An Overview of Apache Spark

CIS 612
Sunnie Chung
MapReduce Processing Model

- MapReduce, the parallel data processing paradigm, greatly simplified the analysis of big data using large clusters of commodity hardware.
  - Define mappers
  - Shuffling is automatic
  - Define reducers
- For complex work, **chain of MR jobs** together
  - Or use a higher level language or DSL that does this for you
Problem of MapReduce Jobs

- Data Streams is the large amount of data created and arrives in high speed continuously.
  Example: credit card transactions, Internet traffic data, sensor data, network alarm data
- For iterative Map Reduce jobs, multiple map-reduce operations needs to be performed sequentially which involves a very high disk I/O and high latency making them too slow.
- Similarly, for interactive queries, data is read from the disk each time the query is executed.

![Diagram showing iterative Map Reduce operations and interactive queries]
Solution: Apache Spark

- Originally developed in 2009 in UC Berkeley’s AMP Lab

- Fully open sourced in 2010 – now a Top Level Project at the Apache Software Foundation

spark.apache.org
github.com/apache/spark
user@spark.apache.org
Easy and Fast Big Data

• Easy to Develop
  – Rich APIs in Java, Scala, Python
  – Interactive shell

  2-5× less code

• Fast to Run
  – General execution graphs
  – In-memory storage

  Up to 10× faster on disk,
  100× in memory
Spark

- Spark and its streaming version built on top of Hadoop and perform data analysis on clusters
- Improves over MapReduce
  - In memory computing primitives
  - General computation graphs
- Improves usability over MapReduce
  - Rich APIs in Scala, Java, Python
  - Interactive extended Scala Shell

Up to 100x Faster (2 -10x on disk)

Super fast interactive analysis of Big Data
Why Spark Works Faster?

• Caches the data to be shared in-memory which allows faster access (Instead of sharing data across different stages of the job by writing it to the disk which involves disk I/O and replication across nodes in MR Jobs for iterative machine learning algorithms and interactive queries)
Unified Platform

Spark SQL (SQL Query)
Spark Streaming (Streaming)
MLlib (Machine learning)
GraphX (Graph computation)

Spark (General execution engine)

YARN / Mesos / Standalone (resource management)

Data Sources
Languages

Amazon S3

Scala
python
Java

© 2014 MapR Technologies
Unified Platform

- Generalize MapReduce
- Provides a relatively small API
- Handle both Batch and Streaming Use cases
- Also Stand alone Clustering Capability
- Rich APIs to Allow to Write for Spark in Scala, Python and Java

Spark SQL (SQL Query)
Spark Streaming (Streaming)
MLlib (Machine learning)
GraphX (Graph computation)

Spark (General execution engine)

YARN / Mesos / Standalone (resource management)
Spark Execution Engine

- Use In Memory and Pipelining to Achieve Performance over MapReduce
- MapReduce inherent
- In Memory RDD (Resilient Distributed Dataset)
In Memory data storage and near real-time data processing with APIs:

- **Spark SQL**: Used for querying over structured data. It allows the users to ETL their data from its current format in JSON, Parquet, a Database, transform it, and expose it for ad-hoc querying.
- **Spark Streaming**: Supports analytical and interactive applications built on live streaming data (More later)

**Ecosystem on Spark Execution Engine**

- **Spark SQL**: (SQL Query)
- **Spark Streaming**: (Streaming)
- **MLlib**: (Machine learning)
- **GraphX**: (Graph computation)
- **Spark**: (General execution engine)
- **YARN / Mesos / Standalone**: (resource management)
Ecosystem on Spark Execution Engine

Spark APIs (Continued):

- **MLLib**: Machine learning library built on the top of Spark and supports many complex machine learning algorithms which runs 100x faster than map-reduce.
- **GraphX**: Graph computation engine which supports complex graph processing algorithms efficiently and with improved performance. Example: Page Rank.

---

- Spark SQL (SQL Query)
- Spark Streaming (Streaming)
- MLlib (Machine learning)
- GraphX (Graph computation)

- **Spark** (General execution engine)

- YARN / Mesos / Standalone (resource management)
Spark SQL

- SQL query over your data
- It allows the users to ETL their data from its current format (in JSON, Parquet, a Database), transform it to Structured, and Querying over structured data for ad-hoc querying
- Automatic schema discovery for JSON
- RDDs (Resilient Distributed Dataset) from SQL queries
- If you have Hive, you can try out Spark SQL very easily
Spark Streaming

- Dstream: Micro-batching of Discretized data stream - a potentially unbounded continuous Dstream
- Batch operations applied to a stream of data
- Streaming makes available additional APIs, but APIs remain very consistent
- Supports analytical and interactive applications built on live streaming data
Spark Streaming

- Memory Abstraction is by RDD (Resilient Distributed Dataset)
- RDD: Efficiently share data across the different stages of a map-reduce job or provide in-memory data sharing.
- Data is divided into Discretized Streams (Dstream): Continuous mini batching
- Spark Application: Driver program that runs the user’s main function and executes various parallel operations on the clusters.
- Caches the data
MLlib: Machine Learning
Machine learning library built on the top of Spark
Both for batch and iterative use cases
Supports many complex machine learning algorithms which runs 100x faster than map-reduce
• K-Means
• L₁ and L₂-regularized Linear Regression
• L₁ and L₂-regularized Logistic Regression
• Alternating Least Squares
• Naive Bayes
• Stochastic Gradient Descent

Spark SQL (SQL Query)
Spark Streaming (Streaming)
MLlib (Machine learning)
GraphX (Graph computation)

Spark (General execution engine)

YARN / Mesos / Standalone (resource management)
GraphX : Graph Computation on Spark

- Graph computation engine which supports complex graph processing algorithms efficiently and with improved performance.
- Solve graph problems on Spark
  - Page Rank
  - Social Network Graph Processing

Spark SQL (SQL Query)  Spark Streaming (Streaming)  MLlib (Machine learning)  GraphX (Graph computation)

Spark (General execution engine)

YARN / Mesos / Standalone (resource management)
Spark Programming Model

• **Resilient Distributed Datasets (RDDs)**
  Distributed collection of objects
  Manipulated through parallel *transformations* (map, filter, reduce, etc.)
  Can be *cached* in memory across cluster
  Automatically rebuilt on failure

• **Programming Interface**
  – Functional APIs in Scala, Java, Python
  – Interactive use from Scala & Python shell
Spark RDD

- RDD is a collection of Java or Python objects partitioned across a cluster.
  
  ```scala
  val ssc = new StreamingContext(sparkUrl, "Tutorial", Seconds(1), sparkHome, Seq(jarFile))
  val tweets = TwitterUtils.createStream(ssc, None)
  Object tweets is DStream: a Continuous stream of RDDs containing objects of type twitter4j.Status
  ```

- RDDs can be manipulated through operations like map, filter, and reduce, which take functions in the programming language and ship them to nodes on the cluster.

```scala
lines = spark.textFile("hdfs://...")
errors = lines.filter(s => s.contains("ERROR"))
println(errors.count())
```

- This code creates an RDD of strings called lines by reading an HDFS file, then transforms it using filter to obtain another RDD, errors. It then performs a count on this data.

- Fault tolerant – Rerunning the filter operation to rebuild missing partitions
Fast to Run - RDDs

• "Fault-tolerant collection of elements that can be operated on in parallel"
  – Parallelized Collection: Scala collection, parallelized
  – Data pulled from some parallelization-aware data source, such as HDFS

• Transformations
  – Creation of a new dataset from an existing
    • map, filter, distinct, union, sample, groupByKey, join, etc…

• Actions
  – Return a value after running a computation
    • collect, count, first, takeSample, foreach, etc…
More on RDDs (Resilient Distributed Data sets)

- RDDs (Resilient Distributed in memory Data sets) is a fundamental component of Spark.
  - You can create RDDs in two ways:
    - by parallelizing an existing data set
      For example: generating an array then telling Spark to distribute it to workers
    - by obtaining data from some external storage system (eg: HDFS).
  - Once created, you can transform the data. However, when you make transformations, you are not actually modifying the data in place, because RDDs are immutable - transformations always create new RDDs.
  - Transformations are your instructions on how to modify an RDD (in Scala, Python or Java).
  - Spark takes your Transformations, and creates a graph of operations to carry out against the data. Nothing actually happens with your data until you perform an action, which forces Spark to evaluate and execute the graph in order to present you some result.
  - Like in MapReduce DSLs, this allows for a “compile” stage of execution that enables optimization - pipelining, for example. Thus, even though your Spark program may spawn many stages, Spark can avoid the intermediate I/O that is one of the reasons MapReduce is considered “slow”.
  - Check the documentation for a complete list
Fast to Run - RDD Persistence, Caching

- Variety of storage levels
  - memory_only (default), memory_and_disk, etc...

- API Calls
  - Persistent Storage Level
  - cache() : shorthand for persist (StorageLevel.MEMORY_ONLY)

- Considerations
  - Read from disk vs. recompute (memory_and_disk)
  - Total memory storage size (memory_only_ser)
  - Replicate to second node for faster fault recovery (memory_only_2)
    - Think about this option if supporting a web application

http://spark.apache.org/docs/latest/scala-programming-guide.html#rdd-persistence
RDD Fault Recovery

- **Lineage information**: maintains RDDs transformation sequences in a Graph that can be used to efficiently recompute lost data

```python
msgs = textFile.filter(lambda s: s.startswith("ERROR"))
  .map(lambda s: s.split("\t")[2])
```
Spark Streaming - DStreams

- An infinite stream of data, broken up into finite RDDs

- Because Streaming uses the same abstraction (RDDs), it is possible to re-use code.

- Not ideal for the lowest latency applications. You need to be ok with >1s.

- It is ideal for ML, and problems that benefit from iteration.
StreamDM

- Open Source ML Data Mining Library on top of Spark Streaming on Hadoop based open-source ecosystem.
- Input/Output Streams: Dstream - Spark streaming: Datasets are divided in several discretized streams
- It allows combining batch processing algorithms with streaming algorithms

<table>
<thead>
<tr>
<th>trait</th>
<th>objective</th>
</tr>
</thead>
<tbody>
<tr>
<td>StreamReader</td>
<td>read and parse Example and create a stream</td>
</tr>
<tr>
<td>Learner</td>
<td>provides the train method from an input stream</td>
</tr>
<tr>
<td>Model</td>
<td>data structure and set of methods used for Learner</td>
</tr>
<tr>
<td>Evaluator</td>
<td>evaluation of predictions</td>
</tr>
<tr>
<td>StreamWriter</td>
<td>output of streams</td>
</tr>
</tbody>
</table>
StreamDM Task

- Internal instance data structure
- Read by Dstream, Reader class parsed it.
- Data mining algorithm implementation in Learner
- The assignments or predictions from Learner are evaluated by an Evaluator
- Finally, the results are output by a Writer class to disk, console or HDFS
StreamDM Task

- An Instance can contain data structure depending of the input format of the streams (e.g., dense instances in CSV text format, sparse instances in LibSVM format, and text instances)
- All operations are made on the Example; this allows for task design without the need to know the underlying implementation of the instance
- It also allows streams which may change their underlying format at any time.
- .init for initializing the Model inside the learner, and .train for updating the model with the data from the stream.
Application Model Example with Twitter

Twitter

Cloud OR On-Premise

Data Store

Spark Cluster (Filtering and Classification)

OAuth

Tweet Stream

Query

Result

Client
Relational Data Processing in Spark
Spark SQL

• DataFrame API:
  Similar to the widely used data frame concept in R
• Catalyst - To support a wide variety of data sources and analytics workloads in Spark SQL
Goals for Spark SQL

• To extend relational processing to cover native RDDs in Spark and a much wider range of data sources. Following goals for Spark SQL:
  1. Support relational processing both within Spark programs (on native RDDs) and on external data sources using a programmer-friendly API.
  2. Provide high performance using established DBMS techniques.
  3. Easily support new data sources, including semi-structured data and external databases amenable to query federation.
  4. Enable extension with advanced analytics algorithms such as graph processing and machine learning.
Programming interface: DataFrame:

- A DataFrame is equivalent to a table in a relational database.
- DataFrames can be constructed from tables in a system catalog or from existing RDDs of native Java/Python objects.
- They can be manipulated with various relational operators, such as Where and GroupBy, which take expressions in a domain-specific language (DSL) similar to data frames in R and Python.
- Spark DataFrames are lazy, in that each DataFrame object represents a logical plan to compute a dataset, but no execution occurs until the user calls a special “output operation” such as save.
Example

To illustrate, the Scala code below defines a DataFrame from a table in Hive, derives another based on it, and prints a result:

ctx = new HiveContext()
users = ctx.table("users")
young = users.where(users("age") < 21)
println(young.count())

• Users and young are DataFrames
• users("age") < 21 is an *expression* in the data frame DSL
• Finally, each DataFrame simply represents a logical plan (*i.e.*, read the users table and filter for age < 21).
• count, which is an output operation, Spark SQL builds a physical plan to compute the final result.
Querying Native Datasets

- Spark SQL allows users to construct DataFrames directly against RDDs of objects native to the programming language.
- Scala code below defines a DataFrame from an RDD of user objects.
- Spark SQL automatically detects the names ("name" and "age") and data types (string and int) of the columns.
- Traditional object-relational mapping (ORM), often incur expensive conversions that translate an entire object into a different format.
- In contrast, Spark SQL accesses the native objects in-place, extracting only the fields used in each query.
- Join the users RDD with a table in Hive:

```scala
case class User(name: String, age: Int)

// Create an RDD of User objects
usersRDD = spark.parallelize(
    List(User("Alice", 22), User("Bob", 19))
)

// View the RDD as a DataFrame
usersDF = usersRDD.toDF

views = ctx.table("pageviews")
usersDF.join(views, usersDF("name") === views("user"))
```
Advanced Analytics Features

Integration with Spark’s Machine Learning Library

- **Pipeline** is a graph of transformations on data, such as feature extraction, normalization, reduction, and model training, which exchange *datasets*.
- MLlib’s created a compact format, allowing multiple types of fields to be stored for each record.

```python
data = <DataFrame of (text, label) records>

tokenizer = Tokenizer()
    .setInputCol("text").setOutputCol("words")
tf = HashingTF()
    .setInputCol("words").setOutputCol("features")
lr = LogisticRegression()
    .setInputCol("features")

pipeline = Pipeline().setStages([tokenizer, tf, lr])
model = pipeline.fit(data)
```
References

Language R

- Data Mining Language and runtime
- Open source
- Highly dynamic
- Interactive environment
- Rich ecosystem of packages
- Powerful visualization infrastructure
- Concept of Data Frame makes data manipulation convenient
Performance Limitations of R

R language
• R’s dynamic design imposes restrictions on optimization

R runtime
• Single threaded
• Everything has to fit in memory
What would be ideal?

Seamless manipulation and analysis of very large data in R

- R’s flexible syntax
- R’s rich package ecosystem
- R’s interactive environment
- Scalability (scale up and out)
- Integration with distributed data sources / storage
Performance Limitations of R

R language
• R’s dynamic design imposes restrictions on optimization

R runtime
• Single threaded
• Everything has to fit in memory
What would be ideal?

Seamless manipulation and analysis of very large data in R

• R’s flexible syntax
• R’s rich package ecosystem
• R’s interactive environment
• Scalability (scale up and out)
• Integration with distributed data sources / storage
Augmenting R with other frameworks

- In practice data scientists use R in conjunction with other frameworks (Hadoop MR, Hive, Pig, Relational Databases, etc)

1. Load, clean, transform, aggregate, sample
2. Save to local storage
3. Read and analyze in R

iterate
SparkR?

An R package distributed with Apache Spark:
- Provides R frontend to Spark
- Exposes Spark Dataframes (inspired by R and Pandas)
- Convenient interoperability between R and Spark DataFrames
How does SparkR solve our problems?

• No local storage involved
• Write every thing in R
• Use Spark’s distributed cache for interactive/iterative analysis at speed of thought
Example SparkR program

# Loading distributed data
df <- read.df("hdfs://bigdata/logs", source = "json")

# Distributed filtering and aggregation
errors <- subset(df, df$type == "error")
counts <- agg(groupBy(errors, df$code), num = count(df$code))

# Collecting and plotting small data
qplot(code, num, data = collect(counts), geom = "bar", stat = "identity") + coord_flip()
SparkR architecture
Easy to Develop - The REPL

- Iterative Development
  - Cache those RDDs
  - Open the shell and ask questions
  - Compile / save your code for scheduled jobs later

- spark-shell
- pyspark

```
$ bin/spark-shell
Spark assembly has been built with Hive, including Datanucleus jars on classpath
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Welcome to
  __              __
 / __/__  ___ _____/ /__
/_ \/ _ \ / _ `/_  `'/
/___/ .__/\_,_/_/ /_/
 version 1.1.0

Using Scala version 2.10.4 (Java HotSpot(TM) 64-Bit Server VM, Java 1.7.0_60)
Type in expressions to have them evaluated.
Type :help for more information.
Spark context available as sc.

scala>
```
Easy to Develop - A small API

• The canonical example - wordcount
  – Short! Even in Java!

```java
SparkContext sc = new SparkContext(master, appName, [sparkHome], [jars]);
JavaRDD<String> file = sc.textFile("hdfs://...");
JavaRDD<String> counts = file.flatMap(line -> Arrays.asList(line.split(" "))).mapToPair(w -> new Tuple2<String, Integer>(w, 1)).reduceByKey((x, y) -> x + y);
counts.saveAsTextFile("hdfs://...");
```
Directed Acyclic Graph (DAG)

- Directed
  - Only in a single direction
- Acyclic
  - No looping
- Why does this matter?
  - This supports fault-tolerance
Comparing Spark Streaming and Storm

- **Spark Streaming**
  - Less mature
  - DStreams of RDDs
  - >1s latency
  - Micro-batched
  - Stateful, Exactly-once
    - Spark handles failure automatically
  - Scala, Java (no python)

- **Storm**
  - More mature
  - Tuples
  - Lowest latency
  - Event-based (tuple at a time)
  - Stateless (w/o Trident)
    - W/o trident, Storm does not handle failure automatically
  - Java, Clojure, Python, Ruby

Spark Streaming and Storm Trident are more directly comparable
How to Choose one over the other?

- **Spark Streaming**
  - Buzzword compliant!
  - High-level, clean APIs
  - Code reuse across batch and streaming use cases

- **Storm/Trident**
  - Production stable
  - More language options
  - Supports at least once, exactly one and at most once reliability

Production in 3 weeks? Storm. Production in 3-6 months? Spark?
Why Are People Loving on Spark?

• The REPL makes for really easy data exploration.

• Batch and streaming, using the same API.

• Spark APIs are not difficult to pick up

• It’s possible to get a lot done in a lot less code than Mapreduce

• Spark leverages existing technologies for data ingest and resource management, making it easy to adopt alongside Hadoop.
What are the Issues?

• Scaling. Anecdotally, small clusters with larger machines do better than large clusters with smaller machines. (though Yahoo has 1200 nodes)

• Failure modes exist where data can be lost. Use a highly reliable storage backend so you can replay things. Kafka is a good idea. This is documented.

• Lots of this comes down to Spark’s relative immaturity, and that will change.
Some Examples
Word Count

- Java MapReduce (~15 lines of code)
- Java Spark (~7 lines of code)
- Scala and Python (4 lines of code)
  - interactive shell: skip line 1 and replace the last line with `counts.collect()`
- Java8 (4 lines of code)

```scala
val sc = new SparkContext(master, appName, [sparkHome], [jars])
val file = sc.textFile("hdfs://...")
val counts = file.flatMap(line => line.split(" "))
  .map(word => (word, 1))
  .reduceByKey(_ + _)
counts.saveAsTextFile("hdfs://...")
```
Spark Streaming - Network Word Count

http://www.michael-noll.com/blog/2014/10/01/kafka-spark-streaming-integration-example-tutorial/

```scala
// Create the context with a 1 second batch size
val ssc = new StreamingContext(args(0), "NetworkWordCount", Seconds(1),
System.getenv("SPARK_HOME"), StreamingContext.jarOfClass(this.getClass))

// Create a NetworkInputDStream on target host:port and count the
// words in input stream of \n delimited text (eg. generated by 'nc')
val lines = ssc.socketTextStream("localhost", 9999, StorageLevel.MEMORY_ONLY_SER)
val words = lines.flatMap(_.split(" ")).
val wordCounts = words.map(x => (x, 1)).reduceByKey(_ + _)
wordCounts.print()
ssc.start()
```
The Spark Community