Data-intensive computing systems

High-Level Languages

University of Verona
Computer Science Department

Damiano Carra

Acknowledgements

- *Credits*
  - *Part of the course material is based on slides provided by the following authors*
    - *Pig/Pig Latin* → *Pietro Michiardi, Jimmy Lin*
    - *Hive* → *Dhruba Borthakur, Zheng Shao, Liyin Tang*

Need for High-Level Languages

- Hadoop is great for large-data processing!
  - But writing Java programs for everything is verbose and slow
  - Custom code required even for basic operations
    - Projection and Filtering need to be “rewritten” for each job
    - Code is difficult to reuse and maintain
    - Optimizations are difficult due to opacity of Map and Reduce
  - Data scientists don’t want to write Java

- Solution: develop higher-level data processing languages
  - Pig: Pig Latin is a bit like Perl
  - Hive: HQL is like SQL

Pig and Hive

- 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
- 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
- Common idea:
  - Provide higher-level language to facilitate large-data processing
  - Higher-level language “compiles down” to Hadoop jobs

Pig: Introduction and Motivations

Use Cases: Rollup aggregates

- Compute aggregates against user activity logs, web crawls, etc.
  - Example: compute the frequency of search terms aggregated over days, weeks, month
  - Example: compute frequency of search terms aggregated over geographical location, based on IP addresses
- Requirements
  - Successive aggregations
  - Joins followed by aggregations
- Pig vs. OLAP systems
  - Datasets are too big
  - Data curation is too costly

Use Cases: Temporal Analysis

- ❑ Study how search query distributions change over time
  - Correlation of search queries from two distinct time periods (groups)
  - Custom processing of the queries in each correlation group
- ❑ Pig supports operators that minimize memory footprint
  - Instead, in a RDBMS such operations typically involve JOINS over very large datasets that do not fit in memory and thus become slow

Use Cases: Session Analysis

- ❑ Study sequences of page views and clicks
- ❑ Example of typical aggregates
  - Average length of user session
  - Number of links clicked by a user before leaving a website
  - Click pattern variations in time
- ❑ Pig supports advanced data structures, and UDFs

Pig Latin

- Pig Latin, a high-level programming language developed at Yahoo!
  - Combines the best of both declarative and imperative worlds
    - High-level declarative querying in the spirit of SQL
    - Low-level, procedural programming á la MapReduce
- Pig Latin features
  - Multi-valued, nested data structures instead of flat tables
  - Powerful data transformations primitives, including joins
- Pig Latin program
  - Made up of a series of operations (or transformations)
  - Each operation is applied to input data and produce output data

  → A Pig Latin program describes a data flow

Example – Pig Latin premiere

- Assume we have the following table:

```
urls:  (url, category, pagerank)
```

Where:

- `url`: is the url of a web page
- `category`: corresponds to a pre-defined category for the web page
- `pagerank`: is the numerical value of the pagerank associated to a web page

- Problem
  - Find, for each sufficiently large category, the average page rank of high-pagerank urls in that category

Example – Solution in SQL

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


Example – Solution in Pig Latin

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

Pig Execution environment

- How do we go from Pig Latin to MapReduce?
  - The Pig system is in charge of this
  - Complex execution environment that interacts with Hadoop MapReduce
  - → The programmer focuses on the data and analysis

- Pig Compiler
  - Pig Latin operators are translated into MapReduce code
  - NOTE: in some cases, hand-written MapReduce code performs better

- Pig Optimizer
  - Pig Latin data flows undergo an (automatic) optimization phase
  - These optimizations are borrowed from the RDBMS community

Pig Latin

Introduction

- Not a complete reference to the Pig Latin language: refer to the Pig Latin wiki
  - Here we cover some interesting aspects
- The focus here is on some language primitives
  - Optimizations are treated separately
  - How they can be implemented is covered later

Data Model

- Supports four types
  - Atom: contains a simple atomic value as a string or a number
    - e.g. `‘alice’`
  - Tuple: sequence of fields, each can be of any data type
    - e.g., `(‘alice’, ‘lakers’)`
  - Bag: collection of tuples with possible duplicates. Flexible schema, no need to have the same number and type of fields
    - Tuples can be nested
    - e.g., $\left\{ \begin{array}{c} (\texttt{'alice'},\ \texttt{'lakers'}) \\ (\texttt{'alice'},\ (\texttt{'ipod'},\texttt{'apple'})) \end{array} \right\}$

Data Model

- Supports four types (cont'd)
  - Map: collection of data items, where each item has an associated key for lookup. The schema, as with bags, is flexible.
    - NOTE: keys are required to be data atoms, for efficient lookup.
    - e.g.,

$$\left[\begin{array}{ccc} \texttt{'fan of'} & \rightarrow & \left\{\begin{array}{c} (\texttt{'lakers'}) \\ (\texttt{'ipod'}) \end{array}\right\} \\ \texttt{'age'} & \rightarrow & \texttt{20} \end{array}\right]$$

    - The key '`fan of`' is mapped to a bag containing two tuples
    - The key '`age`' is mapped to an atom
  - Maps are useful to model datasets in which schema may be dynamic (over time)

Structure

- Pig latin programs are a sequence of steps
  - Can use an interactive shell (called `grunt`)
  - Can feed them as a "script"
- Comments
  - In line: with double hyphens (- -)
  - C-style for longer comments (/* ... */)
- Reserved keywords
  - List of keywords that can't be used as identifiers
  - Same old story as for any language

Expressions

- An expression is something that is evaluated to yield a value

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

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	$\left[\text{'age'} \rightarrow 20 \right]$
Projection	f2.$0	$\left\{ \begin{array}{c} (\text{'lakers'}) \\ (\text{'iPod'}) \end{array} \right\}$
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

Loading and storing data

- The first step in a Pig Latin program is to load data
  - What input files are
  - How the file contents are to be deserialized
  - An input file is assumed to contain a sequence of tuples

- Data loading is done with the `LOAD` command

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

Loading and storing data

- The previous example specifies the following:
  - The input file is `query_log.txt`
  - The input file should be converted into tuples using the custom `myLoad` deserializer
  - The loaded tuples have three fields, specified by the schema

- Optional parts
  - `USING` clause is optional: if not specified, the input file is assumed to be plain text, tab-delimited
  - `AS` clause is optional: if not specified, must refer to fields by position instead of by name

Loading and storing data

- Return value of the `LOAD` command
  - Handle to a bag
  - This can be used by subsequent commands
  - → bag handles are only logical
  - → no file is actually read!

- The command to write output to disk is `STORE`
  - It has similar semantics to the `LOAD` command

Per-tuple processing: Filtering data

- Once you have some data loaded into a relation, the next step is to filter it
  - This is done, e.g., to remove unwanted data
  - HINT: By filtering early in the processing pipeline, you minimize the amount of data flowing trough the system

- A basic operation is to apply some processing over every tuple of a data set
  - This is achieved with the `FOREACH` command

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


Per-tuple processing: Filtering data

- Comments on the previous example:
  - Each tuple of the bag queries should be processed independently
  - The second field of the output is the result of a UDF
- Semantics of the FOREACH command
  - There can be no dependence between the processing of different input tuples
  - → This allows for an efficient parallel implementation
- Semantics of the GENERATE clause
  - Followed by a list of expressions
  - Also flattering is allowed
    - This is done to eliminate nesting in data
    - → Allows to make output data independent for further parallel processing
    - → Useful to store data on disk

Per-tuple processing: Discarding unwanted data

- A common operation is to retain a portion of the input data
  - This is done with the `FILTER` command

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

- Filtering conditions involve a combination of expressions
  - Comparison operators
  - Logical connectors
  - UDF

Per-tuple processing: Streaming data

- The `STREAM` operator allows transforming data in a relation using an external program or script
  - This is possible because Hadoop MapReduce supports "streaming"
  - Example:

```
C = STREAM A THROUGH 'cut -f 2';
```

which use the Unix `cut` command to extract the second filed of each tuple in `A`

- The STREAM operator uses `PigStorage` to serialize and deserialize relations to and from `stdin/stdout`
  - Can also provide a custom serializer/deserializer
  - Works well with python

Getting related data together

- It is often necessary to group together tuples from one or more data sets
  - GROUP command
- Example: Assume we have loaded two relations

```
results:   (queryString, url, position)
revenue:   (queryString, adSlot, amount)
```

  - `results` contains, for different query strings, the urls shown as search results, and the positions at which they where shown
  - `revenue` contains, for different query strings, and different advertisement slots, the average amount of revenue
- To find the total revenue for each query string, we can

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


`JOIN` in Pig Latin

- In many cases, the typical operation on two or more datasets amounts to a join
  - IMPORTANT NOTE: large datasets that are suitable to be analyzed with Pig (and MapReduce) are generally not normalized

  → `JOIN`s are used more infrequently in Pig Latin than they are in SQL
- The syntax of a JOIN

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

  - This is a classic join, where each match between the two relations corresponds to a row in the join result

MapReduce in Pig Latin

- ❑ It is trivial to express MapReduce programs in Pig Latin
  - This is achieved using `GROUP` and `FOREACH` statements
  - A map function operates on one input tuple at a time and outputs a bag of key-value pairs
  - The reduce function operates on all values for a key at a time to produce the final result
- ❑ Example

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

  - where `map()` and `reduce()` are UDF

Validation and nulls

- ❑ Pig does not have the same power to enforce constraints on schema at load time as a RDBMS
  - If a value cannot be cast to a type declared in the schema, then it will be set to a `null` value
  - This also happens for corrupt files
- ❑ A useful technique to partition input data to discern good and bad records
  - Use the `SPLIT` operator

```
SPLIT records INTO good_records IF temperature is not null, bad
_records IF temperature is NULL;
```

Statements

- ❑ As a Pig Latin program is executed, each statement is parsed
  - The interpreter builds a logical plan for every relational operation
  - The logical plan of each statement is added to that of the program so far
  - Then the interpreter moves on to the next statement

- ❑ IMPORTANT: No data processing takes place during construction of logical plan
  - When the interpreter sees the first line of a program, it confirms that it is syntactically and semantically correct
  - Then it adds it to the logical plan
  - It does not even check the existence of files, for data load operations

Statements

→ It makes no sense to start any processing until the whole flow is defined
  - Indeed, there are several optimizations that could make a program more efficient (e.g., by avoiding to operate on some data that later on is going to be filtered)

- ❑ The trigger for Pig to start execution are the `DUMP` and `STORE` statements
  - It is only at this point that the logical plan is compiled into a physical plan

- ❑ How the physical plan is built
  - Pig prepares a series of MapReduce jobs
    - In Local mode, these are run locally on the JVM
    - In MapReduce mode, the jobs are sent to the Hadoop Cluster
  - IMPORTANT: The command `EXPLAIN` can be used to show the MapReduce plan

Statements: Multi-query execution

- There is a difference between `DUMP` and `STORE`
  - `DUMP` → stdout
    - Can be used for diagnosis
  - `STORE` → file
    - Allows for program/job optimizations

- Main optimization objective: minimize I/O
  - Consider the following example:

```
A = LOAD 'input/pig/multiquery/A';
B = FILTER A BY $1 == 'banana';
STORE B INTO 'output/b';
C = FILTER A BY $1 != 'banana';
STORE C INTO 'output/c';
```


Statements: Multi-query execution (cont'd)

- In the example, relations `B` and `C` are both derived from `A`
  - Naively, this means that at the first `STORE` operator the input should be read
  - Then, at the second `STORE` operator, the input should be read again

- Pig will run this as a single MapReduce job
  - Relation `A` is going to be read only once
  - Then, each relation `B` and `C` will be written to the output

- If we use `DUMP` instead of `STORE`, Pig is forced to run two different MapReduce jobs
  - Waste of resources

Hadoop Hive

– Quick overview –

Motivation

- ❑ Limitation of MR
  - Have to use M/R model
  - Not Reusable
  - Error prone
  - For complex jobs:
    - Multiple stage of Map/Reduce functions
    - Just like ask developers to specify physical execution plan in the database

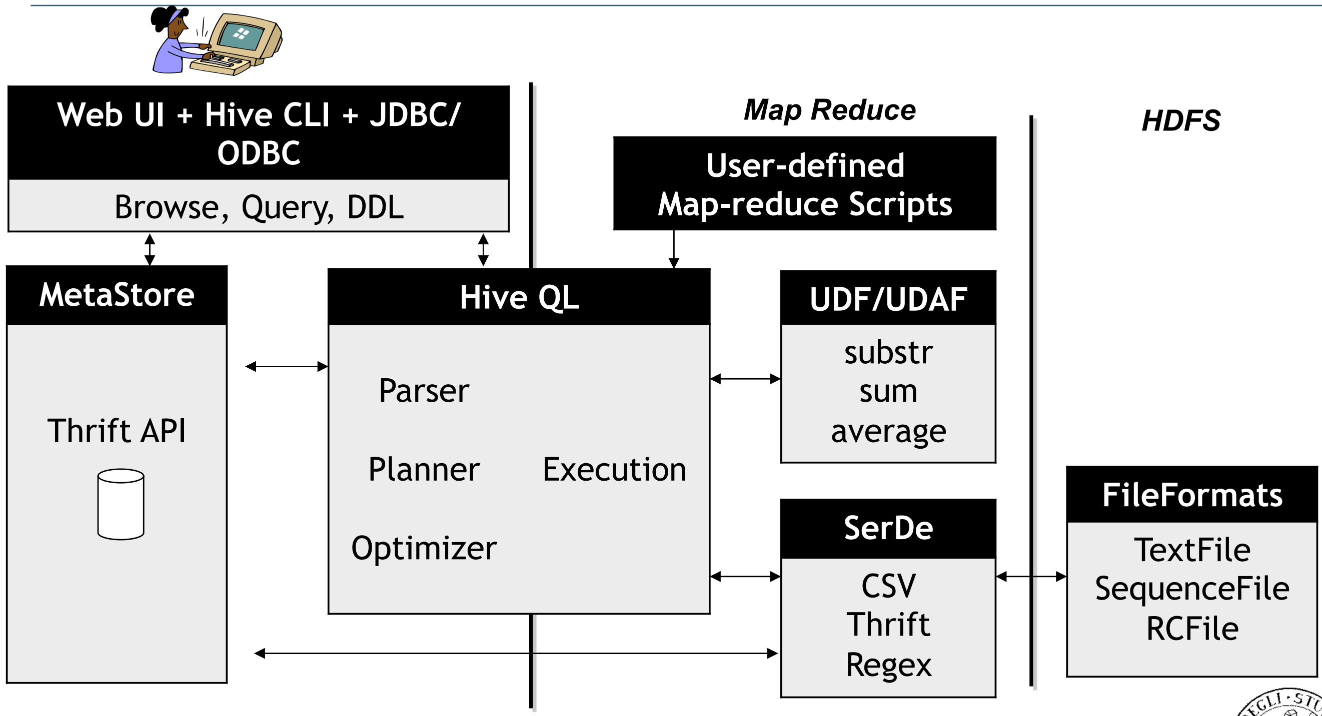
Overview

- ❑ Intuitive
  - Make the unstructured data looks like tables regardless how it really lay out
  - SQL based query can be directly against these tables
  - Generate specific execution plan for this query
- ❑ What's Hive
  - A data warehousing system to store structured data on Hadoop file system
  - Provide an easy query these data by execution Hadoop MapReduce plans

Hive Components

- ❑ Shell Interface: Like the MySQL shell
- ❑ Driver:
  - Session handles, fetch, execution
- ❑ Complier:
  - Parse, plan, optimize
- ❑ Execution Engine:
  - DAG stage
  - Run map or reduce

Hive Architecture



Data Model

- ❑ Tables
  - Basic type columns (int, float, boolean)
  - Complex type: List / Map (associative array)
- ❑ Partitions
- ❑ Buckets

- ❑ Example

```
CREATE TABLE sales(
    id INT,
    items ARRAY<STRUCT<id:INT,name:STRING>>
)PARITIONED BY (ds STRING)
CLUSTERED BY (id) INTO 32 BUCKETS;

SELECT id FROM sales TABLESAMPLE (BUCKET 1 OUT OF 32)
```

Pros and Cons

- Pros
  - A easy way to process large scale data
  - Support SQL-based queries
  - Provide more user defined interfaces to extend
  - Programmability
  - Efficient execution plans for performance
  - Interoperability with other database tools
- Cons
  - No easy way to append data
  - Files in HDFS are immutable

Application

- Log processing
  - Daily Report
  - User Activity Measurement
- Data/Text mining
  - Machine learning (Training Data)
- Business intelligence
  - Advertising Delivery
  - Spam Detection

Hive Usage @ Facebook

- ❑ Statistics per day:
  - 4 TB of compressed new data added per day
  - 135TB of compressed data scanned per day
  - 7500+ Hive jobs on per day

- ❑ Hive simplifies Hadoop:
  - ~200 people/month run jobs on Hadoop/Hive
  - Analysts (non-engineers) use Hadoop through Hive
  - 95% of jobs are Hive Jobs