Hadoop Ecosystem

Corso di Sistemi e Architetture per Big Data
A.A. 2016/17

Valeria Cardellini
Why an ecosystem

• Hadoop released in 2011 by Apache Software Foundation
• A platform around which an entire ecosystem of capabilities has been and is built
  – Dozens of self-standing software projects (some are top projects), each addressing a variety of Big Data space and meeting different needs
• It is an ecosystem: complex, evolving, and not easily parceled into neat categories
Hadoop ecosystem: a partial big picture

See https://hadoopecosystemtable.github.io for a longer list
Some products in the ecosystem

• Distributed file systems
  – **HDFS**, GlusterFS, Lustre, Alluxio, …

• Distributed programming
  – **Apache MapReduce**, Apache Pig, Apache Storm, Apache Spark, Apache Flink, …
  – Pig: simplifies development of applications employing MapReduce
  – Spark: improves performance for certain types of Big Data applications
  – Storm and Flink: stream processing

• NoSQL data stores (various models)
  – (column data model) **Apache Hbase**, Cassandra, Accumulo, …
  – (document data model) **MongoDB**, …
  – (key-value data model) **Redis**, …
  – (graph data model) **neo4j**, …
Some products in the ecosystem

• NewSQL databases
  – InfluxDB, …

• SQL-on-Hadoop
  – Apache Hive: SQL-like language
  – Apache Drill: interactive data analysis and exploration (inspired by Google Dremel)
  – Presto: distributed SQL query engine open sourced by Facebook

• Data ingestion
  – Apache Flume, Apache Sqoop, Apache Kafka, Apache Samza, …

• Service programming
  – Apache Zookeeper, Apache Thrift, Apache Avro, …
  – Apache Avro: framework for modeling, serializing and making RPC
Some products in the ecosystem

- **Scheduling**
  - **Apache Oozie**: workflow scheduler system for MR jobs using DAGs
  - ...

- **Machine learning**
  - Apache Mahout: machine learning library and math library, on top of MapReduce
  - ...

- **System development**
  - **Apache Mesos, YARN**
  - Apache Ambari: Hadoop management web UI
The reference Big Data stack

High-level Interfaces

Data Processing

Data Storage

Resource Management

Support / Integration
Apache Pig: motivation

• Big Data
  – 3V: from multiple sources and in different formats, data sets typically huge
  – No need to alter the original data, just to do reads
  – Data may be temporary; could discard the data set after analysis

• Data analysis goals
  – Quick
    • Exploit parallel processing power of a distributed system
  – Easy
    • Write a program or query without a huge learning curve
    • Have some common analysis tasks predefined
  – Flexible
    • Transforms dataset into a workable structure without much overhead
    • Performs customized processing
  – Transparent
Apache Pig: solution

• High-level data processing built on top of MapReduce which makes it easy for developers to write data analysis scripts
  – Initially developed by Yahoo!
• Scripts translated into MapReduce (MR) programs by the Pig compiler
• Includes a high-level language (Pig Latin) for expressing data analysis program
• Uses MapReduce to execute all data processing
  – Compiles Pig Latin scripts written by users into a series of one or more MapReduce jobs that are then executed
Pig Latin

- Set-oriented and procedural data transformation language
  - Primitives to filter, combine, split, and order data
  - Focus on data flow: no control flow structures like for loop or if structures
  - Users describe transformations in steps
  - Each set transformation is stateless

- Flexible data model
  - Nested bags of tuples
  - Semi-structured data types

- Executable in Hadoop
  - A compiler converts Pig Latin scripts to MapReduce data flows
Pig script compilation and execution

• Programs in Pig Latin are firstly parsed for syntactic and instance checking
  – The output from this parser is a logical plan, arranged in a DAG allowing logical optimizations

• Logical plan compiled by a MR compiler into a series of MR statements

• Then further optimization by a MR optimizer performing tasks such as early partial aggregation, using the MR combiner function

• Finally, MR program submitted to Hadoop job manager for execution
Pig: the big picture

Pig

- Pig Latin Script
- User-Defined Functions

Compile
Optimize

Map-Reduce Statements

Write Results
Read Data

Hadoop

Valeria Cardellini - SABD 2016/17
Pig: pros

• **Ease of programming**
  – Complex tasks comprised of multiple interrelated data transformations encoded as data flow sequences, making them easy to write, understand, and maintain
  – Decrease in development time

• **Optimization opportunities**
  – The way in which tasks are encoded permits the system to optimize their execution automatically
  – Focus on semantics rather than efficiency

• **Extensibility**
  – Supports user-defined functions (UDFs) written in Java, Python and Javascript to do special-purpose processing
Pig: cons

- Slow start-up and clean-up of MapReduce jobs
  - It takes time for Hadoop to schedule MR jobs

- Not suitable for interactive OLAP analytics
  - When results are expected in < 1 sec

- Complex applications may require many UDFs
  - Pig loses its simplicity over MapReduce

- Debugging: some produced errors caused by UDFs not helpful
Pig Latin: data model

- Atom: simple atomic value (i.e., number or string)
- Tuple: sequence of fields; each field any type
- Bag: collection of tuples
  - Duplicates possible
  - Tuples in a bag can have different field lengths and field types
- Map: collection of key-value pairs
  - Key is an atom; value can be any type
LOAD

- Input is assumed to be a bag (sequence of tuples)
- Can specify a serializer with “USING”
- Can provide a schema with “AS”

```java
newBag = LOAD 'filename'
<USING functionName()>
<AS (fieldName1, fieldName2,...)>;
```
**Speaking Pig Latin**

**FOREACH**

- Apply some processing to each tuple in a bag
- Each field can be:
  - A fieldname of the bag
  - A constant
  - A simple expression (ie: f1+f2)
  - A predefined function (ie: SUM, AVG, COUNT, FLATTEN)
  - A UDF (ie: tax(gross, percentage))

```plaintext
newBag = FOREACH bagName
  GENERATE field1, field2, ...;
```

- **GENERATE**: used to define the fields and generate a new row from the original
FILTER

• Select a subset of the tuples in a bag
  \[ \text{newBag} = \text{FILTER bagName BY expression}; \]

• Expression uses simple comparison operators (==, !>, <, >, ...) and logical connectors (AND, NOT, OR)
  \[ \text{some_apples} = \text{FILTER apples BY colour} ! = 'red'; \]

• Can use UDFs
  \[ \text{some_apples} = \text{FILTER apples BY NOT isRed(colour)}; \]
GROUP

• Groups together tuples that have the same group key
  
  \[
  \text{newBag} = \text{GROUP} \ \text{bagName} \ \text{BY} \ \text{expression};
  \]

• Usually the expression is a field
  
  \[
  \text{stat1} = \text{GROUP} \ \text{students} \ \text{BY} \ \text{age};
  \]

• Expression can use operators
  
  \[
  \text{stat2} = \text{GROUP} \ \text{employees} \ \text{BY} \ \text{salary} + \ \text{bonus};
  \]

• Can use UDFs
  
  \[
  \text{stat3} = \text{GROUP} \ \text{employees} \ \text{BY} \ \text{netsal(salary, taxes)};
  \]
**Speaking Pig Latin**

**JOIN**

- Join two datasets by a common field

\[
\text{joined_data} = \text{JOIN results BY queryString, revenue BY queryString}
\]
Pig script for WordCount

data = LOAD ‘input.txt’ as (lines:chararray);
words = FOREACH data GENERATE FLATTEN(tokenize(lines)) AS word;
wordGroup = GROUP words BY word;
counts = FOREACH wordGroup GENERATE group COUNT(words);
STORE counts into ‘counts’;

See http://bit.ly/2q5kZpH
Pig: how is it used in practice?

- Useful for computations across large, distributed datasets
- Abstracts away details of execution framework
- Users can change order of steps to improve performance
- Used in tandem with Hadoop and HDFS
  - Transformations converted to MapReduce data flows
  - HDFS tracks where data is stored
- Operations scheduled nearby their data
Hive: motivation

• Analysis of data made by both engineering and non-engineering people

• Data are growing faster and faster
  – Relational DBMS cannot handle them (limits on table size, depending also on file size constraints imposed by operating system)
  – Traditional solutions are often not scalable, expensive and proprietary

• Hadoop supports data-intensive distributed applications

• But... you have to use MapReduce model:
  – Hard to program
  – Not reusable
  – Error prone
  – Can require multiple stages of MapReduce jobs
  – Most users know SQL
Hive: solution

• Makes the unstructured data looks like tables regardless how it really lays out
• SQL-based query can be directly against these tables
• Generates specify execution plan for this query
• Hive
  – A big data management system storing structured data on HDFS
  – Provides an easy querying of data by executing Hadoop MapReduce programs
  – Can be also used on top of Spark (Hive on Spark)
What is Hive?

• A data warehouse built on top of Hadoop to provide data summarization, query, and analysis
  – Initially developed by Facebook

• Structure
  – Access to different storage
  – HiveQL (very close to a subset of SQL)
  – Query execution via MapReduce

• Key building principles
  – SQL is a familiar language
  – Extensibility: types, functions, formats, scripts
  – Performance
Hive: application scenario

- No real-time queries (high latency)
- No row-level updates
- Not designed for online transaction processing
- Best use: batch processing over large sets of immutable data
  - Log processing
  - Data/text mining
  - Business intelligence
Hive deployment

• To deploy Hive, you also need to deploy a **metastore service**
  – To store the metadata for Hive tables and partitions in a relational database, and provides Hive access to this information

• To deploy the metastore service you need to do install mysql server which will be used to store the metastore information
Example with Amazon EMR

• Launch an Amazon EMR cluster and run a Hive script to analyze a series of web log files
  http://amzn.to/2pCQe8v
Example with Amazon EMR

• Create a Hive table

CREATE EXTERNAL TABLE IF NOT EXISTS cloudfront_logs (  
    DateObject Date,  
    Time STRING,  
    Location STRING,  
    Bytes INT,  
    RequestIP STRING,  
    Method STRING,  
    Host STRING,  
    Uri STRING,  
    Status INT,  
    Referrer STRING,  
    OS String,  
    Browser String,  
    BrowserVersion String  
)
Example with Amazon EMR

- The Hive script:
  - Create the `cloudfront_logs` table
  - Load the log files into the `cloudfront_logs` table parsing the log files using the regular expression serializer/deserializer (RegEx SerDe)
  - Submit a query in HiveQL to retrieve the total number of requests per operating system for a given time frame
    ```sql
    SELECT os, COUNT(*) count FROM cloudfront_logs
    WHERE date BETWEEN '2014-07-05' AND '2014-08-05' GROUP BY os;
    ```
  - Write the query result to Amazon S3
Performance evaluation of high-level interfaces

- Compare hand-coded Java MR jobs, Pig Latin, Hive QL and JAQL
- JAQL: functional data processing and query language most commonly used for JSON query processing on Big Data

Source: “Comparing High Level MapReduce Query Languages”, 2011.
Performance evaluation of high-level interfaces

• Results from “Comparing High Level MapReduce Query Languages” (2011)
  – HiveQL scaled best and hand-coded Java MR jobs are only slightly faster
  – Java also had better scale-up performance than Pig
  – Pig and JAQL scaled the same except when using joins: Pig significantly outperformed JAQL on that regard
  – However, this study considered simple MR jobs with small jobs and Pig definitely suffered from the overhead to launch them due to JVM setup

• But the performance gap between Java MR jobs and Pig almost disappears for complex MR jobs
Managing complex jobs

• How to simplify the management of complex Hadoop jobs?

• How to manage a recurring query?
  – i.e., a query that repeats periodically
  – Naïve approach: manually re-issue the query every time it needs to be executed
    • Lacks convenience and system-level optimizations
Apache Oozie

- Workflow scheduler to write scripts for automatic scheduling of jobs
  - On top of Hadoop
- Java web app that runs in a Java servlet-container
- Integrated with the rest of Hadoop stack ecosystem: supports different types of jobs
  - E.g., Java MapReduce, Pig, Hive, Streaming MapReduce
Oozie workflow

• A **workflow** is a collection of actions (i.e. Hadoop MapReduce jobs, Pig jobs) arranged in a control dependency **DAG** (Direct Acyclic Graph)
  – Control dependency from one action to another means that the second action can't run until the first action has completed

• A **Coordinator job** is a recurrent Oozie workflow job triggered by time (frequency) and data availability

• Workflow definition written in hPDL (a XML Process Definition Language)
Oozie workflow

• **Control flow nodes** in the workflow
  – Define beginning and end of a workflow (start, end and fail nodes)
  – Provide a mechanism to control the workflow execution path (decision, fork and join)

• **Action nodes** in the workflow
  – Mechanism by which a workflow triggers the execution of a computation/processing task
  – Can be extended to support additional type of actions

• Oozie workflows can be parameterized using variables like `${inputDir}` within the workflow definition
  – If properly parameterized (i.e. using different output directories) several identical workflow jobs can concurrently run
Oozie workflow: example

- Example of Oozie workflow: Wordcount
Oozie workflow: example

<workflow-app name='wordcount-wf'
xmlns="uri:oozie:workflow:0.1">
  <start to='wordcount'/>
  <action name='wordcount'>
    <map-reduce>
      <job-tracker>${jobTracker}</job-tracker>
      <name-node>${nameNode}</name-node>
      <configuration>
        <property>
          <name>mapred.mapper.class</name>
          <value>org.myorg.WordCount.Map</value>
        </property>
        <property>
          <name>mapred.reducer.class</name>
          <value>org.myorg.WordCount.Reduce</value>
        </property>
      </configuration>
    </map-reduce>
  </action>
</workflow-app>
Oozie workflow: example

```xml
<property>
  <name>mapred.input.dir</name>
  <value>${inputDir}</value>
</property>

<property>
  <name>mapred.output.dir</name>
  <value>${outputDir}</value>
</property>

</configuration>
</map-reduce>
<ok to='end'/>
<error to='end'/>
</action>
<kill name='kill'>
  <message>Something went wrong:
    ${wf:errorCode('wordcount')}</message>
</kill/>
<end name='end'/>
</workflow-app>
```
Oozie: fork and join

- A fork node splits one path of execution into multiple concurrent paths of execution.
- A join node waits until every concurrent execution path of a previous fork node arrives to it.
- The fork and join nodes must be used in pairs.
- The join node assumes concurrent execution paths are children of the same fork node.

**Main workflow**

-start
  - Decision: Input exists?
  - Yes: Fork
  - No: Fork
  - Fork: Sub-workflow [hive actions]
  - OnError: Run report [java main action]
  - OnError: Notify via email [email action]
  - Join
  - END
  - Kill
Oozie: fork and join example

```xml
<workflow-app name="sample-wf" xmlns="uri:oozie:workflow:0.1">
  ...
  <fork name="forking">
    <path start="firstparalleljob"/>
    <path start="secondparalleljob"/>
  </fork>
  <action name="firstparalleljob">
    <map-reduce>
      <job-tracker>foo:9001</job-tracker>
      <name-node>bar:9000</name-node>
      <job-xml>job1.xml</job-xml>
    </map-reduce>
    <ok to="joining"/>
    <error to="kill"/>
  </action>
  <action name="secondparalleljob">
    <map-reduce>
      <job-tracker>foo:9001</job-tracker>
      <name-node>bar:9000</name-node>
      <job-xml>job2.xml</job-xml>
    </map-reduce>
    <ok to="joining"/>
    <error to="kill"/>
  </action>
  <join name="joining" to="nextaction"/>
  ...
</workflow-app>
```
