CS562: Advanced Topics in Databases
Agenda

Introduction to Spark
Anatomy of a Spark Application
Spark Components
RDD Fundamentals
Shuffle
Performance Tips
Libraries

Focused on spark v.1.6
History of Spark

Spark is an open source cluster computing framework

Initially started by Matei Zaharia in 2009 at UC Berkeley’s ampalab

- Open-sourced in 2010
- Apache Software Foundation 2013
- Supported by Databricks

One of the most active apache projects

- More than 1000 contributors
- Mostly written in Scala
- Used by major companies
Spark Goals

Generality
- Diverse workloads, operators, job sizes

Low Latency
- Sub-second

Fault Tolerance
- Faults are the norm not the exception

Simplicity
- Often comes from generality
Motivation for spark

Software Engineering
- Hadoop code base is huge
- Contributions/Extensions are difficult
- Java only

System/Framework
- Unified pipeline
- Simplified data flow
- Faster processing speed

Data abstraction
- New fundamental data abstraction that is
- ... easy to extend with new operators
- ... allows for a more descriptive computing model
Hadoop – No unified vision

<table>
<thead>
<tr>
<th>General Batching</th>
<th>Specialized systems</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Streaming</td>
</tr>
<tr>
<td>MapReduce</td>
<td>Storm</td>
</tr>
<tr>
<td></td>
<td>S4</td>
</tr>
<tr>
<td></td>
<td>Samza</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Sparse modules
Diversity of APIs
Higher operational costs
Spark – A unified pipeline

Spark Streaming (stream processing)
GraphX (graph processing)
MLlib (machine learning library)
Spark SQL (SQL on Spark)
Spark basic features

A fast and general engine for large-scale data processing

An open-source implementation of Resilient Distributed Datasets (RDD)

Has an advanced DAG execution engine and in-memory computing

Uses the scala collections functional API for manipulating data at scale
Spark basic features

Provides **in-memory data caching and reuse across computations**

Applies a set of coarse-grained **transformations over partitioned data**

**Failure recovery relies on lineage** to re-compute failed tasks

**Supports the majority of input formats** and integrates with Mesos/Yarn
Spark basic features

Fast

- Run machine learning iterative programs
  ...up to 100x faster than Hadoop in memory
  ...or 10x faster on disk
- Avoid materializing data on HDFS after each iteration

Easy to use

- Fluent Scala/Java/Python/R API
- Interactive shell (repl)
- 2-5x less code (than Hadoop MapReduce)
A Simplified Data Flow

Diagram from Introduction to Spark Internals
Hadoop: Bloated Computing Model

```java
public class WordCount {
    public static class TokenizerMapper
        extends Mapper<Object, Text, Text, IntWritable> {
        private final static IntWritable one = new IntWritable(1);
        private Text word = new Text();

        public void map(Object key, Text value, Context context
            ) throws IOException, InterruptedException {
            StringTokenizer itr = new StringTokenizer(value.toString());
            while (itr.hasMoreTokens()) {
                word.set(itr.nextToken());
                context.write(word, one);
            }
        }
    }

    public static class IntSumReducer
        extends Reducer<Text,IntWritable,Text,IntWritable> {
        private IntWritable result = new IntWritable();

        public void reduce(Text key, Iterable<IntWritable> values,
            Context context
            ) throws IOException, InterruptedException {
            int sum = 0;
            for (IntWritable val : values) {
                sum += val.get();
            }
            result.set(sum);
            context.write(key, result);
        }
    }
}
```

```java
public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    Job job = Job.getInstance(conf, "word count");
    job.setJarByClass(WordCount.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
}
```
Spark: Descriptive Computing Model

Organize computation into multiple stages in a processing pipeline
- Transformations apply user code to distributed data in parallel
- Actions assemble final output of an algorithm from distributed data

```scala
// Load the file
val file = sc.textFile("hdfs:// ... ")

// Here is the computation!
val counts = file.flatMap(line ⇒ line.split(" "))
  .map(word ⇒ (word, 1))
  .reduceByKey(_ + _)

// Save file
counts.saveAsTextFile("hdfs:// ... ")
```

![Diagram showing data flow through transformations and actions](https://example.com/diagram.png)
Spark Applications: The Big Picture

There are two ways to manipulate data in Spark

- Use the interactive shell (i.e. repl)
- Standalone applications (i.e. driver programs)
Spark Applications: The Big Picture

**Spark Driver**
- Separate process to execute user application
- Creates SparkContext to schedule jobs execution and negotiate with cluster manager

**Executors**
- Run tasks scheduled by driver
- Store computation results in memory, on disk or off-heap
- Interact with storage systems

**Cluster Manager**
- Mesos
- Yarn
- Spark standalone
Spark Deployment Modes

One SparkContext per JVM

Worker nodes are machines that run executors
- Host one or multiple Workers
- One JVM (1 process) per Worker
- Each worker can spawn one or more Executors

Executors run tasks
- Run in child JVM
- Execute one or more tasks using threads in a ThreadPool
Comparison to Hadoop

Hadoop MapReduce
- One process per task
- MultithreadedMapper - advanced feature to have threads in Map Tasks
- Short-lived Executor with one large task

Spark
- Tasks run in one or more threads, within a single process
- Executor process statically allocated worker, even with no threads
- Long-lived Executor with many small tasks
Benefits of Spark Architecture

**Isolation**
- Applications are completely isolated
- Task scheduling per application

**Low overhead**
- Task setup cost is that of spawning a thread not a process (10-100 times faster)
- Small tasks → mitigate effects of data skew

**Sharing data**
- Applications cannot share data in memory natively
- Use an external storage service like Tachyon

**Resource allocation**
- Static process provisioning for executors even without active tasks
- Dynamic provision also available
Units of Physical Execution

Jobs
- Individual executed action — top level work item
- Composed by a set of tasks arranged in stages

Stages
- A wave of work within a job corresponding to one or more pipelined RDDs
- Job split based on previous cached action or shuffle

Tasks
- A unit of work within a stage corresponding to one RDD partition — The minimum unit of physical execution

Shuffle
- Redistribution of data across nodes - The transfer of data between stages
Data locality

Data locality principle
- Same as Hadoop MapReduce
- Avoid network I/O, workers should manage local data

Data locality and caching
- When loading data from HDFS so use HDFS locality prefs (blocks)
- If RDD is in cache use its locations
- If something fails out of cache go back to HDFS
Spark Components

The Task Scheduler
- Responsible for sending tasks to the cluster, running them, retrying if there are failures and mitigating strugglers
- Reports to the DAG Scheduler

The Scheduler Backend
- Backend interface for scheduling systems that allows plugging in different implementations (Mesos, Yarn, etc.)

BlockManager
- Provides interfaces for putting and retrieving blocks both locally and remotely into various stores (memory, disk, off-heap)
Execution workflow

rrd1.join(rrd2)
.groupBy(...) .filter(...) Build the operator DAG

split graph into stages of tasks submit each stage as ready

launches tasks via cluster manager retries failed or struggling tasks executes tasks store and serve blocks

Agnostic to operators Does not know about stages

RDDs

RDD - Resilient Distributed Dataset

Properties
- Immutable
- Distributed
- Lazily evaluated
- Serializable
- Type safe
- Cacheable
- Fault-tolerant
RDD: Resilient Distributed Dataset

A data structure that
- either points to a direct data source (e.g. HDFS)
- apply some transformations to its parent RDD(s) to generate new data elements

Computations on RDDs
- are represented by lazily evaluated lineage DAGs composed by chain RDDs

Provide an API for
- Manipulating the collection elements (transformations and materialization)
- Persisting intermediate results in memory for later use
- Controlling partitioning to optimize data placement
RDD: Resilient Distributed Dataset

**Can be created**
- From storage (distributed file system, dataset, plain file)
- From another RDD

**Stores information about parent RDDs**
- For execution optimization and operation pipelining
- To re-compute the data in case of failure

**Overall objective**
- Support a wide array of operators (more than map and reduce)
- Allow arbitrary composition of such operators

**Simplify scheduling**
- Avoid to modify the scheduler for each operator
RDD: Partitions

RDDs are automatically distributed across the network by means of partitions
A partition is a logical division of data
RDD data is just a collection of partitions
Spark automatically decides the number of partitions when creating an RDD
All input, intermediate and output data will be presented as partitions
Partitions are basic units of parallelism
A task is launched per each partition
RDD: Partitions

http://datalakes.com/rdds-simplified/
RDD: Partitions Immutability & Distribution

All partitions are immutable

- Every transformation generates a new partition
- Partition immutability driven by underneath storage like HDFS
- Partition immutability allows for fault recovery

Partitions from HDFS are distributed by default

- Partitions are also location aware (data locality)
- For computed data, using caching we can distribute in memory also
RDDs: A developer’s view

Distributed immutable data + lazily evaluated operations
- partitioned data
- transformations & actions

An interface defining 5 main properties
- `getPartitions` - list of partitions (splits)
- `getDependencies` - list of dependencies on other RDDs
- `compute` - function for computing each split
- `getPrefferedLocations` - list of preferred locations to compute each split on
- `partitioner` - partitioner for key-value RDDs
RDDs Example

HadoopRDD
- **getPartitions** — HDFS blocks
- **getDependencies** — None
- **compute** — load block in memory
- **getPreferredLocations** — HDFS block locations
- **partitioner** — None

MapPartitionsRDD
- **getPartitions** — same as parent
- **getDependencies** — parent RDD
- **compute** — compute parent and apply map
- **getPreferredLocations** — same as parent
- **partitioner** — None

Joined RDD

- **getPartitions** — one per reduce task
- **getDependencies** — shuffle on each parent
- **compute** — read and join shuffled data
- **getPrefferedLocations** — none
- **partitioner** - HashPartitioner(numTasks)
RDDs Costs

Cheap
- No serialization
- No IO
- Pipelined

Expensive
- Serialize data
- Write to disk
- Network transfer
- De-serialize data

Be careful though!
It is easy to build an inefficient RDD lineage

Expensive
rdd.groupBy().filter()

Faster
rdd.filter().groupBy()
RDD operations - Transformations

As in relational algebra, the application of a transformation to an RDD yields a new RDD (immutability)

Transformations are lazily evaluated which allow for optimizations to take place before execution

- The lineage keeps track of all transformations that have to be applied when an action happens

```scala
rdd = sc.textFile("spam.txt")
filtered = rdd.filter()
filtered.count()
```

http://datalakes.com/rrds-simplified/
RDD Transformations

map()  groupByKey()
flatMap()  reduceByKey()
filter()  sortByKey()
mapPartitions()  join()
mapPartitionsWithIndex()  cogroup()
sample()  cartesian()
union()  pipe()
intersection()  coalesce()
distinct()  repartion()
cache()  partitionBy()
persist()  ...
### RDD Common Transformations

**Unary**

<table>
<thead>
<tr>
<th>RDD Method</th>
<th>Input RDD</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>rdd. map(x =&gt; x * x)</td>
<td>{1, 2, 3, 3}</td>
<td>{1, 4, 9, 9}</td>
</tr>
<tr>
<td>rdd.flatMap(line =&gt; line.split(&quot; &quot;))</td>
<td>{&quot;hello world&quot;, &quot;hi&quot;}</td>
<td>{&quot;hello&quot;, world&quot;, &quot;hi&quot;}</td>
</tr>
<tr>
<td>rdd.filter(x =&gt; x != 1)</td>
<td>{1, 2, 3, 3}</td>
<td>{2, 3, 3}</td>
</tr>
<tr>
<td>rdd.distinct ()</td>
<td>{1, 2, 3, 3}</td>
<td>{1, 2, 3}</td>
</tr>
</tbody>
</table>

**Binary**

<table>
<thead>
<tr>
<th>RDD Method</th>
<th>RDD</th>
<th>Other</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>rdd.union (other)</td>
<td>{1, 2, 3}</td>
<td>{3,4,5}</td>
<td>{1,2,3,3,4,5}</td>
</tr>
<tr>
<td>rdd.intersection(other)</td>
<td>{1, 2, 3}</td>
<td>{3,4,5}</td>
<td>{3}</td>
</tr>
<tr>
<td>rdd.subtract(other)</td>
<td>{1, 2, 3}</td>
<td>{3,4,5}</td>
<td>{1, 2}</td>
</tr>
<tr>
<td>rdd.cartesian(other)</td>
<td>{1, 2, 3}</td>
<td>{3,4,5}</td>
<td>{(1,3),(1,4), ... (3,5)}</td>
</tr>
</tbody>
</table>
RDD operations - Actions

Apply transformation chains on RDDs, eventually performing some additional operations (e.g. counting)
– i.e. trigger job execution

Used to materialize computation results

Some actions only store data from the RDD upon which the action is applied and convey it to the driver
RDD Actions

- reduce()
- collect()
- count()
- first()
- take()
- takeSample()
- saveToCassandra()
- takeOrdered()
- saveAsTextFile()
- saveAsSequenceFile()
- saveAsObjectFile()
- countByKey()
- forEach()
- ...

...
RDD Actions

- **reduce()**: Takes a function that operates on two elements of the type in your RDD and returns a new element of the same type. The function is applied on all elements.

<table>
<thead>
<tr>
<th>RDD</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>rdd.reduce((x, y) =&gt; x + y)</code></td>
<td><code>{1,2,3}</code></td>
</tr>
</tbody>
</table>

- **fold()**: also takes a function with the same signature as needed for `reduce()`, but in addition takes a “zero value” to be used for the initial call on each partition.

  ```scala
  val rdd = sc.makeRDD(List(('Jack', 1000.0), ('Tom', 800.0), ('Mark', 2200.0)))
  val noneEmployee = ('none',0.0)
  val maxSalaryEmployee = rdd.fold(noneEmployee)((a,b) => {if(a._2 < b._2) b else a})
  println("Employee with max salary is " + maxSalaryEmployee._1 + " with salary " + maxSalaryEmployee._2)
  Output: Employee with max salary is Mark with salary 2200.0
  ```
**RDD Actions**

- **collect()**: returns the entire RDD’s contents (commonly used in unit tests where the entire contents of the RDD are expected to fit in memory). The restriction here is that all of your data must fit on a single machine, as it all needs to be copied to the driver.

- **take()**: returns n elements from the RDD and tries to minimize the number of partitions it accesses. No expected order

- **count()**: returns the number of elements

<table>
<thead>
<tr>
<th>Example</th>
<th>RDD</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>rdd.collect()</td>
<td>{1,2,3}</td>
<td>{1,2,3}</td>
</tr>
<tr>
<td>rdd.take(2)</td>
<td>{1,2,3,4}</td>
<td>{1,3}</td>
</tr>
<tr>
<td>rdd.count()</td>
<td>{1,2,3,3}</td>
<td>4</td>
</tr>
</tbody>
</table>
RDD: Transformation and Actions

RDD (immutable)

transformations
map, filter, ...

pointer to parent

actions

saveAsTextFile, reduce, ...

Save/Display

New RDD

RDD Cache - rdd.cache()

If we need the results of an RDD many times, it is best to cache it

- RDD partitions are loaded into the memory of the nodes that hold it
- avoids re-computation of the entire lineage
- in case of node failure compute the lineage again

http://datalakes.com/rdds-simplified/
Broadcast & Accumulators

Broadcast variables
- Efficiently send a large read only value to all worker nodes
- Uses a bittorrent technique
  Data is split to blocks
  When a leecher fetches a block, then it acts as a source for this block
- e.g. send a large feature vector in a ML algorithm to all nodes or send a read-only dataset

Accumulators
- Aggregate values from worker nodes back to the driver program
- Only the driver can access the value of an accumulator not the tasks
- e.g. count events that occur during job execution for debugging purposes
Broadcast & Accumulators

val broadcast = sc.broadcast(addressesMap)

@driver

http://datalakes.com/rdds-simplified2/
Shuffle

Redistributes data among partitions

Partition keys into buckets (user-defined partitioner)

Optimizations:
- Avoided when possible, if data is already properly partitioned
- Partial aggregation reduces data movement
Shuffle

Spark runs jobs stage by stage

Stages are build up by DAGScheduler according to RDDs

ShuffleDependency

- ShuffleRDD / CoGroupedRDD will have a shuffle dependency

- Many operators create ShuffleRDD / CoGroupedRDD

  repartition, combineByKey, groupBy, cogroup

  many other operators will further call into the above operators (e.g. join operator)

Each ShuffleDependency maps to one stage in Spark Job and then will lead to a shuffle
Shuffle

Diagram from Spark Shuffle Introduction
Why Shuffle is expensive

During shuffle data no longer stays in memory

Shuffling involves

- **Data partition** which might involve very expensive data sorting works
- **Serialization/Deserialization** to transfer data through the network or across processes
- **Data compression** to reduce IO increases CPU usage though
- **Disk IO** multiple times on one single data block
Spark offers a pluggable Shuffle Framework

**ShuffleManager**
- Manages shuffle related components
- Default shuffle is sort (pre 1.2 used hash)

**ShuffleWriter**
- Handle shuffle data output logics

**ShuffleReader**
- Fetch shuffle data to be used by e.g. ShuffleRDD

**ShuffleBlockManager**
- Manage the mapping relation between abstract bucket and materialized data block
Conclusions

- Data flow engines are becoming an important platform for numerical algorithms
- While early models like MapReduce were inefficient, new ones like Spark close this gap
Acknowledgments - Resources

Aaron Davidson — Building a unified data pipeline in apache spark

Madhukara Phatak - Anatomy of RDD

Anton Kirillov - Apache spark in depth — Core concepts, architecture & internals

Patrick Wendell — Tuning and Debugging in Apache Spark
Acknowledgments - Resources

Spark — Devops advanced class

Spark Shuffle Introduction

Pietro Michiardi - Apache Spark Internals

Databricks — Advanced Spark