6CS030 Lecture 8

SQL on Hadoop

Hadoop Stack '**Zoo**':

- Apache Spark
- Apache HBase
- Apache Pig
- Apache Hive

SQL on Hadoop

- MapReduce is very complex when compared to SQL
- Need for a more database-like setup on top of Hadoop
- Various projects can be used on top of Hadoop
  - See http://hadoop.apache.org/ for a list.
- Sometimes referred to as a "Zoo"
- ZooKeeper provides a high-performance coordination service for distributed applications

SQL on Hadoop

- HBase
  - https://hbase.apache.org
- Pig
  - https://pig.apache.org/
- Hive
  - https://hive.apache.org/
- Spark
  - https://spark.apache.org/

HBase

- First Hadoop database inspired by Google's Bigtable
- Runs on top of HDFS
- NoSQL like data storage platform
  - No typed columns, triggers, advanced query capabilities, etc.
- Offers a simplified structure and query language in a way that is highly scalable and can tackle large volumes

HBase

- Similar to RDBMS, HBase organizes data in tables with rows and columns
- HBase table consists of multiple rows
- A row consists of a row key and one or more columns with values associated with them
- Rows in a table are sorted alphabetically by the row key

HBase

- Each column in HBase is denoted by a column family and qualifier (separated by a colon, ':')
- A column family physically co-locates a set of columns and their values
- Every row has the same column families, but not all column families need to have a value per row
- Each cell in a table is hence defined by a combination of the row key, column family and column qualifier, and a timestamp

HBase

Creating a HBASE table with Column Family Schema

```
hbase(main):001:0> create 'users', 'name', 'email'

0 row(s) in 2.8350 seconds
=> Hbase::Table - users
```

HBase `users` Table (Conceptual View)

Row Key	name:first	name:last	email:work	email:personal
`user1`	John	Doe	john.doe@company.com	johnny123@gmail.com
`user2`	Alice	Smith	alice@tech.com	alice.smith@yahoo.com
`user3`	Bob	Brown	bob@startup.io	bob_b@hotmail.com

HBase

Description Result

```
{
  "users": {
    "column_families": {
      "email": {
        "NAME": "email",
        "BLOOMFILTER": "ROW",
        "VERSIONS": "1",
        "IN_MEMORY": "false",
        "KEEP_DELETED_CELLS": "FALSE",
        "DATA_BLOCK_ENCODING": "NONE",
        "TTL": "FOREVER",
        "COMPRESSION": "NONE",
        "MIN_VERSIONS": "0",
        "BLOCKCACHE": "true",
        "BLOCKSIZE": "65536",
        "REPLICATION_SCOPE": "0"
      },
      "name": {
        "NAME": "name",
        "BLOOMFILTER": "ROW",
        "VERSIONS": "1",
        "IN_MEMORY": "false",
        "KEEP_DELETED_CELLS": "FALSE",
        "DATA_BLOCK_ENCODING": "NONE",
        "TTL": "FOREVER",
        "COMPRESSION": "NONE",
        "MIN_VERSIONS": "0",
        "BLOCKCACHE": "true",
        "BLOCKSIZE": "65536",
        "REPLICATION_SCOPE": "0"
      }
    }
  }
}
```

Describing The Created Table

```
hbase(main):002:0> describe 'users'
```

Description Result

```
Table users is ENABLED
users
COLUMN FAMILIES DESCRIPTION
```

HBase

- Now lets, build up on the initial contextual example : HBase table to store and query users
- The row key will be the user id
- column families:qualifiers
  - name:first
  - name:last
  - email (without a qualifier)

HBase

Upserting Value to A HBASE Table

```
put 'users', 'seppe', 'name:last', 'vanden Broucke'
put 'users', 'seppe', 'email:work', 'seppe.vandenbroucke@kuleuven'
```

Scanning A HBASE Table

```
ROW       COLUMN+CELL
seppe     column=email:work,   timestamp=..., value=seppe.vandenbroucke@kuleuven.be
seppe     column=name:first,   timestamp=..., value=Seppe
seppe     column=name:firstt,  timestamp=..., value=Seppe
seppe     column=name:last,    timestamp=..., value=vanden Broucke
```

HBase

Fetching A HBASE Data By Row

```
get 'users', 'seppe'
```

Fetching A HBASE Data By Row : Output

```
COLUMN                CELL
 email:work           timestamp=1495293082872, value=seppe.vandenbroucke@kuleuven.be
 name:first           timestamp=1495293050816, value=Seppe
 name:last            timestamp=1495293067245, value=vanden Broucke
```

HBase

- HBase's query facilities are very limited
- Essentially a key-value, distributed data store with simple get/put operations
- Includes facilities to write MapReduce programs
- HBase (similar to Hadoop) doesn't perform well on less than 5 HDFS DataNodes with an additional NameNode
  - only makes the effort worthwhile when you can invest in, set up and maintain at least 6-10 nodes

Pig

- Yahoo! Developed "Pig", which was made open source as Apache Pig in 2007
- High-level platform for creating programs that run on Hadoop which uses MapReduce underneath
  - The language used is Pig Latin

- Resembles the querying facilities of SQL

Pig

Loading a file using PIG LATIN

```
timesheet = LOAD 'timesheet.csv' USING PigStorage(',');
```

FIltering column 0

```
raw_timesheet = FILTER timesheet BY $0 > 100;
```

Fetching value based on alias

```
timesheet_logged = FOREACH raw_timesheet GENERATE $0 AS driverId, $2 AS hours_logged, $3 AS
miles_logged;
```

Pig

Group BY

```
grp_logged = GROUP timesheet_logged BY driverId;
```

Group BY

```
sum_logged = FOREACH grp_logged GENERATE
    group AS driverId,
    SUM(timesheet_logged.hours_logged) AS sum_hourslogged,
    SUM(timesheet_logged.miles_logged) AS sum_mileslogged;
```

Pig

- RDBMS and SQL are substantially faster than MapReduce – and hence Pig
- Pig Latin is relatively procedural versus declarative SQL

You tell the system how to do something, step-by-step.

Break

Hive

- Initially developed by Facebook but open-sourced afterwards
- Data warehouse solution offering SQL querying facilities on top of Hadoop
- Converts SQL-like queries to a MapReduce pipeline
- Also offers a JDBC and ODBC interface
- Can run on top of HDFS, as well as other file systems

Hive

- Hive **Metastore** stores metadata for each table such as its schema and location on HDFS.
- Driver service is responsible to receive and handle incoming queries
  - query is first converted to an abstract syntax tree, which is then converted to a directed acyclic graph representing an execution plan
  - the directed acyclic graph will contain a number of MapReduce stages and tasks

```
HQL Execution

Query
  ↓
Driver Service
  ↓
Abstract Syntax Tree (AST)
  ↓
Directed Acyclic Graph (DAG)
  ↓
MapReduce Stages
```

Hive

- HiveQL does not completely follow the full SQL-92 standard
  - E.g., lacks strong support for indexes, transactions, materialized views, and only has limited subquery support
- Example:
  SELECT genre, SUM(nrPages) FROM books GROUP BY genre
- HiveQL also allows to query data sets other than structured tables

Hive

```
CREATE TABLE docs (line STRING); -- create a docs table



-- load in file from HDFS to docs table, overwrite existing data:
LOAD DATA INPATH '/testfile.txt' OVERWRITE INTO TABLE docs;



-- perform word count
SELECT word, count(1) AS count
FROM (                                  -- split each line in docs into words
  SELECT explode(split(line, '\s')) AS word FROM docs
) GROUP BY t.word
 ORDER BY t.word;
```

Hive

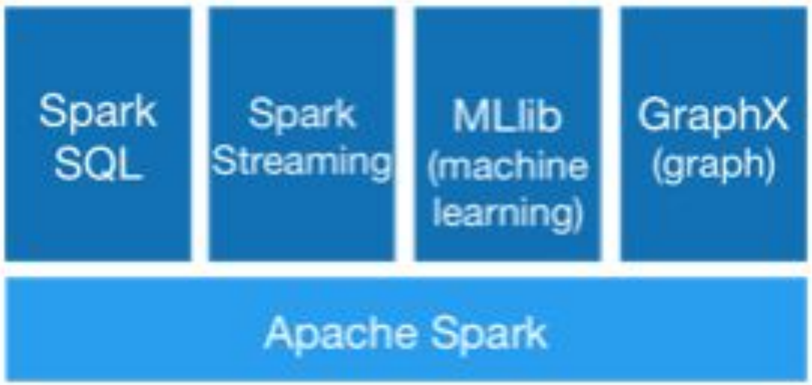
- One difference with traditional RDBMS is that Hive does not enforce the schema at the time of loading the data
  - Hive: schema-on-read
  - RDBMS: schema-on-write
- Recent versions of Hive support full ACID transaction management
- Performance and speed of SQL queries still forms the main disadvantage of Hive today
  - Solutions to bypass MapReduce (e.g. Apache Tez, Cloudera Impala, Facebook Presto)

Apache Spark

- Open-source alternative for MapReduce
- New programming paradigm centered on a data structure called the Resilient Distributed Dataset (RDD)
  - RDDs can enable the construction of iterative programs that have to visit a data set multiple times, as well as more interactive or exploratory programs
  - RDD is a fundamental data structure of Spark
  - Each dataset in RDD is divided into logical partitions that can be computed on different nodes of the cluster
  - Is maintained in a fault tolerant way
  - RDDs can contain any type of Python, Java or Scala objects, including user-defined classes.
- Many orders of magnitude faster than MapReduce implementations
- Rapidly adopted by many Big Data vendors

Apache Spark

- Similar to Hadoop, Spark works with HDFS and requires a cluster manager (e.g. YARN)
- Key components
  - Spark Core
  - Spark SQL
  - MLib, Spark Streaming, GraphX



Spark Core

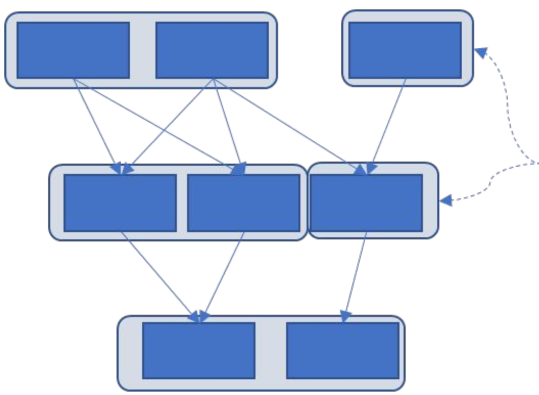
- Foundation for all other components
- Provides functionality for task scheduling and a set of basic data transformations
- Can be used through many programming languages
  - For example: Java, Python, Scala and R
- RDDs are the primary data abstraction in Spark
  - designed to support in-memory data storage and operations, distributed across a cluster
- Can be used to handle JSON and CSV data

Spark Core

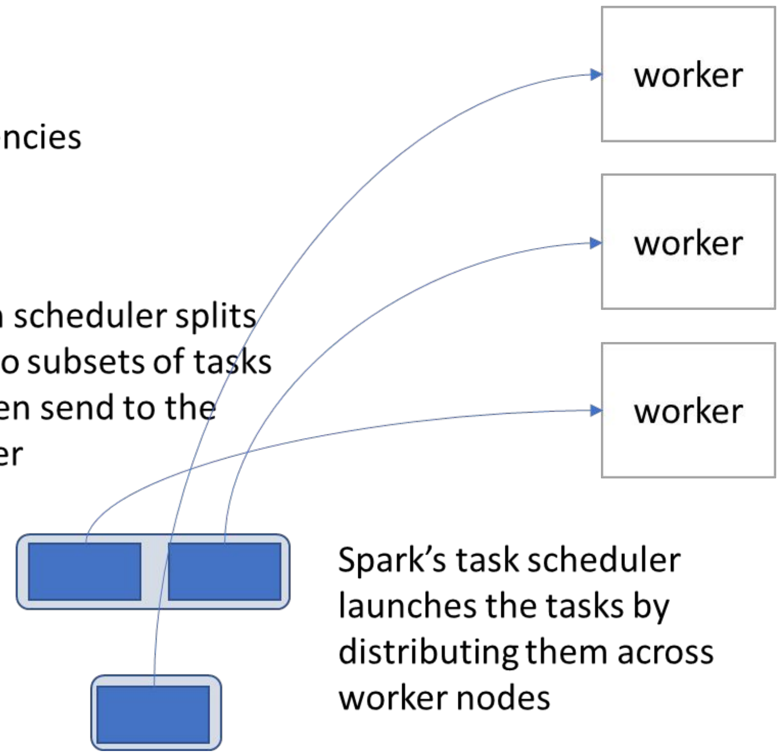
A programmer writes a Spark program using its API:

```
rdd1.join(rdd2).groupBy(…).filter(…)
```

Based on this, Spark builds a directed acyclic graph of operations with their dependencies



Spark's graph scheduler splits the graph into subsets of tasks which are then send to the task scheduler



Spark's task scheduler launches the tasks by distributing them across worker nodes

Spark Core

- Once data is loaded into an RDD, two basic types of operations can be performed:
  - Transformation which creates a new RDD through changing the original one
  - Actions which measure but do not change the original data
- Transformations are lazily evaluated
  - executed when a subsequent action has a need for the result
  - So can mean errors do not immediately appear
    - E.g., file does not exist
- RDDs will also be kept as long as possible in memory
- A chain of RDD operations gets compiled by Spark into a directed acyclic graph but which is then spread out and calculated over the cluster

Spark SQL

- Spark SQL runs on top of Spark Core and introduces another data abstraction called DataFrames
- DataFrames can be created from RDDs by specifying a schema on how to structure the data elements in the RDD, or can be loaded in directly from various sorts of file formats
- Even although DataFrames continue to use RDDs behind the scenes, they represent themselves to the end user as a collection of data organized into named columns
- You can run Spark directly using:
  - pyspark – uses Python
  - spark-shell – uses Scala
  - spark-sql – to run SQL queries
  - spark-submit – to run a program file, such as Python
- Or can access it via a programming language, such as Java
- See here for information:
  - https://spark.apache.org/docs/latest/sql-programming-guide.html

Spark Core – Map Reduce

- Spark's RDD API is relatively easy to work with compared to writing MapReduce programs

Load in an RDD from a text file, the RDD will represent a
collection of text strings
#(one for each line)

```
text_file = sc.textFile("testfile.txt")
```

Count the word occurrences

```
counts = text_file.flatMap(lambda line: line.split(" ")) \
.map(lambda word: (word, 1)) \
.reduceByKey(lambda a, b: a + b)
```

counts is a PythonRDD, Need loop to print items:

```
for x in counts.collect():
    print x
```

```
(u'A', 1)
(u'ago', 1)
(u'episode', 1)
(u'far', 2)
(u'away', 1)
(u'long', 1)
(u'a', 1)
(u'Another', 1)
(u'Star', 1)
(u'galaxy', 1)
(u'of', 1)
(u'in', 1)
(u'Wars', 1)
(u'time', 1)
```

Spark SQL – Handling JSON (pyspark)

Create a DataFrame object by reading in a file

```
df = spark.read.json("student.json")
df.show()
```

```
+----+----------------+--------------------+------+
| age|          course|               email|  name|
+----+----------------+--------------------+------+
|null|BSc Horticulture|                null|   Tom|
|  45| MSc Agriculture|                null| Helen|
|  30|            null|S.Carter.borchest...| Alice|
|  21|BSc Horticulture|                null|Johnny|
+----+----------------+--------------------+------+
```

student.json:

```
{"name":"Tom",
    "course": "BSc Horticulture"}
{"name":"Helen", "age":45,
    "course": "MSc Agriculture"}
{"name":"Alice", "age":30,
"email":"S.Carter@borchester.ac.uk"}
{"name":"Johnny", "age":21,
    "course": "BSc Horticulture"}
```

DataFrames are structured in columns and rows:

```
df.printSchema()
```

```
root
 |-- age: long (nullable = true)
 |-- course: string (nullable = true)
 |-- email: string (nullable = true)
 |-- name: string (nullable = true)
```

Spark SQL (pyspark)

df.select("name").show()

```
+-------+
|   name|
+-------+
|    Tom|
|  Helen|
|  Alice|
|Johnny|
+-------+
```

SQL-like operations can now easily be expressed:

df.select(df['name'], df['age'] + 1).show()

```
+------+---------+
|  name|(age + 1)|
+------+---------+
|   Tom|     null|
| Helen|       46|
| Alice|       31|
|Johnny|       22|
+------+---------+
```

Spark SQL (pyspark)

df.filter(df['age'] > 21).show()

```
--+---------------+--------------------+-----+
|age|          course|                email| name|
+---+---------------+--------------------+-----+
| 45|MSc Agriculture|                 null|Helen|
| 30|           null|S.Carter.borchest...|Alice|
+---+---------------+--------------------+-----+
```

df.groupBy("course").count().show()

```
+----------------+-----+
|          course|count|
+----------------+-----+
| MSc Agriculture|    1|
|            null|    1|
|BSc Horticulture|    2|
+----------------+-----+
```

Spark SQL (pyspark)

- Spark implements a full SQL query engine which can convert SQL statements to a series of RDD transformations and actions

- First Register the DataFrame as a SQL temporary view:

```
df.createOrReplaceTempView("student")
```

Can then use SQL like syntax:

```
sqlDF = spark.sql("SELECT name, age, course FROM student WHERE
age > 21")

sqlDF.show()

+-----+---+---------------+
| name|age|         course|
+-----+---+---------------+
|Helen| 45|MSc Agriculture|
|Alice| 30|           null|
+-----+---+---------------+
```

You can not just type the SQL Code at the command line

See Workbook for further examples

MLlib, Spark Streaming and GraphX

- There are lots of other Spark tools to help work with Big Data:
  - MLlib is Spark's machine learning library
    - offers classification, regression, clustering, and recommender system algorithms
  - Spark Streaming uses Spark Core and its scheduling engine to perform streaming analytics
    - provides a high-level concept called DStream, which represents a continuous stream of data
  - GraphX is Spark's component implementing programming abstractions to deal with graph based structures
    - based on the RDD abstraction

MLlib, Spark Streaming and GraphX

- Example: Word counting

```
from pyspark import SparkContext
from pyspark.streaming import StreamingContext
sc = SparkContext("local[2]", "StreamingWordCount")
ssc = StreamingContext(sc, 1)

# Create a DStream that will connect to server.mycorp.com:9999 as a source
lines = ssc.socketTextStream("server.mycorp.com ", 9999)

# Split each line into words
words = lines.flatMap(lambda line: line.split(" "))

# Count each word in each batch
pairs = words.map(lambda word: (word, 1))
wordCounts = pairs.reduceByKey(lambda x, y: x + y)

# Print out first ten elements of each RDD generated in the wordCounts Dstream
wordCounts.pprint()

# Start the computation
ssc.start()
ssc.awaitTermination()
```

Conclusion

- This lecture has looked at:
  - SQL on Hadoop
  - Apache Spark

- This week's workbook will look at using:
  - Further Hadoop examples
    - How to use CSV files
  - Apache Spark
    - SQL queries
    - JSON and CSV data