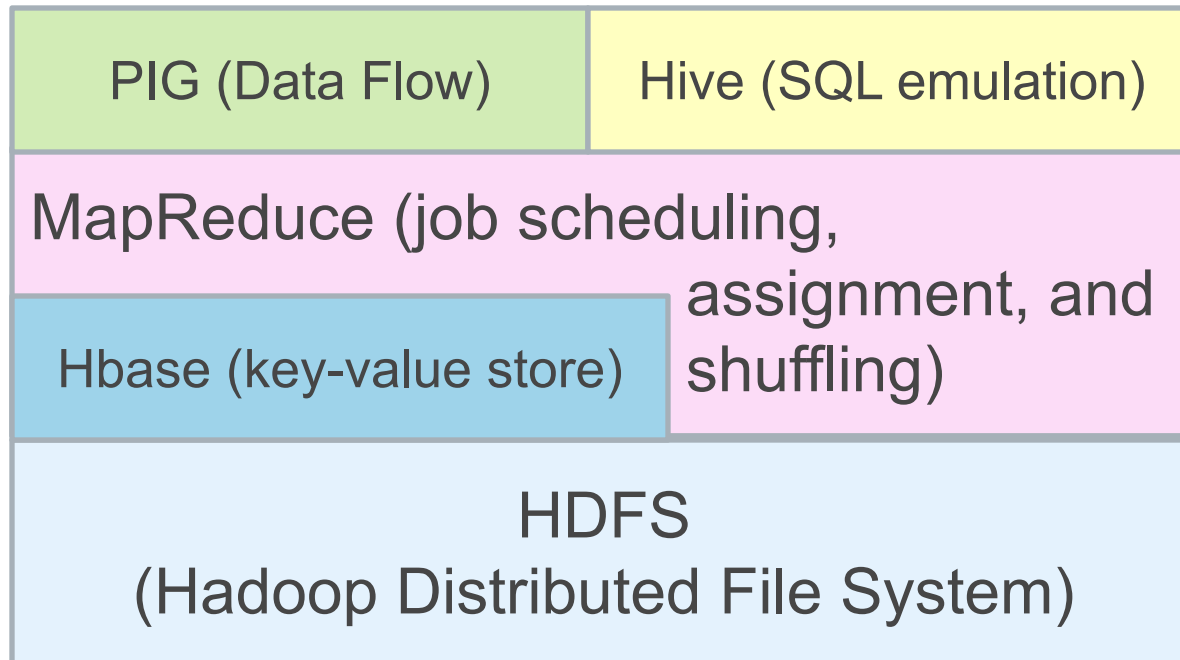
- Hadoop is a open-source reimplementation of
  - A distributed file system
  - A map-reduce processing framework
- Inspired by Google's description of the technologies underpinning their search engine
- It is a layer-cake of APIs, written mostly in Java, that one can use to write large, distributed, and scalable applications to search and process large datasets



# *Hadoop Layer Cake*

While Hadoop has many advantages, it is not intuitive to translate every data exploration/manipulation/searching task into a series of map-reduce operations.

Higher-level languages were needed.





# High-level Hadoop Interfaces

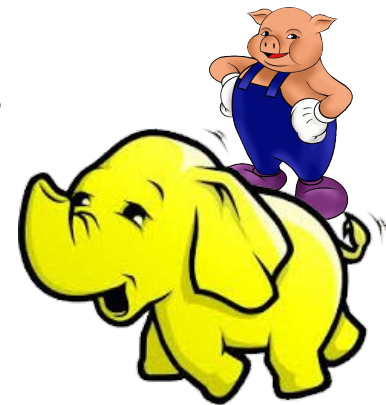
- **PIG** – A scripting language for transforming big data
  - Useful for “cleaning” and “normalizing” data
  - Three parts:
    - Pig Latin – The scripting language
    - Grunt – A interactive shell
    - Piggybank – A repository of Pig extensions
  - Deferred execution model
- **Hive** – A SQL-inspired query-oriented language
  - Imposes structure, in the form of schemas, on Hadoop data
  - Creates “data warehouse” layers





# *Pig Latin's data model*

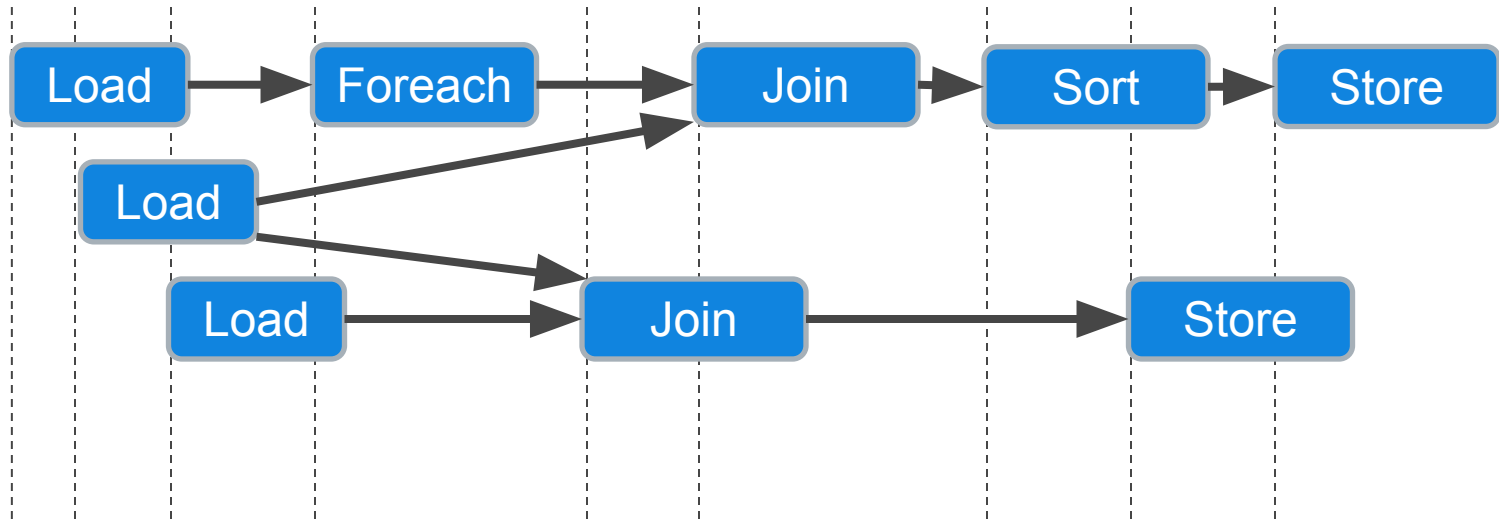
- PIG – A dataflow scripting language
  - Automatically translated to a series of Map-Reduce jobs that are run on Hadoop
  - It requires no meta-data or schema
  - It is extensible, via user-defined functions (UDFs) written in Java or other languages (C, Python, etc.)
  - Provides run-time and debugging environments
  - A language specifically designed for data manipulations and analysis
    - Supports join, sort, filter, etc.
    - Automatically partitions large operations into smaller jobs and chains them together





# *Pig Latin scripts describe dataflows*


- Every Pig Latin script describes one or more flows of data through a series of operations that can be processed in parallel (i.e. the next one can start before the ones providing inputs to it finish).
- Dataflows are Directed Acyclic Graphs (DAGS)
- Ordering and Scheduling is deferred until a node requires data





# Pig Latin Processing

- Pig Latin script are processed line by line
  - Syntax and References are checked
  - Valid statements are added to a logical plan
  - Execution is deferred until either a DUMP or STORE statement is reached
  - Reused intermediate results are mapped to a common node

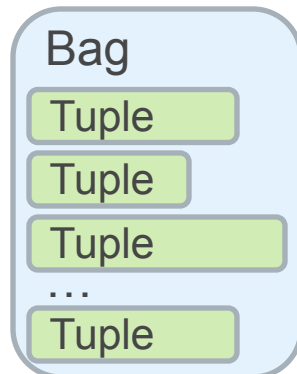
 An HDFS filename, don't worry about where it really is.

```
grunt> roster = LOAD "comp521/NFLrosters" USING PigStorage(',');
grunt> RecentQBs = FILTER roster BY $5='QB' AND $1>2000;
grunt> DUMP RecentQBs;
```



# Pig Relations

- Pig variables are bags of tuples
  - Fields – data items
  - Tuples – a vector of fields
  - Bags – a collection of unordered tuples
    - Unlike *Relations* in relational databases the tuples in a *Pig bag*, need not have the same number of fields, or the same types
- Pig also supports Maps
  - Maps – a dictionary of name-value pairs








# Pig Latin Examples

- Pig scripts are easy to read

 With AS we define an "on-the-fly" data schema

```
roster = LOAD "comp521/NFLrosters" USING PigStorage(',') AS
  (team:chararray, year:int, jersey:int, name:chararray,
   position:chararray, starts:int, games:int);
recentQBs = FILTER roster BY position='QB' AND year>2000 AND starts > 0;
Groups = GROUP recentQBs BY name;
STORE Groups INTO "recentQBTable";
```

- FOREACH to specify processing steps for all tuples in a bagexample

```
e1 = LOAD "input/Employees" USING PigStorage(',') AS
  (name:chararray, age:int, zip:int, salary:double);
f = FOREACH e1 GENERATE age, salary; -- a projection
DESCRIBE f; -- gives the schema of relation f
DUMP f;
```



# More Pig Latin Examples

- ORDER

```
emp = LOAD "input/Employees" USING PigStorage(',') AS
  (name:chararray, age:int, zip:int, salary:double);
sorted = ORDER emp BY salary;
```

- LIMIT, SAMPLE

```
emp = LOAD "input/Employees" USING PigStorage(',') AS
  (name:chararray, age:int, zip:int, salary:double);
agegroup = GROUP emp BY age;
shortlist = LIMIT agegroup 100;
```

- JOIN

```
emp = LOAD "input/Employees" USING PigStorage(',') AS
  (name:chararray, age:int, zip:int, salary:double);
pbk = LOAD "input/Phonebook" USING PigStorage(',') AS
  (name:chararray, phone:chararray);
contact = JOIN emp BY name, pbk BY name;
DESCRIBE contact;
DUMP contact;
```



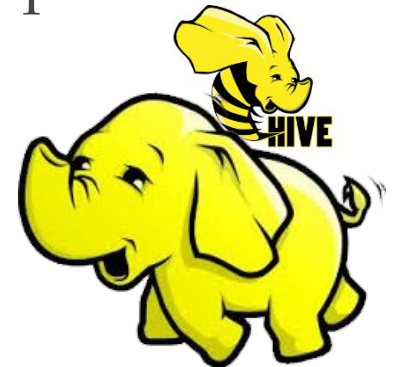
# Hive Query Language

- Hive is an alternative/complement to Pig
  - Developed by Facebook around 2007
  - Hive is a "SQL-like" Query language
  - It imposes "Structure" on "Unstructured" data
  - Needs a predefined schema definition
  - It is also extensible, via user-defined functions (UDFs) written in Java or other languages (C, Python, etc.)
- Hive isn't a relational database
  - No transactions, no isolation, no consistency promises
  - Searches and processes Hadoop data stores
  - Not suitable for real-time queries and row-level updates
  - Generally much higher latency than a DBMS, but higher performance
  - Best for batch jobs over large "immutable" data



# Hive Usage

- Hive is best used to perform analyses and summaries over large data sets
- Hive requires a meta-store to keep information about virtual tables
- It evaluates query plans, selects the most promising one, and then evaluates it using a series of map-reduce functions
- Hive is best used to *answer a single instance of a specific question* whereas Pig is best used to accomplish *frequent reorganization, combining, and reformatting* tasks





# *Hive Interface*

- Hive is similar to SQL-92
- Based on familiar database concepts, tables, rows, columns, and schemas
- Makes "Big Data" appear as tables on the fly
- Like Pig, Hive has a command-line shell

```
$ hive  
hive>
```

- Or it can execute scripts

```
$ hive -f myquery.hive
```

- There are also GUIs



# Defining Hive Tables

- A Hive table consists of
  - Data linked to a file or multiple files in an HDFS
  - Schema stored as mapping of the data to a set of columns with types
- Schema and Data are separated
  - Allows multiple schemas on the same data

```
$ hive
hive> CREATE TABLE Roster (
    team string,
    year int,
    jersey string,
    player string,
    position string,
    starts int,
    games int)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
STORED AS TEXTFILE;
```



# *Operations on Hive Tables*

<b>Table Operation</b>	<b>Command Syntax</b>
See current tables	hive> SHOW TABLES;
Check schema	hive> DESCRIBE Roster;
Change table name	hive> ALTER TABLE Roster RENAME TO PlayedFor;
Add a column	hive> ALTER TABLE Roster ADD COLUMNS (pid, int);
Drop a partition	hive> ALTER TABLE Roster DROP PARTITION (team='dallas-cowboys');



# *Loading Hive Tables*

- Use `LOAD DATA` to import data into a HIVE table

```
$ hive
hive> LOAD DATA LOCAL INPATH 'comp521/NFLrosters'
      INTO TABLE Roster;
```

- No files are modified by Hive, the schema simply imposes structure on the file as it is read
- You can use the keyword `OVERWRITE` to modify previous loaded files

```
hive> LOAD DATA INPATH 'comp521/NFLrosters'
hive>      OVERWRITE INTO TABLE Rosters;
hive> INSERT INTO recentQBs
hive>      SELECT * FROM Rosters
hive>      WHERE position = 'QB' AND year > 2000
hive>      AND starts > 0;
```





# Hive Queries

- SELECT

```
$ hive
```

```
hive> SELECT * FROM recentQBs WHERE name = "*Brady";
```

- Supports the following:

- WHERE clause
- UNION ALL
- DISTINCT
- GROUP BY and HAVING
- LIMIT
- JOIN,
- LEFT OUTER JOIN, RIGHT OUTER JOIN, OUTER JOIN
  - Returned rows are random, and may vary between calls



# Hive Query Examples

```
hive> SELECT * FROM customers;
hive> SELECT COUNT(*) FROM customers;
hive>
hive> SELECT first, last, address, zip FROM customers
hive> WHERE orderID > 0
hive> GROUP BY zip;
hive>
hive> SELECT customers.*, orders.*
hive> FROM customers JOIN orders
hive>     ON (customers.customerID = orders.customerID);
hive>
hive> SELECT customers.*, orders.*
hive> FROM customers LEFT OUTER JOIN orders
hive>     ON (customers.customerID = orders.customerID);
```

- If you understand SQL, you should be able to follow
- Note: These are queries, not transactions
- The data's state could change between and within a query



# *Hive Subqueries*

- Hive allows subqueries only within FROM clauses

```
hive> SELECT sid, mid, total FROM  
hive>     (SELECT sid, mid, refCnt + altCnt AS total  
hive>     FROM genotype) gtypeTotals  
hive> WHERE total > 20;
```

- Subqueries are generally materialized (computed and saved as hive tables)
- You **MUST** to include a name for the subquery result table
- The columns of a subquery's SELECT list are available to the outer query



# *Sorting in Hive*

---

- Hive supports ORDER BY, but its result differs from SQL's
  - Only one Reduce step is applied and partial results are broadcast and combined
  - No need for any intermediate files
  - This allows optimization to a single MapReduce step
- Hive also supports SORT BY with multiple fields
  - Produces a "total ordering" of all results
  - Might require multiple MapReduce operations
  - Might materialize several intermediate tables



# Summary

- There are two primary "high-level" programming languages for Hadoop-- Pig and Hive
- *Pig* is a "scripting language" that excels in specifying a processing pipeline that is automatically parallelized into Map-Reduce operations
  - Deferred execution allows for optimizations in scheduling Map-Reduce operations
  - Good for general data manipulation and cleaning
- *Hive* is a "query language" that borrows heavily from SQL, good for searching and summarizing data
  - Requires the specification of an "external" schema
  - Often materializes many more intermediate results than a DBMS would

