Lightning-Fast Cluster Computing with Spark and Shark

Mayuresh Kunjir and Harold Lim
Duke University
Outline

• Spark
  – Spark Overview
  – Components
  – Life of a Job
  – Spark Deployment

• Shark
  – Motivation
  – Architecture

• Results and Live Demo
Spark Overview

• Open source cluster computing system that aims to make data analytics *fast*
  – Supports diverse workloads
  – sub-second latency
  – fault tolerance
  – Simplicity

• Research Paper: Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing [Zaharia et al., NSDI 2012]
Small Codebase

- Spark core: 16,000 LOC
  - Operators: 2000
  - Block manager: 2700
  - Accumulators: 200
  - Scheduler: 2500
  - Networking: 1200
  - Broadcast: 3500

- Interpreter: 3300 LOC

- Hadoop I/O: 400 LOC

- Mesos backend: 700 LOC

- Standalone backend: 1700 LOC

- Borrowed from Spark User Meetup 2012, Introduction to Spark Internals
Components

Your program

Spark client (app master)

Spark worker

- RDD graph
- Scheduler
- Block tracker
- Shuffle tracker
- Cluster manager
- Task threads
- Block manager
- HDFS, HBase, ...

Your program

```
sc = new SparkContext
f = sc.textFile("...")
f.filter(...)
.count()
```

- Borrowed from Spark User Meetup 2012, Introduction to Spark Internals
Spark Program

- Can be written using Scala, Java, or Python.
- Spark includes spark-shell to run spark interactively
- There is also a higher-level abstraction called Shark (explained in the 2nd half of talk) that exposes HiveQL language and compiles down to Spark program
- Latest release of Spark can be downloaded from spark-project.org/downloads.
  - Includes examples, e.g., K-means, logistic regression, alternating least squares matrix factorization, etc
RDD

• A Spark program revolves around the concept of resilient distributed datasets (RDD)
  – Fault-tolerant collection of elements that can be operated on in parallel
  – Perform operations on RDD
    • Transformations (e.g., map, flatMap, union, filter, etc) that creates new RDD
    • Actions returns a value to the driver program (e.g., collect, count, etc)
Example Program

- val sc = new SparkContext("spark://...", "MyJob", home, jars)
- val file = sc.textFile("hdfs://...")
- val errors = file.filter(_.contains("ERROR"))
- errors.cache()
- errors.count()

- Borrowed from Spark User Meetup 2012, Introduction to Spark Internals
RDD Graph

Dataset-level view:

- First run: data not in cache, so use HadoopRDD’s locality prefs (from HDFS)
- Second run: FilteredRDD is in cache, so use its locations
- If something falls out of cache, go back to HDFS

file:

HadoopRDD
path = hdfs://...

FilteredRDD
func = _.contains(…)
shouldCache = true

errors:

• Borrowed from Spark User Meetup 2012, Introduction to Spark Internals
### Scheduling Process

**RDD Objects**
- `rdd1.join(rdd2)
groupBy(...)
filter(...)`

**DAGScheduler**
- Build operator DAG
- Split graph into **stages** of tasks
- Submit each stage as ready
- Agnostic to operators!

**TaskScheduler**
- Launch tasks via cluster manager
- Retry failed or straggling tasks
- Doesn’t know about stages

**Worker**
- Execute tasks
- Store and serve blocks

---

- Borrowed from Spark User Meetup 2012, Introduction to Spark Internals
RDD Abstractions

• Extensible (Can implement new RDD operations, e.g., to read from different sources)

• The current implemented RDD operations can support a wide-range of workloads

• The RDD Interface
  – Set of partitions (“splits”)
  – List of dependencies on parent RDDs
  – Function to compute a partition given parents
  – Optional preferred locations
  – Optional partitioning info (Partitioner)
Example: JoinedRDD

- partitions = one per reduce task
- dependencies = “shuffle” on each parent
- compute\((partition)\) = read and join shuffled data
- preferredLocations\((part)\) = none
- partitioner = HashPartitioner(numTasks)

Spark will now know this data is hashed!

Borrowed from Spark User Meetup 2012, Introduction to Spark Internals
Dependency Types

• Unlike Hadoop, supports a wide range of dependency between operations

“Narrow” deps:

- map, filter
- union

“Wide” (shuffle) deps:

- join with inputs co-partitioned
- groupByKey
- join with inputs not co-partitioned
DAG Scheduler Optimizations

- Pipelines narrow ops. within a stage
- Picks join algorithms based on partitioning (minimize shuffles)
- Reuses previously cached data

Borrowed from Spark User Meetup 2012, Introduction to Spark Internals
Task Details

• Each Task object is self-contained
  – Contains all transformation code up to input boundary (e.g. HadoopRDD => filter => map)
• Allows Tasks on cached data even if they fall out of cache

Design goal: any Task can run on any node

Only way a Task can fail is lost map output files

• Borrowed from Spark User Meetup 2012, Introduction to Spark Internals
TaskScheduler Details

• Can run multiple concurrent TaskSets (Stages), but currently does so in FIFO order
  – Would be really easy to plug in other policies!
• Responsible for scheduling and launching tasks on Worker nodes
• We (Duke) have implemented a Fair Scheduler
Worker

- Implemented by the Executor class
- Receives self-contained Task objects and calls run() on them in a thread pool
- Tasks share the same JVM, which allows launching new tasks quickly
- Has a BlockManager for serving shuffle data and cached RDDs (uses the same JVM memory space)
- Cached RDD are configurable
  - can be stored as Java object (no serialization/deserialization overhead) or Serialized objects.
  - Whether to spill to disk or recompute partitions from parent RDDs when data fall out of cache
  - LRU eviction policy
Spark Deployment

• Spark with Mesos (fine-grained)
  – Incubator.apache.org/mesos
  – Mesos offers resources to Spark programs (using some configurable policy)
  – Each spark tasks run as separate Mesos tasks

• Spark with Mesos (Coarse-grained)
  – Only 1 Mesos task is launched on each machine
  – Mesos Tasks are long-running and released after program has completed
  – Spark program bypasses Mesos scheduler and dynamically schedules spark tasks on Mesos tasks (can schedule more spark tasks on a Mesos task)
Spark Deployment

• Spark Stand-alone Mode
  – Similar to Mesos Coarse-grained mode
  – No need to have Mesos running on the cluster

• Spark with YARN (NextGen Hadoop)
  – Requests pre-defined number of resource containers from YARN
  – Holds on to resource containers until the entire Spark program finishes
  – Spark schedules which tasks gets run on the obtained resource containers
Another Example Spark Program

```scala
val sc = new SparkContext(args(0), "SparkLocalKMeans", home, jars)
val lines = sc.textFile(args(1))
val data = lines.map(parseVector._).cache()
val K = args(2).toInt
val convergeDist = args(3).toDouble
var kPoints = data.takeSample(false, K, 42).toArray
var tempDist = 1.0
while(tempDist > convergeDist) {
  var closest = data.map (p => (closestPoint(p, kPoints), (p, 1)))
  var pointStats = closest.reduceByKey{case ((x1, y1), (x2, y2)) =>
                                        (x1 + x2, y1 + y2)}
  var newPoints = pointStats.map {pair =>
                                   (pair._1, pair._2._1 / pair._2._2)}
                              .collectAsMap()
  tempDist = 0.0
  for (i <- 0 until K) {
    tempDist += kPoints(i).squaredDist(newPoints(i))
  }
  for (newP <- newPoints) {
    kPoints(newP._1) = newP._2
  }
  println("Finished iteration (delta = " + tempDist + ")")
}
println("Final centers:")
kPoints.foreach(println)
```
Other Spark Features: Shared Variables

• Normally, Spark operations work on separate copies of all variables
• Spark now has support for limited type of read-write shared variables across tasks:
  – Broadcast variables: Keep a read-only variable cached on each machine (no need to ship a copy of variable with tasks)
    • E.g., Give every node a copy of a large input dataset in efficient manner
    • Spark uses efficient broadcast algorithms
  – Accumulators: variables that are only “added” to through an associative operation.
    • E.g., To implement counters or sums
    • Tasks can add to the accumulator value and the driver program can read the value
Some Issues

• RDDs cannot be shared across different Spark Programs
  – Others have implemented a “server” program/shell that maintains a long-lived SparkContext (Spark Program) and users submits queries to this server
  – Shark has a server mode

• Task operations can be memory-intensive and cause GC problems
  – Unlike Hadoop, task’s input are put into memory (e.g., grouping is done using in-memory hash table)

• Base on experience, GC problems can result in poor performance
  – Have to ensure level of parallelism is high enough
  – Ensure enough memory partition is set for tasks’ working set (spark.storage.memoryFraction)
Outline

• Spark
  – Spark Overview
  – Components
  – Life of a Job
  – Spark Deployment

• Shark
  – Motivation
  – Architecture

• Results and Live Demo
Apache Hive

- Data warehouse over Hadoop developed at Facebook
- SQL-like language, HiveQL interface to query structured data on HDFS
- Queries compile to Hadoop MapReduce jobs
- Very popular: 90+% of Facebook Hadoop jobs generated by Hive
Hive Architecture

Borrowed from AMP Camp One – Big Data Bootcamp Berkeley, August 2012, “Structured Data with Hive and Shark”
Hive Principles

• SQL provides a familiar interface for users
• Extensible types, functions, and storage formats
• Horizontally scalable with high performance on large datasets

Borrowed from AMP Camp One – Big Data Bootcamp Berkeley, August 2012, “Structured Data with Hive and Shark”
Hive Downsides

• Not interactive
  – Hadoop startup latency is ~20 seconds, even for small jobs

• No query locality
  – If queries operate on the same subset of data, they still run from scratch
  – Reading data from disk is often bottleneck

• Requires separate machine learning dataflow
Shark Motivations

• Data warehouses exhibit a huge amount of temporal locality
  – 90% of Facebook queries could be served in RAM

• Can we keep all the benefits of Hive (scalability and extensibility) and exploit the temporal locality?
Hive

Borrowed from AMP Camp One – Big Data Bootcamp Berkeley, August 2012, “Structured Data with Hive and Shark”
Shark

Borrowed from AMP Camp One – Big Data Bootcamp Berkeley, August 2012, “Structured Data with Hive and Shark”
Introducing Shark

- Shark = **Spark** + **Hive**
- Run HiveQL queries through Spark with Hive UDF, UDAF, SerDe
- Utilize Spark’s in-memory RDD caching and flexible language capabilities
- Integrates with Spark for machine learning operations

Borrowed from Spark User Meetup, February 2012, “Shark – Hive on Spark”
Caching Data in Shark

CREATE TABLE mytable_cached AS SELECT * from mytable WHERE count > 10;

- Creates a table cached in a cluster’s memory using RDD.cache()
Example: Log Mining

- Load error messages from a log into memory, then interactively search for various patterns

**Spark:**

```scala
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
messages = errors.map(_.split('t')(1))
messages.cache()
messages.filter(_.contains("foo")).count
messages.filter(_.contains("bar")).count
```

**Shark:**

```sql
CREATE TABLE log(header string, message string) ROW FORMAT DELIMITED FIELDS TERMINATED BY 't' LOCATION "hdfs://...";
CREATE TABLE errors_cached AS SELECT message FROM log WHERE header == "ERROR";
SELECT count(*) FROM errors_cached WHERE message LIKE "%foo%";
SELECT count(*) FROM errors_cached WHERE message LIKE "%bar%";
```

Borrowed from Spark User Meetup, February 2012, “Shark – Hive on Spark”
Data Model

• Tables: unit of data with the same schema
• Partitions: e.g. range-partition tables by date
• Buckets: hash partitions within partitions
  – not yet supported in Shark
Data Types

• Primitive types
  – TINYINT, SMALLINT, INT, BIGINT
  – BOOLEAN
  – FLOAT, DOUBLE
  – STRING

• Complex types
  – Structs: STRUCT {a INT; b INT}
  – Arrays: [‘a’, ‘b’, ‘c’]
  – Maps (key-value pairs): M[‘key’]

Borrowed from AMP Camp One – Big Data Bootcamp Berkeley, August 2012, “Structured Data with Hive and Shark”
HiveQL

• Subset of SQL
  – Projection, Selection
  – Group-by and aggregations
  – Sort by and order by
  – Joins
  – Sub queries, unions

• Hive-specific
  – Supports custom map/reduce scripts (TRANSFORM)
  – Hints for performance optimizations

Borrowed from AMP Camp One – Big Data Bootcamp Berkeley, August 2012, “Structured Data with Hive and Shark”
Performance Optimizations

- Caching data in-memory
- Hash-based shuffles for group-by
- Push-down of limits
- Join optimizations through Partial DAG Execution
- Columnar memory storage
Caching

```
SELECT * FROM pages WHERE body LIKE '%%XYZ%%'
```

- **Hive**: 38 seconds
- **Shark (disk)**: 11 seconds
- **Shark (RAM)**: 2 seconds

Borrowed from AMP Camp One – Big Data Bootcamp Berkeley, August 2012, “Structured Data with Hive and Shark”
Sort, limit, hash shuffle

```sql
SELECT sourceIP, AVG(pageRank), SUM(adRevenue) AS earnings
FROM rankings AS R, userVisits AS V ON R.pageURL = V.destURL
WHERE V.visitDate BETWEEN '1999-01-01' AND '2000-01-01'
GROUP BY V.sourceIP
ORDER BY earnings DESC LIMIT 1
```

Borrowed from AMP Camp One – Big Data Bootcamp Berkeley, August 2012, “Structured Data with Hive and Shark”
Outline

• Spark
  – Spark Overview
  – Components
  – Life of a Job
  – Spark Deployment

• Shark
  – Motivation
  – Architecture

• Results and Live Demo
TPC-H Data

- 5 node cluster running Hive 0.9 and Shark 0.2
- 50GB data on HDFS
- Data read as Hive external tables
Hive versus Shark

<table>
<thead>
<tr>
<th>Query</th>
<th>On Hive</th>
<th>On Shark (disk)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0:06:10</td>
<td>0:02:20</td>
</tr>
<tr>
<td>2</td>
<td>0:10:00</td>
<td>0:07:30</td>
</tr>
<tr>
<td>3</td>
<td>0:14:00</td>
<td>0:05:10</td>
</tr>
<tr>
<td>4</td>
<td>0:11:40</td>
<td>0:04:30</td>
</tr>
<tr>
<td>5</td>
<td>0:17:30</td>
<td>0:07:20</td>
</tr>
<tr>
<td>6</td>
<td>0:03:10</td>
<td>0:01:35</td>
</tr>
<tr>
<td>7</td>
<td>0:29:10</td>
<td>0:17:40</td>
</tr>
<tr>
<td>8</td>
<td>0:19:10</td>
<td>0:09:50</td>
</tr>
<tr>
<td>9</td>
<td>0:48:20</td>
<td>0:19:45</td>
</tr>
<tr>
<td>10</td>
<td>0:15:00</td>
<td>0:03:50</td>
</tr>
<tr>
<td>11</td>
<td>0:07:30</td>
<td>0:02:00</td>
</tr>
<tr>
<td>12</td>
<td>0:10:30</td>
<td>0:06:20</td>
</tr>
<tr>
<td>13</td>
<td>0:10:00</td>
<td>0:04:00</td>
</tr>
<tr>
<td>14</td>
<td>0:05:35</td>
<td>0:01:50</td>
</tr>
<tr>
<td>15</td>
<td>0:07:30</td>
<td>0:01:40</td>
</tr>
<tr>
<td>16</td>
<td>0:12:50</td>
<td>0:04:50</td>
</tr>
<tr>
<td>17</td>
<td>0:20:00</td>
<td>0:10:30</td>
</tr>
<tr>
<td>18</td>
<td>0:30:00</td>
<td>0:17:10</td>
</tr>
<tr>
<td>19</td>
<td>0:11:40</td>
<td>0:07:05</td>
</tr>
<tr>
<td>20</td>
<td>0:15:00</td>
<td>0:04:10</td>
</tr>
<tr>
<td>21</td>
<td>0:36:40</td>
<td>0:19:15</td>
</tr>
<tr>
<td>22</td>
<td>0:10:10</td>
<td>0:03:40</td>
</tr>
</tbody>
</table>

Number of reducers have to be explicitly set in Shark
Performance Tuning

• Two parameters that can significantly affect performance:
  1. Setting the number of reducers
  2. Map-side aggregation

Borrowed from AMP Camp One – Big Data Bootcamp Berkeley, August 2012, “Structured Data with Hive and Shark”
Number of Reducers

- `SET mapred.reduce.tasks = 50;`
- Shark relies on Spark to infer the number of map tasks (automatically based on input size)
- Number of reduce tasks need to be specified by the user
- Out of memory error on slaves if num too small

Borrowed from AMP Camp One – Big Data Bootcamp Berkeley, August 2012, “Structured Data with Hive and Shark”
Map-side Aggregation

- SET hive.map.aggr = TRUE;
- Aggregation functions are algebraic and can be applied on mappers to reduce shuffle data
- Each mapper builds a hash-table to do the first-level aggregation

Borrowed from AMP Camp One – Big Data Bootcamp Berkeley, August 2012, “Structured Data with Hive and Shark”
Possible Improvements

• Caching is currently explicitly set
  – Can this be set automatically?
• Multi-query optimization
  – What to cache?
• Treating workload as a sequence
  – When to cache?
  – When to run a query?
• Notion of Fairness
  – Is the notion of Hadoop fairness still valid, given that Spark can also utilize memory (cached RDD) resources?
• Better support for Multi-tenancy?
  – Spark was originally designed/implemented to have each user workload as separate Spark program
  – However, RDDs can’t be shared across different Spark Programs
  – Current workaround: Have a single Spark program server and implement a fair task scheduler
  – Is this good enough?
Useful Links

• Project home pages
  – http://spark-project.org/
  – http://shark.cs.berkeley.edu/

• Research Papers

• AMP Camp – Big Data Bootcamp
  – http://ampcamp.berkeley.edu/amp-camp-one-berkeley-2012/
Questions?

Thank you!

• mayuresh@cs.duke.edu
• harold@cs.duke.edu