DEVOPS ADVANCED CLASS

March 2015: Spark Summit East 2015

http://spark-summit.org/east/training/devops

www.linkedin.com/in/blueplastic
making big data simple

• Founded in late 2013
• by the creators of Apache Spark
• Original team from UC Berkeley AMPLab
• Raised $47 Million in 2 rounds
• ~50 employees
• We’re hiring!  (http://databricks.workable.com)
• Level 2/3 support partnerships with
  • Cloudera
  • Hortonworks
  • MapR
  • DataStax

Databricks Cloud:
“A unified platform for building Big Data pipelines – from ETL to Exploration and Dashboards, to Advanced Analytics and Data Products.”
The Databricks team contributed more than **75%** of the code added to Spark in the past year.
AGENDA

Before Lunch

• History of Spark
• RDD fundamentals
• Spark Runtime Architecture Integration with Resource Managers (Standalone, YARN)
• GUIs
• Lab: DevOps 101

After Lunch

• Memory and Persistence
• Jobs -> Stages -> Tasks
• Broadcast Variables and Accumulators
• PySpark
• DevOps 102
• Shuffle
• Spark Streaming
Some slides will be skipped

Please keep Q&A low during class

(5pm – 5:30pm for Q&A with instructor)

2 anonymous surveys: Pre and Post class

Lunch: noon – 1pm

2 breaks (before lunch and after lunch)
• AMPLab project was launched in Jan 2011, 6 year planned duration

• Personnel: ~65 students, postdocs, faculty & staff

• Funding from Government/Industry partnership, NSF Award, Darpa, DoE, 20+ companies

• Created BDAS, Mesos, SNAP. Upcoming projects: Succinct & Velox.

“Unknown to most of the world, the University of California, Berkeley’s AMPLab has already left an indelible mark on the world of information technology, and even the web. But we haven’t yet experienced the full impact of the group[...] Not even close”

- Derrick Harris, GigaOm, Aug 2014
Scheduling

Monitoring

Distributing
Distributions:
- CDH
- HDP
- MapR
- DSE
Just realized Berkeley AMPLab is the Xerox PARC of this century. #sparksummit
General Batch Processing

Specialized Systems
(iterative, interactive, ML, streaming, graph, SQL, etc)

General Unified Engine
In a Nutshell, Apache Spark...

... has had 17,297 commits made by 448 contributors representing 332,309 lines of code

... is mostly written in Scala
with a well-commented source code

... has a codebase with a long source history
maintained by a very large development team
with stable Y-O-Y commits

... took an estimated 88 years of effort (COCOMO model)
starting with its first commit in March, 2009 Aug 2009
ending with its most recent commit 2 days ago

Languages

- Scala: 76%
- Python: 9%
- Java: 7%
- Other: 8%

Source: openhub.net
CPUs:
- 10 GB/s
- 100 MB/s
- 0.1 ms random access
- $0.45 per GB

1 Gb/s or 125 MB/s
- 600 MB/s
- 3-12 ms random access
- $0.05 per GB

Network:
- 1 Gb/s or 125 MB/s
- 0.1 Gb/s

Nodes in another rack:
- 1 Gb/s or 125 MB/s

Nodes in same rack:
- 1 Gb/s or 125 MB/s
The main abstraction in Spark is that of a resilient distributed dataset (RDD), which represents a read-only collection of objects partitioned across a set of machines that can be rebuilt if a partition is lost.

Users can explicitly cache an RDD in memory across machines and reuse it in multiple MapReduce-like parallel operations.

RDDs achieve fault tolerance through a notion of lineage: if a partition of an RDD is lost, the RDD has enough information about how it was derived from other RDDs to be able to rebuild just that partition. Although RDDs are not a general shared memory abstraction, they represent a sweet-spot between expressivity on the one hand and scalability and reliability on the other hand, and we have found them well-suited for a variety of applications.

Spark is implemented in Scala [5], a statically-typed high-level programming language for the Java VM, and exposes a functional programming interface similar to PACT/REDO [29]. In addition, Spark can be used interactively from a modulated version of the Scala interpreter, which allows the user to define RDDs, functions, variables and classes and use them in parallel operations on a cluster. We believe that Spark is the first system to allow an efficient, general-purpose programming language to be used interactively to process large datasets on a cluster.

Although our implementation of Spark is still a prototype, early experience with the system is encouraging. We show that Spark can outperform Hadoop by 10x in iterative machine learning workloads and can be used interactively to scan a 70 GB dataset with sub-second latency.

This paper is organized as follows. Section 2 describes
Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing

Matei Zaharia, Mosharaf Chowdhury, Togtugas Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica

University of California, Berkeley

Abstract

We present Resilient Distributed Datasets (RDDs), a distributed memory abstraction that lets programmers perform in-memory computations on large clusters in a fault-tolerant manner. RDDs are motivated by two types of applications that current computing frameworks handle inefficiently: iterative algorithms and interactive data mining tools.

RDDS are motivated by two types of applications that current computing frameworks handle inefficiently: iterative algorithms and interactive data mining tools.

In both cases, keeping data in memory can improve performance by an order of magnitude.

Best Paper Award and Honorable Mention for Community Award
- NSDI 2012

April 2012

Discretized Streams: Fault-Tolerant Streaming Computation at Scale
Matei Zaharia, Tathagata Das, Haozuan Li, Timothy Hunter, Scott Shenker, Ion Stoica
University of California, Berkeley

Abstract
Many “big data” applications must act on data in real time. Running these applications at ever-larger scales requires parallel platforms that automatically handle faults and failures. Unfortunately, current distributed stream processing models provide fault recovery in an expensive manner, requiring hot replication or long recovery times, and do not handle failures. We propose a new processing model, discretized streams (D-Streams), that overcomes these challenges. D-Streams enable a parallel recovery mechanism that improves efficiency over traditional replication and backup schemes, and tolerates failures. We show that they support a rich set of queries while retaining high throughput similar to single-node systems. Applying linear scaling to 100 nodes, sub-second latency, and sub-second fault recovery. Finally, D-Streams can easily be composed with batch and interactive query models like MapReduce, enabling rich applications that combine these models. We implement D-Streams in a system called Spark Streaming.

1 Introduction
Much of “big data” is received in real time, and is most valuable at its time of arrival. For example, a social network may wish to detect trending conversation topics in

TwitterUtils.createStream(...)
  .filter(_.getText.contains("Spark"))
  .countByWindow(Seconds(5))

- 2 Streaming Paper(s) have been cited 138 times
Seemlessly mix SQL queries with Spark programs.

sqlCtx = new HiveContext(sc)
results = sqlCtx.sql("SELECT * FROM people")
names = results.map(lambda p: p.name)

(Will be published in the upcoming weeks for SIGMOD 2015)
graph = Graph(vertices, edges)
messages = spark.textFile("hdfs://...")

graph2 = graph.joinVertices(messages) {
(id, vertex, msg) => ...
}

BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data

Sameer Agarwal*, Barzan Mozafari*, Aurojit Panda*, Henry Milner*, Samuel Madden*, Ion Stoica*

†University of California, Berkeley  *Massachusetts Institute of Technology  *Conviva Inc.
{sameerag, apanda, henrym, istoica}@cs.berkeley.edu, {barzan, madden}@csail.mit.edu

Abstract

In this paper, we present BlinkDB, a massively parallel, approximate query engine for running interactive SQL queries on large volumes of data. BlinkDB allows users to trade-off query accuracy for response time, enabling interactive queries over massive data by running queries on data samples and presenting results annotated with meaningful error bars. To achieve this, BlinkDB uses two key ideas: (1) an adaptive optimization framework that builds and maintains a set of multi-dimensional stratified samples from original data over time, and (2) a dynamic sample selection strategy that selects an appropriately sized sample based on a query's accuracy or response time requirements. We evaluate BlinkDB against the well-known TPC-H benchmarks and a real-world analytic workload derived from Conviva Inc., a company that manages video distribution over the Internet. Our experiments on a 16 node cluster show that BlinkDB can answer queries on up to 17 TBs of data in less than 2 seconds (over 1000x faster than Hive), within an error of ±10%.

1. Introduction

Modern data analytics applications involve computing aggregates over a large number of records to roll-up web clicks, censoring of large amounts of data by trading result accuracy for response time and space. These techniques include sampling [10, 14], sketches [13], and on-line aggregation [9]. To illustrate the utility of such techniques, consider the following simple query that computes the average sessionTime over all users originating in New York:

SELECT AVG(sessionTime) FROM Sessions
WHERE City = 'New York'

Suppose the Sessions table contains 100 million tuples for New York, and cannot fit in memory. In that case, the above query may take a long time to execute, since disk reads are expensive, and such a query would need multiple disk accesses to stream through all the tuples. Suppose we instead executed the same query on a sample containing only 10,000 New York tuples, such that the entire sample fits in memory. This would be orders of magnitude faster, while still providing an approximate result within a few percent of the actual value, an accuracy good enough for many practical purposes. Using sampling theory we could even provide confidence bounds on the accuracy of the answer [16].

Previously described approximation techniques make different trade-offs between efficiency and the generality of the

SELECT avg(sessionTime) FROM Table
WHERE city='San Francisco'
WITHIN 2 SECONDS
ERROR 0.1 CONFIDENCE 95.0%

Queries with Time Bounds

Queries with Error Bounds

Learning Spark

Lightning-Fast Data Analysis

Holden Karau, Andy Konwinski, Patrick Wendell & Matei Zaharia

eBook: $33.99
Print: $39.99

Shipping now!

$30 @ Amazon:
Spark
Lightning-fast cluster computing

Download Libraries Documentation Examples Community FAQ

Latest News
Spark 1.2.1 released (Feb 09, 2015)
Spark Summit East agenda posted, CFP open for West (Jan 21, 2015)
Spark 1.2.0 released (Dec 18, 2014)
Spark 1.1.1 released (Nov 26, 2014)

Spark Community

Mailing Lists
Get help using Spark or contribute to the project on our mailing lists:

- user@spark.apache.org is for usage questions, help, and announcements. (subscribe) (unsubscribe) (archives)
- dev@spark.apache.org is for people who want to contribute code to Spark. (subscribe) (unsubscribe) (archives)
Lab: Intro to Spark 1.1 on DSE 4.6

Lab created on: Sept 2, 2014 (last updated Dec 9, 2014)

License: ☑️ Biolistic

Objective:
This lab will introduce you to using Apache Spark 1.1 on DataStax Enterprise Edition 4.6.0 in the Amazon cloud. The lab assumes that the audience is a beginner to both Cassandra and Spark. So the document walks the reader through installing DSE, learning Cassandra and then learning Spark. The ultimate goal here is to introduce students to Cassandra + Spark in a devops manner: looking at config files, writing some simple CQL or Spark code, breaking things and troubleshooting issues, exploring the Spark source code, etc. Although the ideal way to use this lab is actually type + run the commands in a parallel environment, the lab can still be used for purely reading. All the output of the commands are pasted in this lab, so you can get a very clear idea of what would happen if you had actually run the command.

The following high level steps are part of this lab:
- Connect via SSH to your EC2 instance
- Create a new keyspace and table in C* and add data to it
- Start the scala based Spark shell
- Import the fresh data into a Spark RDD

http://tinyurl.com/dsesparklab

- 102 pages
- DevOps style
- For complete beginners
- Includes:
  - Spark Streaming
  - Dangers of GroupByKey vs. ReduceByKey
Labs: Intro to HDFS/YARN & Apache Spark on CDH 5.2

Lab created on: Dec, 2014
(please send edits and corrections to): sameerf@dataricks.com

Estimated lab completion time: 2 hours (spread throughout the day)

License: ☞☞☞☞

Objective:
This lab will introduce you to using 3 Hadoop ecosystem components in Cloudera’s distribution: HDFS, Spark 1.1.0 and YARN. The lab will first walk you through the Cloudera Manager installation on a VM in AWS, followed by a CDH 5.2 binaries deployment on the same node. Then the lab will introduce students to Hadoop in a DevOps manner: experimenting with the distributed file system, looking at the XML config files, running a batch analytics workload with Spark from disk and from memory, writing some simple scala Spark code, running SQL commands with Spark SQL, breaking things and troubleshooting issues, etc.

The following high level steps are in the initial part of this lab:
- Connect via SSH to your Amazon instance
- Install Cloudera Manager and CDH 5.2
- Create a new folder in HDFS and add data files to it
- Start the scala based Spark shell
- Import the fresh data into Spark a RDD

http://tinyurl.com/cdhsparklab

- 109 pages
- DevOps style
- For complete beginners
- Includes:
  - PySpark
  - Spark SQL
  - Spark-submit
A community index of packages for Apache Spark.

33 packages

databricks/spark-avro
Integration utilities for using Spark with Apache Avro data
@pwendell / Latest release: 0.1 (11/27/14) / Apache 2.0 / ★★★★★ (95)

3 sql 3 input 2 library

dibbhatt/kafka-spark-consumer
Low Level Kafka-Spark Consumer
@dibbhatt / No release yet / ★★★★★ (3)

2 streaming 1 kafka

sigmoidanalytics/spork
Pig on Apache Spark
Next slide is only for on-site students...
Spark Summit East NYC Pre-Class Questionnaire

Welcome to the Advanced Spark DevOps class! Please take a few minutes to complete this anonymous survey. (Note that Databricks is only capturing the timestamp of when you submit this survey. We do not collect your name, email or any other identifying information.)

In 5 minutes the instructor will share a new link with you containing the group's visualized results.

* Required

Which of the following Spark components are you mostly interested in? *

- Core Spark
- Spark SQL
- Spark Streaming
- MLib (machine learning)
- GraphX
- BlinkDB
RDD FUNDAMENTALS
INTERACTIVE SHELL

```
ubuntu@ip-10-0-53-24:~$ dse spark
Welcome to

version 0.9.1

Using Scala version 2.10.3 (Java HotSpot(TM) 64-Bit Server VM, Java 1.7.0_51)
Type in expressions to have them evaluated.
Type :help for more information.
Creating SparkContext...
Created spark context...
Spark context available as sc.
Type in expressions to have them evaluated.
Type :help for more information.

scala> val myRDD = sc.cassandraTable("tinykeyspace", "keyvalueable")

scala> myRDD.count()
res2: Long = 5
```

(Scala & Python only)
more partitions = more parallelism
An RDD can be created 2 ways:

- Parallelize a collection
- Read data from an external source (S3, C*, HDFS, etc)
# Parallelize in Python
```python
wordsRDD = sc.parallelize(\["fish", "cats", "dogs"])
```

- Take an existing in-memory collection and pass it to SparkContext’s parallelize method

- Not generally used outside of prototyping and testing since it requires entire dataset in memory on one machine

// Parallelize in Scala
```scala
val wordsRDD = sc.parallelize(List("fish", "cats", "dogs"))
```

JavaRDD<String> wordsRDD = sc.parallelize(Arrays.asList("fish", "cats", "dogs"));
# Read a local txt file in Python
linesRDD = sc.textFile("/path/to/README.md")

// Read a local txt file in Scala
val linesRDD = sc.textFile("/path/to/README.md")

// Read a local txt file in Java
JavaRDD<String> lines = sc.textFile("/path/to/README.md");

- There are other methods to read data from HDFS, C*, S3, HBase, etc.
```python
errorsRDD = errorsRDD.coalesce(2)
cleanedRDD = cleanedRDD.collect()
```
Execute DAG!
logLinesRDD

.errorsRDD

.cleanedRDD

.filter($fix$)

.coalesce(2)

.collect()
Driver

logLinesRDD

cleanedRDD

data

errorsRDD

Pipelined Stage-1

.logLinesRDD.filter().coalesce(2, shuffle=False).collect()
Driver

logLinesRDD

errorsRDD

cleanedRDD
Driver data
```python
# logLinesRDD

# errorsRDD

# Error, ts, msg1
# Error, ts, msg3
# Error, ts, msg1
# Error, ts, msg4
# Error, ts, msg1

# cleanedRDD

# .filter(fix)

# Error, ts, msg1
# Error, ts, msg4

# errorMsg1RDD

# .saveToCassandra()

# .count()

# 5
```
logLinesRDD

errorsRDD

Error, ts, msg1
Error, ts, msg3
Error, ts, msg1
Error, ts, msg4
Error, ts, msg1

cleanedRDD

.filter()

Error, ts, msg1
Error, ts, msg4
Error, ts, msg1

errorMsg1RDD

.collect()

.count()

5

.saveToCassandra()
RDD GRAPH

Dataset-level view:

logLinesRDD (HadoopRDD)
Path = hdfs://...

errorsRDD (filteredRDD)

func = _.contains(...) shouldCache=false

Partition-level view:

logLinesRDD

Task-1 Task-2 Task-3 Task-4

errorsRDD
LIFECYCLE OF A SPARK PROGRAM

1) Create some input RDDs from external data or parallelize a collection in your driver program.

2) Lazily transform them to define new RDDs using transformations like `filter()` or `map()`

3) Ask Spark to `cache()` any intermediate RDDs that will need to be reused.

4) Launch actions such as `count()` and `collect()` to kick off a parallel computation, which is then optimized and executed by Spark.
Most transformations are element-wise (they work on one element at a time), but this is not true for all transformations.
<table>
<thead>
<tr>
<th>ACTIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>reduce()</td>
</tr>
<tr>
<td>collect()</td>
</tr>
<tr>
<td>count()</td>
</tr>
<tr>
<td>first()</td>
</tr>
<tr>
<td>take()</td>
</tr>
<tr>
<td>takeSample()</td>
</tr>
<tr>
<td>saveToCassandra()</td>
</tr>
</tbody>
</table>
TYPES OF RDDS

- HadoopRDD
- FilteredRDD
- MappedRDD
- PairRDD
- ShuffledRDD
- UnionRDD
- PythonRDD
- DoubleRDD
- JdbcRDD
- JsonRDD
- SchemaRDD
- VertexRDD
- EdgeRDD
- CassandraRDD (DataStax)
- GeoRDD (ESRI)
- EsSpark (ElasticSearch)
/*
 * Licensed to the Apache Software Foundation (ASF) under one or more
 * contributor license agreements. See the NOTICE file distributed with
 * this work for additional information regarding copyright ownership.
 * The ASF licenses this file to You under the Apache License, Version 2.0
 * (the "License"); you may not use this file except in compliance with
 * the License. You may obtain a copy of the License at
 * http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

package org.apache.spark.rdd
<table>
<thead>
<tr>
<th>File Name</th>
<th>Description</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>AsyncRDDActions.scala</td>
<td>[SPARK-4397][Core] Cleanup 'import SparkContext._' in core</td>
<td>3 months</td>
</tr>
<tr>
<td>BinaryFileRDD.scala</td>
<td>[SPARK-4719][API] Consolidate various narrow dep RDD classes with Map...</td>
<td>3 months</td>
</tr>
<tr>
<td>BlockRDD.scala</td>
<td>[SPARK-4077][Streaming] WriteAheadLogBackedBlockRDD to read received</td>
<td>4 months</td>
</tr>
<tr>
<td>CartesianRDD.scala</td>
<td>[SPARK-4090] Only throw IOException from [write</td>
<td>read][Object][External]</td>
</tr>
<tr>
<td>CheckpointRDD.scala</td>
<td>[SPARK-4014] Add TaskContext attemptNumberOfDeprecateTaskContext...</td>
<td>4 months</td>
</tr>
<tr>
<td>CoGroupedRDD.scala</td>
<td>[SPARK-3280] All fields in TaskMetrics should be private and use gett...</td>
<td>29 days</td>
</tr>
<tr>
<td>CoalescedRDD.scala</td>
<td>[SPARK-4758] Fix driver hanging from coalescing partitions</td>
<td>2 months</td>
</tr>
<tr>
<td>DoubleRDDFunctions.scala</td>
<td>[SPARK-4397][Core] Cleanup 'import SparkContext._' in core</td>
<td>3 months</td>
</tr>
<tr>
<td>EmptyRDD.scala</td>
<td>SPARK-1093: Annotate developer and experimental APIs</td>
<td>10 months</td>
</tr>
<tr>
<td>HadoopRDD.scala</td>
<td>[SPARK-4874][CORE] Collect record count metrics</td>
<td>10 months</td>
</tr>
<tr>
<td>JdbcRDD.scala</td>
<td>SPARK-5239 [CORE] JdbcRDD throws &quot;java.lang.AbstractMethodError: orac...</td>
<td>7 days</td>
</tr>
</tbody>
</table>
1) Set of partitions ("splits")
2) List of dependencies on parent RDDs
3) Function to compute a partition given parents
4) Optional preferred locations
5) Optional partitioning info for k/v RDDs (Partitioner)

This captures all current Spark operations!
EXAMPLE: HADOOPRDD

* Partitions = one per HDFS block
* Dependencies = none
* Compute (partition) = read corresponding block
  * preferredLocations (part) = HDFS block location
  * Partitioner = none
EXAMPLE: FILTEREDRDD

- Partitions = same as parent RDD
- Dependencies = “one-to-one” on parent
- Compute (partition) = compute parent and filter it

- preferredLocations (part) = none (ask parent)
- Partitioner = none
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)
val cassandraRDD = sc
  .cassandraTable("ks", "mytable")
  .select("col-1", "col-3")
  .where("col-5 = ?", "blue")
Start the Spark shell by passing in a custom cassandra.input.split.size:

```bash
ubuntu@ip-10-0-53-24:~$ dse spark -Dspark.cassandra.input.split.size=2000
```

Welcome to

```
  __
 /__
___
_____/
/___/
_
/____
```

version 0.9.1

Using Scala version 2.10.3 (Java HotSpot(TM) 64-Bit Server VM, Java 1.7.0_51)
Type in expressions to have them evaluated.
Type :help for more information.
Creating SparkContext...
Created spark context..
Spark context available as sc.
Type in expressions to have them evaluated.
Type :help for more information.

scala>

The cassandra.input.split.size parameter defaults to 100,000. This is the approximate number of physical rows in a single Spark partition. If you have really wide rows (thousands of columns), you may need to lower this value. The higher the value, the fewer Spark tasks are created. Increasing the value too much may limit the parallelism level."
- Open Source
- Implemented mostly in Scala
- Scala + Java APIs
- Does automatic type conversions
Spark Cassandra Connector

Lightning-fast cluster computing with Spark and Cassandra

This library lets you expose Cassandra tables as Spark RDDs, write Spark RDDs to Cassandra tables, and execute arbitrary CQL queries in your Spark applications.

Features

- Compatible with Apache Cassandra version 2.0 or higher and DataStax Enterprise 4.5
- Compatible with Apache Spark 1.0 and 1.1
- Exposes Cassandra tables as Spark RDDs
- Maps table rows to CassandraRow objects or tuples
- Offers customizable object mapper for mapping rows to objects of user-defined classes
- Saves RDDs back to Cassandra by implicit `saveToCassandra` call
- Converts data types between Cassandra and Scala
- Supports all Cassandra data types including collections
- Filters rows on the server side via the CQL `WHERE` clause
- Allows for execution of arbitrary CQL statements
- Plays nice with Cassandra Virtual Nodes
“Simple things should be simple, complex things should be possible”

- Alan Kay
DEMO: DATABRICKS CLOUD GUI
SPARK RESOURCE MANAGERS
WAYS TO RUN SPARK

- Local
- Standalone Scheduler
- YARN
- Mesos

Static Partitioning
Dynamic Partitioning
History: 2 MR APPS RUNNING
LOCAL MODE
val conf = new SparkConf()
  .setMaster("local[12]")
  .setAppName("MyFirstApp")
  .set("spark.executor.memory", "3g")
val sc = new SparkContext(conf)
different spark-env.sh

- SPARK_WORKER_CORES

> ./bin/spark-submit --name "SecondApp"
   --master spark://host1:port1
   myApp.jar
different spark-env.sh

- SPARK_WORKER_CORES

> ./bin/spark-submit --name "SecondApp"
  --master spark://host1:port1,host2:port2
  myApp.jar

I'm HA via ZooKeeper

More Masters can be added live

spark-env.sh
- SPARK_LOCAL_DIRS
SPARK_WORKER_INSTANCES: [default: 1] # of worker instances to run on each machine

SPARK_WORKER_CORES: [default: ALL] # of cores to allow Spark applications to use on the machine

SPARK_WORKER_MEMORY: [default: TOTAL RAM – 1 GB] Total memory to allow Spark applications to use on the machine

SPARK_DAEMON_MEMORY: [default: 512 MB] Memory to allocate to the Spark master and worker daemons themselves
- Apps submitted will run in FIFO mode by default

**spark.cores.max**: maximum amount of CPU cores to request for the application from across the cluster

**spark.executor.memory**: Memory for each executor
Spark Master at spark://10.0.64.177:7077

Total potential memory this Spark cluster has access to is 4 GB (aka sum of how much memory each Worker, below, has access to)

<table>
<thead>
<tr>
<th>Workers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Id</td>
</tr>
<tr>
<td>---------</td>
</tr>
<tr>
<td>worker-20140905191420-10.0.64.177-33571</td>
</tr>
</tbody>
</table>

Amount of potential memory this particular Spark worker has access to

Running Applications

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Cores</th>
<th>Memory per Node</th>
<th>Submitted Time</th>
<th>User</th>
<th>State</th>
<th>Duration</th>
</tr>
</thead>
</table>

Completed Applications

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Cores</th>
<th>Memory per Node</th>
<th>Submitted Time</th>
<th>User</th>
<th>State</th>
<th>Duration</th>
</tr>
</thead>
</table>
Spark Master at spark://10.0.12.60:7077

URL: spark://10.0.12.60:7077
Workers: 1
Cores: 3 Total, 3 Used
Memory: 7.7 GB Total, 512.0 MB Used
Applications: 1 Running, 0 Completed
Drivers: 0 Running, 0 Completed
Status: ALIVE

**Workers**

<table>
<thead>
<tr>
<th>Id</th>
<th>Address</th>
<th>State</th>
<th>Cores</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>worker-20141110195851-10.0.12.60-35935</td>
<td>10.0.12.60.35935</td>
<td>ALIVE</td>
<td>(3 Used)</td>
<td>7.7 GB (512.0 MB Used)</td>
</tr>
</tbody>
</table>

**Running Applications**

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Cores</th>
<th>Memory per Node</th>
<th>Submitted Time</th>
<th>User</th>
<th>State</th>
<th>Duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>app-20141110204831-0000</td>
<td>Spark shell</td>
<td>3</td>
<td>512.0 MB</td>
<td>2014/11/10 20:48:31</td>
<td>ec2-user</td>
<td>RUNNING</td>
<td>23 min</td>
</tr>
</tbody>
</table>

**Completed Applications**

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Cores</th>
<th>Memory per Node</th>
<th>Submitted Time</th>
<th>User</th>
<th>State</th>
<th>Duration</th>
</tr>
</thead>
</table>
Spark Worker at 10.0.12.60:35935

ID: worker-20141110195851-10.0.12.60-35935
Master URL: spark://10.0.12.60:7077
Cores: 3 (3 Used)
Memory: 7.7 GB (512.0 MB Used)

Back to Master

Running Executors (1)

<table>
<thead>
<tr>
<th>ExecutorID</th>
<th>Cores</th>
<th>State</th>
<th>Memory</th>
<th>Job Details</th>
<th>Logs</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>RUNNING</td>
<td>512.0 MB</td>
<td>ID: app-20141204831-0000</td>
<td>stdout stderr</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Name: Spark shell</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>User: cassandra</td>
<td></td>
</tr>
</tbody>
</table>
## Spark Jobs

**Total Duration:** 39 min  
**Scheduling Mode:** FIFO  
**Active Jobs:** 0  
**Completed Jobs:** 4  
**Failed Jobs:** 0  

### Active Jobs (0)

<table>
<thead>
<tr>
<th>Job Id</th>
<th>Description</th>
<th>Submitted</th>
<th>Duration</th>
<th>Stages: Succeeded/Total</th>
<th>Tasks (for all stages): Succeeded/Total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Completed Jobs (4)

<table>
<thead>
<tr>
<th>Job Id</th>
<th>Description</th>
<th>Submitted</th>
<th>Duration</th>
<th>Stages: Succeeded/Total</th>
<th>Tasks (for all stages): Succeeded/Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>collect at &lt;console&gt;:19</td>
<td>2014/12/01 16:18:24</td>
<td>39 ms</td>
<td>1/1 (1 skipped)</td>
<td>2/2 (2 skipped)</td>
</tr>
<tr>
<td>2</td>
<td>collect at &lt;console&gt;:19</td>
<td>2014/12/01 16:18:22</td>
<td>55 ms</td>
<td>1/1 (1 skipped)</td>
<td>2/2 (2 skipped)</td>
</tr>
<tr>
<td>1</td>
<td>collect at &lt;console&gt;:19</td>
<td>2014/12/01 16:18:07</td>
<td>0.2 s</td>
<td>2/2</td>
<td>4/4</td>
</tr>
<tr>
<td>0</td>
<td>count at &lt;console&gt;:15</td>
<td>2014/12/01 16:17:39</td>
<td>0.3 s</td>
<td>1/1</td>
<td>2/2</td>
</tr>
</tbody>
</table>

### Failed Jobs (0)

<table>
<thead>
<tr>
<th>Job Id</th>
<th>Description</th>
<th>Submitted</th>
<th>Duration</th>
<th>Stages: Succeeded/Total</th>
<th>Tasks (for all stages): Succeeded/Total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Spark Stages (for all jobs)

**Total Duration**: 39 min  
**Scheduling Mode**: FIFO  
**Active Stages**: 0  
**Completed Stages**: 5  
**Failed Stages**: 0

### Active Stages (0)

<table>
<thead>
<tr>
<th>Stage Id</th>
<th>Description</th>
<th>Submitted</th>
<th>Duration</th>
<th>Tasks: Succeeded/Total</th>
<th>Input</th>
<th>Output</th>
<th>Shuffle Read</th>
<th>Shuffle Write</th>
</tr>
</thead>
</table>

### Completed Stages (5)

<table>
<thead>
<tr>
<th>Stage Id</th>
<th>Description</th>
<th>Submitted</th>
<th>Duration</th>
<th>Tasks: Succeeded/Total</th>
<th>Input</th>
<th>Output</th>
<th>Shuffle Read</th>
<th>Shuffle Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td><code>collect at &lt;console&gt;</code>: 19</td>
<td>2014/12/01 16:18:24</td>
<td>28 ms</td>
<td>2/2</td>
<td>502.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td><code>collect at &lt;console&gt;</code>: 19</td>
<td>2014/12/01 16:18:22</td>
<td>45 ms</td>
<td>2/2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td><code>collect at &lt;console&gt;</code>: 19</td>
<td>2014/12/01 16:18:07</td>
<td>69 ms</td>
<td>2/2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td><code>map at &lt;console&gt;</code>: 16</td>
<td>2014/12/01 16:18:07</td>
<td>76 ms</td>
<td>2/2</td>
<td>254.0</td>
<td>737.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td><code>count at &lt;console&gt;</code>: 15</td>
<td>2014/12/01 16:17:40</td>
<td>0.2 s</td>
<td>2/2</td>
<td>254.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RDD Name</td>
<td>Storage Level</td>
<td>Cached Partitions</td>
<td>Fraction Cached</td>
<td>Size in Memory</td>
<td>Size in Tachyon</td>
<td>Size on Disk</td>
<td></td>
<td></td>
</tr>
<tr>
<td>----------</td>
<td>---------------</td>
<td>-------------------</td>
<td>----------------</td>
<td>----------------</td>
<td>----------------</td>
<td>--------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Memory Deserialized 1x Replicated</td>
<td>2</td>
<td>100%</td>
<td>562.0 B</td>
<td>0.0 B</td>
<td>0.0 B</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
RDD Storage Info for 5

Storage Level: Memory Deserialized 1x Replicated
Cached Partitions: 2
Total Partitions: 2
Memory Size: 552.0 B
Disk Size: 0.0 B

Data Distribution on 1 Executors

<table>
<thead>
<tr>
<th>Host</th>
<th>Memory Usage</th>
<th>Disk Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>localhost:38329</td>
<td>552.0 B (265.4 MB Remaining)</td>
<td>0.0 B</td>
</tr>
</tbody>
</table>

2 Partitions

<table>
<thead>
<tr>
<th>Block Name</th>
<th>Storage Level</th>
<th>Size in Memory</th>
<th>Size on Disk</th>
<th>Executors</th>
</tr>
</thead>
<tbody>
<tr>
<td>rdd_5_0</td>
<td>Memory Deserialized 1x Replicated</td>
<td>424.0 B</td>
<td>0.0 B</td>
<td>localhost:38329</td>
</tr>
<tr>
<td>rdd_5_1</td>
<td>Memory Deserialized 1x Replicated</td>
<td>128.0 B</td>
<td>0.0 B</td>
<td>localhost:38329</td>
</tr>
</tbody>
</table>
### Environment

#### Runtime Information

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java Home</td>
<td>/usr/java/jdk1.7.0_67/jre</td>
</tr>
<tr>
<td>Java Version</td>
<td>1.7.0_67 (Oracle Corporation)</td>
</tr>
<tr>
<td>Scala Version</td>
<td>version 2.10.4</td>
</tr>
</tbody>
</table>

#### Spark Properties

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>spark.app.id</td>
<td>local-1417468637156</td>
</tr>
<tr>
<td>spark.app.name</td>
<td>Spark shell</td>
</tr>
<tr>
<td>spark.driver.host</td>
<td>ip-10-0-125-125.us-west-2.compute.internal</td>
</tr>
<tr>
<td>spark.driver.port</td>
<td>52091</td>
</tr>
<tr>
<td>spark.executor.id</td>
<td>driver</td>
</tr>
<tr>
<td>spark.filesystem.uri</td>
<td><a href="http://10.0.125.125:58999">http://10.0.125.125:58999</a></td>
</tr>
<tr>
<td>spark.jars</td>
<td></td>
</tr>
<tr>
<td>spark.master</td>
<td>local[“”]</td>
</tr>
<tr>
<td>spark.repl.class.uri</td>
<td><a href="http://10.0.125.125:57870">http://10.0.125.125:57870</a></td>
</tr>
<tr>
<td>spark.scheduler.mode</td>
<td>FIFO</td>
</tr>
<tr>
<td>spark.tachyonStore.folderName</td>
<td>spark-a5c91951-a6b4-4425-bad0-a1e2e9146a70</td>
</tr>
</tbody>
</table>

#### System Properties

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Executors (1)

### Memory
- 552.0 B Used (265.4 MB Total)
- Disk: 0.0 B Used

<table>
<thead>
<tr>
<th>Executor ID</th>
<th>Address</th>
<th>RDD Blocks</th>
<th>Memory Used</th>
<th>Disk Used</th>
<th>Active Tasks</th>
<th>Failed Tasks</th>
<th>Complete Tasks</th>
<th>Total Tasks</th>
<th>Task Time</th>
<th>Input</th>
<th>Shuffle Read</th>
<th>Shuffle Write</th>
<th>Thread Dump</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;driver&gt;</td>
<td>localhost:38329</td>
<td>2</td>
<td>552.0 B / 265.4 MB</td>
<td>0.0 B</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>10</td>
<td>740 ms</td>
<td>1050.0 B</td>
<td>0.0 B</td>
<td>737.0 B</td>
<td>Thread Dump</td>
</tr>
</tbody>
</table>
Thread dump for executor <driver>

Updated at 2014/12/01 16:57:39

Expand All

Thread 1: main (RUNNABLE)
Thread 2: Reference Handler (WAITING)
Thread 3: Finalizer (WAITING)
Thread 4: Signal Dispatcher (RUNNABLE)
Thread 11: qtp1844705036-11 (TIMED_WAITING)
Thread 12: qtp1844705036-12 (TIMED_WAITING)
Thread 13: qtp1844705036-13 (TIMED_WAITING)
Thread 14: qtp1844705036-14 Acceptor0 SocketConnector@0.0.0.0.6787 (RUNNABLE)
Thread 15: qtp1844705036-15 (TIMED_WAITING)
Thread 16: qtp1844705036-16 (TIMED_WAITING)
Thread 17: qtp1844705036-17 (TIMED_WAITING)
Thread 18: qtp1844705036-18 (TIMED_WAITING)
Thread 56: qtp1837961181-56 (TIMED_WAITING)
Thread 57: qtp1837961181-57 (TIMED_WAITING)
Thread 58: qtp1837961181-58 (TIMED_WAITING)
Thread 59: Timer-0 (WAITING)
Thread 60: Driver Heartbeater (TIMED_WAITING)
Thread 69: shuffle-server-0 (RUNNABLE)

sun.nio.ch.EPollArrayWrapper.epollWait(Native Method)
sun.nio.ch.SelectorImpl1.lockAndDoSelect(SelectorImpl1.java:87)
sun.nio.ch.SelectorImpl1.select(SelectorImpl1.java:88)
io.netty.channel.nio.kioeventloop.select(kioeventloop.java:622)
io.netty.channel.nio.kioeventloop.run(kioeventloop.java:318)
java.lang.Thread.run(Thread.java:745)

Thread 78: Spark Context Cleaner (TIMED_WAITING)
Thread 79: sparkDriver-akka.actor.default-dispatcher-14 (TIMED_WAITING)
Thread 83: task-result-getter-0 (WAITING)
Thread 84: task-result-getter-1 (WAITING)
Thread 85: ForkJoinPool-3-worker-7 (WAITING)
YARN MODE
NodeManager
Resource Manager
Scheduler
Apps Master
I'm HA via ZooKeeper

Client #1

Client #2

App Master

Container

App Master

Container

Container

Container
NodeManager

Resource
Manager

App Master

Container

Executor

Client #1

Driver

SPARK YARN
(client mode)

NodeManager

Container

Executor

RD

T

RD

T

RD

T

RD

T
- Does not support Spark Shells
YARN settings

--num-executors: controls how many executors will be allocated

--executor-memory: RAM for each executor

--executor-cores: CPU cores for each executor

Dynamic Allocation:

spark.dynamicAllocation.enabled
spark.dynamicAllocation.minExecutors
spark.dynamicAllocation.maxExecutors
spark.dynamicAllocation.sustainedSchedulerBacklogTimeout (N)
spark.dynamicAllocation.schedulerBacklogTimeout (M)
spark.dynamicAllocation.executorIdleTimeout (K)

YARN resource manager UI: http://<ip address>:8088
(No apps running)
[ec2-user@ip-10-0-72-36 ~]$ spark-submit --class org.apache.spark.examples.SparkPi --deploy-mode client --master yarn /opt/cloudera/parcels/CDH-5.2.1-1.cdh5.2.1.p0.12/jars/spark-examples-1.1.0-cdh5.2.1-hadoop2.5.0-cdh5.2.1.jar 10
App running in *client* mode

### All Applications

#### Cluster Metrics

<table>
<thead>
<tr>
<th>Apps Submitted</th>
<th>Apps Pending</th>
<th>Apps Running</th>
<th>Apps Completed</th>
<th>Containers Running</th>
<th>Containers Pending</th>
<th>Containers Reserved</th>
<th>Memory Used</th>
<th>Memory Total</th>
<th>Memory Reserved</th>
<th>VCore Used</th>
<th>VCore Total</th>
<th>Active Nodes</th>
<th>Decommissioned Nodes</th>
<th>Lost Nodes</th>
<th>Unhealthy Nodes</th>
<th>Rebooted Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.9 GB</td>
<td>3.46 GB</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

#### User Metrics for dr_who

<table>
<thead>
<tr>
<th>Apps Submitted</th>
<th>Apps Pending</th>
<th>Apps Running</th>
<th>Apps Completed</th>
<th>Containers Running</th>
<th>Containers Pending</th>
<th>Containers Reserved</th>
<th>Memory Used</th>
<th>Memory Total</th>
<th>Memory Reserved</th>
<th>VCore Used</th>
<th>VCore Total</th>
<th>Active Nodes</th>
<th>Decommissioned Nodes</th>
<th>Lost Nodes</th>
<th>Unhealthy Nodes</th>
<th>Rebooted Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.9 GB</td>
<td>3.46 GB</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

### Show 20 entries

<table>
<thead>
<tr>
<th>ID</th>
<th>User</th>
<th>Name</th>
<th>Application Type</th>
<th>Queue</th>
<th>StartTime</th>
<th>FinishTime</th>
<th>State</th>
<th>FinalStatus</th>
<th>Progress</th>
<th>Tracking UI</th>
</tr>
</thead>
<tbody>
<tr>
<td>app_no_1</td>
<td>user</td>
<td>Spark PI</td>
<td>SPARK</td>
<td>root</td>
<td>Thu, 04 Dec 2014 15:30:43 GMT</td>
<td>Thu, 04 Dec 2014 15:31:14 GMT</td>
<td>FINISHED</td>
<td>SUCEEDED</td>
<td>History</td>
<td></td>
</tr>
<tr>
<td>app_no_2</td>
<td>user</td>
<td>Spark PI</td>
<td>SPARK</td>
<td>root</td>
<td>Thu, 04 Dec 2014 15:26:19 GMT</td>
<td>Thu, 04 Dec 2014 15:26:19 GMT</td>
<td>FINISHED</td>
<td>SUCEEDED</td>
<td>History</td>
<td></td>
</tr>
<tr>
<td>app_no_3</td>
<td>user</td>
<td>Spark PI</td>
<td>SPARK</td>
<td>root</td>
<td>Thu, 04 Dec 2014 15:25:35 GMT</td>
<td>Thu, 04 Dec 2014 15:25:35 GMT</td>
<td>FINISHED</td>
<td>SUCEEDED</td>
<td>History</td>
<td></td>
</tr>
</tbody>
</table>
### Application Overview

- **User:** ec2-user  
- **Name:** Spark Pi  
- **Application Type:** SPARK  
- **State:** FINISHED  
- **FinalStatus:** SUCCEEDED  
- **Started:** 4-Dec-2014 10:30:43  
- **Elapsed:** 3 sec  
- **Tracking URL:** History  
- **Diagnostics:**

### Application Metrics

- **Total Resource Preempted:** `<memory:0, vCores:0>`  
- **Total Number of Non-AM Containers Preempted:** 0  
- **Total Number of AM Containers Preempted:** 0  
- **Resource Preempted from Current Attempt:** `<memory:0, vCores:0>`  
- **Number of Non-AM Containers Preempted from Current Attempt:** 0  
- **Aggregate Resource Allocation:** 57388 MB-seconds, 45 vcore-seconds

### ApplicationMaster

<table>
<thead>
<tr>
<th>Attempt Number</th>
<th>Start Time</th>
<th>Node</th>
<th>Logs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4-Dec-2014 10:30:43</td>
<td>ip-10-0-72-36.us-west-2.compute.internal.8042</td>
<td>logs</td>
</tr>
</tbody>
</table>
[ec2-user@ip-10-0-72-36 ~]$ spark-submit --class org.apache.spark.examples.SparkPi --deploy-mode cluster --master yarn /opt/cloudera/parcels/CDH-5.2.1-1.cdh5.2.1.p0.12/jars/spark-examples-1.1.0-cdh5.2.1-hadoop2.5.0-cdh5.2.1.jar 10
App running in **cluster** mode
App running in **cluster** mode

---

**Application Overview**

- **User**: ec2-user
- **Name**: org.apache.spark.examples.SparkPI
- **Application Type**: SPARK
- **Application Tags**: 
- **State**: FINISHED
- **FinalStatus**: SUCCEEDED
- **Started**: 4-Dec-2014 10:37:10
- **Elapsed**: 43sec
- **Tracking URL**: History
- **Diagnostics**: 

**Application Metrics**

- **Total Resource Preempted**: <memory:0, vCores:0>
- **Total Number of Non-AM Containers Preempted**: 0
- **Total Number of AM Containers Preempted**: 0
- **Resource Preempted from Current Attempt**: <memory:0, vCores:0>
- **Number of Non-AM Containers Preempted from Current Attempt**: 0
- **Aggregate Resource Allocation**: 83705 MB-seconds, 66 vcore-seconds

**ApplicationMaster**

<table>
<thead>
<tr>
<th>Attempt Number</th>
<th>Start Time</th>
<th>Node</th>
<th>Logs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4-Dec-2014 10:37:10</td>
<td>ip-10-0-72-36.us-west-2.compute.internal.8042</td>
<td>logs</td>
</tr>
</tbody>
</table>
App running in **cluster** mode

---

**Log Type:** stderr
**Log Length:** 22704

Showing 4096 bytes of 22704 total. Click here for the full log.

---

**Log Type:** stdout
**Log Length:** 23

**PI is roughly 3.142362**
### Spark History Server

**Event Log Location:** hdfs://ip-10-0-72-36.us-west-2.compute.internal:8020/user/spark/applicationHistory

**Showing 1-2 of 2**

<table>
<thead>
<tr>
<th>App Name</th>
<th>Started</th>
<th>Completed</th>
<th>Duration</th>
<th>Spark User</th>
<th>Last Updated</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spark shell</td>
<td>2014/12/04 09:14:01</td>
<td>2014/12/04 09:21:19</td>
<td>7.3 min</td>
<td>ec2-user</td>
<td>2014/12/04 09:21:20</td>
</tr>
<tr>
<td>Spark Central Master</td>
<td>Who starts Executors?</td>
<td>Tasks run in</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>----------------------</td>
<td>-----------------------</td>
<td>--------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Local</strong></td>
<td>[none]</td>
<td>Executor</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Standalone</strong></td>
<td>Standalone Master</td>
<td>Executor</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>YARN</strong></td>
<td>YARN App Master</td>
<td>Executor</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Mesos</strong></td>
<td>Mesos Master</td>
<td>Executor</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mesos Slave</td>
<td>Executor</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
spark-submit provides a uniform interface for submitting jobs across all cluster managers

```
bin/spark-submit --master spark://host:7077
  --executor-memory 10g
  my_script.py
```

Table 7-2. Possible values for the --master flag in spark-submit

<table>
<thead>
<tr>
<th>Value</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>spark://host:port</td>
<td>Connect to a Spark Standalone master at the specified port. By default Spark Standalone master's listen on port 7077 for submitted jobs.</td>
</tr>
<tr>
<td>mesos://host:port</td>
<td>Connect to a Mesos cluster master at the specified port. By default Mesos masters listen on port 5050 for submitted jobs.</td>
</tr>
<tr>
<td>yarn</td>
<td>Indicates submission to YARN cluster. When running on YARN you'll need to export HADOOP_CONF_DIR to point the location of your Hadoop configuration directory.</td>
</tr>
<tr>
<td>local</td>
<td>Run in local mode with a single core.</td>
</tr>
<tr>
<td>local[N]</td>
<td>Run in local mode with N cores.</td>
</tr>
<tr>
<td>local[*]</td>
<td>Run in local mode and use as many cores as the machine has.</td>
</tr>
</tbody>
</table>

Source: Learning Spark
MEMORY AND PERSISTENCE
Recommended to use at most only 75% of a machine’s memory for Spark

Minimum Executor heap size should be 8 GB

Max Executor heap size depends… maybe 40 GB (watch GC)

Memory usage is greatly affected by storage level and serialization format
RDD.cache() == RDD.persist(MEMORY_ONLY)

most CPU-efficient option
## Storage

<table>
<thead>
<tr>
<th>RDD Name</th>
<th>Storage Level</th>
<th>Cached Partitions</th>
<th>Fraction Cached</th>
<th>Size in Memory</th>
<th>Size on Disk</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Memory Deserialized 1x Replicated</td>
<td>2</td>
<td>100%</td>
<td>55.6 KB</td>
<td>0.0 B</td>
</tr>
</tbody>
</table>
RDD.persist(MEMORY_ONLY_SER)
.persist(MEMORY_AND_DISK)
.persist(MEMORY_AND_DISK_SER)
JVM

.persist(DISK_ONLY)
RDD.persist(MEMORY_ONLY_2)
.persist(MEMORY_AND_DISK_2)
.unpersist()
- If RDD fits in memory, choose MEMORYONLY
- If not, use MEMORYONLYSER w/ fast serialization library
- Don’t spill to disk unless functions that computed the datasets are very expensive or they filter a large amount of data. (recomputing may be as fast as reading from disk)
- Use replicated storage levels sparingly and only if you want fast fault recovery (maybe to serve requests from a web app)
Remember!

Intermediate data is automatically persisted during shuffle operations.
PySpark: stored objects will always be serialized with Pickle library, so it does not matter whether you choose a serialized level.
Default Memory Allocation in Executor JVM

- **Cached RDDs**: 60%
- **Shuffle memory**: 20%
- **User Programs (remainder)**: 20%

spark.shuffle.memoryFraction
spark.storage.memoryFraction
Spark uses memory for:

**RDD Storage:** when you call `.persist()` or `.cache()`. Spark will limit the amount of memory used when caching to a certain fraction of the JVM’s overall heap, set by `spark.storage.memoryFraction`.

**Shuffle and aggregation buffers:** When performing shuffle operations, Spark will create intermediate buffers for storing shuffle output data. These buffers are used to store intermediate results of aggregations in addition to buffering data that is going to be directly output as part of the shuffle.

**User code:** Spark executes arbitrary user code, so user functions can themselves require substantial memory. For instance, if a user application allocates large arrays or other objects, these will count for overall memory usage. User code has access to everything “left” in the JVM heap after the space for RDD storage and shuffle storage are allocated.
DETERMINING MEMORY CONSUMPTION

1. Create an RDD
2. Put it into cache
3. Look at SparkContext logs on the driver program or Spark UI

logs will tell you how much memory each partition is consuming, which you can aggregate to get the total size of the RDD

INFO BlockManagerMasterActor: Added rdd_0_1 in memory on mbk.local:50311 (size: 717.5 KB, free: 332.3 MB)
Serialization is used when:

Transferring data over the network

Spilling data to disk

Caching to memory serialized

Broadcasting variables
Java serialization vs. Kryo serialization

Java serialization:
- Uses Java’s ObjectOutputStream framework
- Works with any class you create that implements java.io.Serializable
- You can control the performance of serialization more closely by extending java.io.Externalizable
- Flexible, but quite slow
- Leads to large serialized formats for many classes

Kryo serialization:
- Recommended serialization for production apps
- Use Kryo version 2 for speedy serialization (10x) and more compactness
- Does not support all Serializable types
- Requires you to register the classes you’ll use in advance
- If set, will be used for serializing shuffle data between nodes and also serializing RDDs to disk

```java
conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer")
```
To register your own custom classes with Kryo, use the `registerKryoClasses` method:

```scala
val conf = new SparkConf().setMaster(...).setAppName(...) 
conf.registerKryoClasses(Seq(classOf[MyClass1], classOf[MyClass2])) 
val sc = new SparkContext(conf)
```

- If your objects are large, you may need to increase `spark.kryoserializer.buffer.mb` config property.

- The default is 2, but this value needs to be large enough to hold the largest object you will serialize.
TUNING FOR Spark

High churn

Low churn
Cost of GC is proportional to the # of Java objects

(so use an array of Ints instead of a LinkedList)

To measure GC impact:

-verbose:gc -XX:+PrintGCDetails -XX:+PrintGCTimeStamps
<table>
<thead>
<tr>
<th>GC Type</th>
<th>Flags (JVM)</th>
<th>Description</th>
<th>Use Cases</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Parallel GC</strong></td>
<td><code>-XX:+UseParallelGC</code></td>
<td>Uses multiple threads to do young gen GC</td>
<td>- Garage First is available starting Java 7</td>
</tr>
<tr>
<td></td>
<td><code>-XX:ParallelGCThreads=&lt;#&gt;</code></td>
<td></td>
<td>- Designed to be long term replacement for CMS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Will default to Serial on single core machines</td>
<td>- Is a parallel, concurrent and incrementally compacting low-pause GC</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Aka “throughput collector”</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Good for when a lot of work is needed and long pauses are acceptable</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Use cases: batch processing</td>
<td></td>
</tr>
<tr>
<td><strong>Parallel Old GC</strong></td>
<td><code>-XX:+UseParallelOldGC</code></td>
<td>Uses multiple threads to do both young gen and old gen GC</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Also a multithreading compacting collector</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- HotSpot does compaction only in old gen</td>
<td></td>
</tr>
<tr>
<td><strong>CMS GC</strong></td>
<td><code>-XX:+UseConcMarkSweepGC</code></td>
<td>Concurrent Mark Sweep aka “Concurrent low pause collector”</td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>-XX:ParallelCMSThreads=&lt;#&gt;</code></td>
<td>Tries to minimize pauses due to GC by doing most of the work concurrently with application threads</td>
<td></td>
</tr>
<tr>
<td><strong>G1 GC</strong></td>
<td><code>-XX:+UseG1GC</code></td>
<td>Uses same algorithm on young gen as parallel collector</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Use cases:</td>
<td></td>
</tr>
</tbody>
</table>
SCHEDULING PROCESS

**RDD Objects**
- Rdd1.join(rdd2)
  .groupBy(...)  
  .filter(...)
  - Build operator DAG

**DAG Scheduler**
- Split graph into stages of tasks
- Submit each stage as ready

**Task Scheduler**
- Launches individual tasks
- Retry failed or straggling tasks

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

**Agnostic to operators**
- Doesn't know about stages

**Stage failed**
Figure 1: Lineage graph for the third query in our example. Boxes represent RDDs and arrows represent transformations.

```python
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
errors.persist()
```
“One of the challenges in providing RDDs as an abstraction is choosing a representation for them that can track lineage across a wide range of transformations.”

“The most interesting question in designing this interface is how to represent dependencies between RDDs.”

“We found it both sufficient and useful to classify dependencies into two types:
• narrow dependencies, where each partition of the parent RDD is used by at most one partition of the child RDD
• wide dependencies, where multiple child partitions may depend on it.”
Examples of narrow and wide dependencies.

Each box is an RDD, with partitions shown as shaded rectangles.
STAGES

- **= RDD**
- **= cached partition**
- **= lost partition**

**Stage 1**
- GroupBy
  - A:
  - B:

**Stage 2**
- Map
- Filter
  - C:
  - D:
  - E:

**Stage 3**
- Join
  - F:
“This distinction is useful for two reasons:

1) Narrow dependencies allow for pipelined execution on one cluster node, which can compute all the parent partitions. For example, one can apply a map followed by a filter on an element-by-element basis.

In contrast, wide dependencies require data from all parent partitions to be available and to be shuffled across the nodes using a MapReduce-like operation.

2) Recovery after a node failure is more efficient with a narrow dependency, as only the lost parent partitions need to be recomputed, and they can be recomputed in parallel on different nodes. In contrast, in a lineage graph with wide dependencies, a single failed node might cause the loss of some partition from all the ancestors of an RDD, requiring a complete re-execution.”
To display the lineage of an RDD, Spark provides a `toDebugString` method:

```scala
scala> input.toDebugString

res85: String =
(2) data.text MappedRDD[292] at textFile at <console>:13
 | data.text HadoopRDD[291] at textFile at <console>:13

scala> counts.toDebugString
res84: String =
(2) ShuffledRDD[296] at reduceByKey at <console>:17
 +-(2) MappedRDD[295] at map at <console>:17
 | FilteredRDD[294] at filter at <console>:15
 | MappedRDD[293] at map at <console>:15
 | data.text MappedRDD[292] at textFile at <console>:13
 | data.text HadoopRDD[291] at textFile at <console>:13
```
How do you know if a shuffle will be called on a Transformation?

- repartition, join, cogroup, and any of the *By or *ByKey transformations can result in shuffles

- If you declare a numPartitions parameter, it’ll probably shuffle

- If a transformation constructs a shuffledRDD, it’ll probably shuffle

- combineByKey calls a shuffle (so do other transformations like groupByKey, which actually end up calling combineByKey)

Note that repartition just calls coalesce w/ True:

```scala
  coalesce(numPartitions, shuffle = true)
}
```
How do you know if a shuffle will be called on a Transformation?

Transformations that use “numPartitions” like distinct will probably shuffle:

```scala
    map(x => (x, null)).reduceByKey((x, y) => x, numPartitions).map(_._1)
```
- An extra parameter you can pass a k/v transformation to let Spark know that you will not be messing with the keys at all

- All operations that shuffle data over network will benefit from partitioning

- Operations that benefit from partitioning:
cogroup, groupWith, join, leftOuterJoin, rightOuterJoin, groupByKey, reduceByKey, combineByKey, lookup, . . .

How-to: Tune Your Apache Spark Jobs (Part 1)

by Sandy Ryza | March 09, 2015 | 0 no comments

Learn techniques for tuning your Apache Spark jobs for optimal efficiency.

(Editor’s note: Sandy presents on “Estimating Financial Risk with Spark” at Spark Summit East on March 18.)

When you write Apache Spark code and page through the public APIs, you come across words like transformation, action, and RDD. Understanding Spark at this level is vital for writing Spark programs. Similarly, when things start to fail, or when you venture into the web UI to try to understand why your application is taking so long, you’re confronted with a new vocabulary of words like job, stage, and task. Understanding Spark at this level is vital for writing good Spark programs, and of course by good, I mean fast. To write a Spark program that will execute efficiently, it is very, very helpful to understand Spark’s underlying execution model.

In this post, you’ll learn the basics of how Spark programs are actually executed on a cluster. Then, you’ll get some practical recommendations about what Spark’s execution model means for writing efficient programs.

How Spark Executes Your Program

A Spark application consists of a single driver process and a set of executor processes scattered across nodes on the cluster.

The driver is the process that is in charge of the high-level control flow of work that needs to be done. The executor processes are responsible for executing this work, in the form of tasks, as well as for storing any data that the user chooses to cache. Both the driver and the executors typically stick around for the entire time the application is running, although dynamic resource allocation changes that for the latter. A single executor has a number of slots for running tasks, and will run many concurrently throughout its lifetime. Deploying these processes on the cluster is up to the cluster manager in use (YARN, Mesos, or Spark Standalone), but the driver and executor themselves exist in every Spark application.
How many Stages will this code require?

```scala
sc.textFile("someFile.txt").map(mapFunc).flatMap(flatMapFunc).filter(filterFunc).count()
```

Source: Cloudera
How many Stages will this DAG require?
How many Stages will this DAG require?

Source: Cloudera
BROADCAST VARIABLES
&
ACCUMULATORS

databricks
USE CASES:

- **Broadcast variables** – Send a large read-only lookup table to all the nodes, or send a large feature vector in a ML algorithm to all nodes.

- **Accumulators** – count events that occur during job execution for debugging purposes. Example: How many lines of the input file were blank? Or how many corrupt records were in the input dataset?
Spark supports 2 types of shared variables:

- **Broadcast variables** – allows your program to efficiently send a large, read-only value to all the worker nodes for use in one or more Spark operations. Like sending a large, read-only lookup table to all the nodes.

- **Accumulators** – allows you to aggregate values from worker nodes back to the driver program. Can be used to count the # of errors seen in an RDD of lines spread across 100s of nodes. Only the driver can access the value of an accumulator, tasks cannot. For tasks, accumulators are write-only.
Broadcast variables let programmer keep a read-only variable cached on each machine rather than shipping a copy of it with tasks.

For example, to give every node a copy of a large input dataset efficiently.

Spark also attempts to distribute broadcast variables using efficient broadcast algorithms to reduce communication cost.
BROADCAST VARIABLES

Scala:

```scala
val broadcastVar = sc.broadcast(Array(1, 2, 3))
broadcastVar.value
```

Python:

```python
broadcastVar = sc.broadcast(list(range(1, 4)))
broadcastVar.value
```
ORCHESTRA IS THE DEFAULT BROADCAST MECHANISM IN APACHE SPARK

With its recent release, Apache Spark has promoted Correr—BitTorrent-like broadcast mechanism proposed in Orchestra (SIGCOMM'11)—to become its default broadcast mechanism. It's great to see our research see the light of the real-world! Many thanks to Reynolds and others for making it happen.

MLlib, the machine learning library of Spark, will enjoy the biggest boost from this change because of the broadcast-heavy nature of many machine learning algorithms.
Managing Data Transfers in Computer Clusters with Orchestra

Mosharaf Chowdhury, Matei Zaharia, Justin Ma, Michael I. Jordan, Ion Stoica
University of California, Berkeley
{mosharaf, matei, jma, jordan, istoica}@cs.berkeley.edu

ABSTRACT

Classical computing applications like MapReduce and Dryad transfer massive amounts of data between their computation stages. These transfers can have a significant impact on job performance, accounting for more than 50% of job completion times. Despite this impact, there has been relatively little work on optimizing the performance of these data transfers, with networking researchers traditionally focusing on per-flow traffic management. We address this limitation by proposing a global management architecture and a set of algorithms that (1) improve the transfer times of common communication patterns, such as broadcast and shuffle, and (2) allow scheduling policies at the transfer level, such as prioritizing a transfer over other transfers. Using a prototype implementation, we show that our solution improves broadcast completion times by up to 4.5x compared to the status quo in Hadoop. We also show that transfer-level scheduling can reduce the completion time of high-priority transfers by 1.7x.

Categories and Subject Descriptors
C.2 [Computer communication networks]: Distributed systems—Cloud computing

General Terms
Algorithms, design, performance

Keywords
Data-intensive applications, data transfer, datacenter networks

1 Introduction

The last decade has seen a rapid growth of cluster computing frameworks to analyze the increasing amounts of data collected and generated by applications like Google, Facebook, and Yahoo!. These clusters, operators aim to maximize the cluster utilization, while accommodating a variety of applications, workloads, and user requirements. To achieve these goals, several solutions have recently been proposed to reduce job completion times [11, 29, 43], accommodate interactive workloads [29, 43], and increase utilization [26, 29]. While in large part successful, these solutions have so far been focusing on scheduling and managing computation and storage resources, while mostly ignoring network resources.

However, managing and optimizing network activity is critical for improving job performance. Indeed, Hadoop traces from Facebook show that, on average, transferring data between successive stages accounts for 33% of the running times of jobs with reduce phases. Existing proposals for full-bahn bandwidth networks [24, 23, 24, 35] along with flow-level scheduling [10, 23] can improve network performance, but they do not account for collective behaviors of flows due to the lack of job-level semantics.

In this paper, we argue that to maximize job performance, we need to optimize at the level of transfers, instead of individual flows. We define a transfer as the set of all flows transporting data between two stages of a job. In frameworks like MapReduce and Dryad, a stage cannot complete (or sometimes even start) before it receives all the data from the previous stage. Thus, the job running time depends on the time it takes to complete the entire transfer, rather than the duration of individual flows comprising it. To this end, we focus on two transfer patterns that occur in virtually all cluster computing frameworks and are responsible for most of the network traffic in these clusters: shuffle and broadcast. Shuffle captures the many-to-many communication pattern between the map and reduce stages in MapReduce, and between Dryad’s stages. Broadcast captures the one-to-many communication pattern employed by iterative optimization algorithms [45] as well as fragment-replicate joins in Hadoop [6].

(a) Logistic Regression
(b) Collaborative Filtering

Figure 2: Per-iteration work flow diagrams for our motivating machine learning applications. The circle represents the master node and the boxes represent the set of worker nodes.

Figure 4: Orchestra architecture. An Inter-Transfer Controller (ITC) manages Transfer Controllers (TCs) for the active transfers. Each TC can choose among multiple transfer mechanisms depending on data size, number of nodes, and other factors. The ITC performs inter-transfer scheduling.
History: Old technique for broadcast

Uses HTTP

20 MB file
BITTorent Technique for Broadcast

Uses bittorrent

20 MB file
BITTORENT TECHNIQUE FOR BROADCAST

Source: Scott Martin
BITTORENT TECHNIQUE FOR BROADCAST
Accumulators are variables that can only be “added” to through an associative operation.

Used to implement counters and sums, efficiently in parallel.

Spark natively supports accumulators of numeric value types and standard mutable collections, and programmers can extend for new types.

Only the driver program can read an accumulator’s value, not the tasks.
Scala:

```scala
val accum = sc.accumulator(0)

sc.parallelize(Array(1, 2, 3, 4)).foreach(x => accum += x)

accum.value
```

Python:

```python
accum = sc.accumulator(0)
rdd = sc.parallelize([1, 2, 3, 4])
def f(x):
    global accum
    accum += x

rdd.foreach(f)

accum.value
```
PySpark at a Glance

Write Spark jobs in Python

Run interactive jobs in the shell

Supports C extensions
Spark Core Engine (Scala)

Standalone Scheduler

YARN

Mesos

Local

Java API

PySpark

41 files
8,100 loc
6,300 comments
PYSPARK ARCHITECTURE

Driver Machine

- Spark Context
- Py4j Socket
- Local Disk
- F(x)
- Driver JVM

Worker Machine

- Spark Context
- RDD
- MLlib, SQL, shuffle
- F(x)
- Executor JVM

Pipe

daemon.py
Data is stored as Pickled objects in an RDD[Array[Byte]]

RDD[Array[ ]] (100 KB – 1MB each picked object)
Choose Your Python Implementation

Driver Machine

Spark Context

Worker Machine

CPython
(default python)

pypy
• JIT, so faster
• less memory
• CFFI support

$ PYSPARK_DRIVER_PYTHON=pypy PYSPARK_PYTHON=pypy ./bin/pyspark

OR

$ PYSPARK_DRIVER_PYTHON=pypy PYSPARK_PYTHON=pypy ./bin/spark-submit wordcount.py
The performance speed up will depend on work load (from 20% to 3000%).

Here are some benchmarks:

<table>
<thead>
<tr>
<th>Job</th>
<th>CPython 2.7</th>
<th>PyPy 2.3.1</th>
<th>Speed up</th>
</tr>
</thead>
<tbody>
<tr>
<td>Word Count</td>
<td>41 s</td>
<td>15 s</td>
<td>2.7 x</td>
</tr>
<tr>
<td>Sort</td>
<td>46 s</td>
<td>44 s</td>
<td>1.05 x</td>
</tr>
<tr>
<td>Stats</td>
<td>174 s</td>
<td>3.6 s</td>
<td>48 x</td>
</tr>
</tbody>
</table>

Here is the code used for benchmark:

```python
rdd = sc.textFile("text")
def wordcount():
    rdd.flatMap(lambda x:x.split('/'))
    .map(lambda x:(x,1)).reduceByKey(lambda x,y:x+y).collectAsMap()
def sort():
    rdd.sortBy(lambda x:x, 1).count()
def stats():
    sc.parallelize(range(1024), 20).flatMap(lambda x: xrange(5024)).stats()
```

https://github.com/apache/spark/pull/2144
<table>
<thead>
<tr>
<th>spark.python.worker.memory</th>
<th>512m</th>
</tr>
</thead>
</table>

Amount of memory to use per python worker process during aggregation, in the same format as JVM memory strings (e.g. 512m, 2g). If the memory used during aggregation goes above this amount, it will spill the data into disks.
Spark sorted the same data **3X faster** using **10X fewer machines** than Hadoop MR in 2013.

All the sorting took place on disk (HDFS) without using Spark’s in-memory cache!

More info:

- [http://sortbenchmark.org](http://sortbenchmark.org)

---

### 100TB Daytona Sort Competition 2014

<table>
<thead>
<tr>
<th></th>
<th>Hadoop MR Record</th>
<th>Spark Record</th>
<th>Spark 1 PB</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data Size</strong></td>
<td>102.5 TB</td>
<td>100 TB</td>
<td>1000 TB</td>
</tr>
<tr>
<td><strong>Elapsed Time</strong></td>
<td>72 mins</td>
<td>23 mins</td>
<td>234 mins</td>
</tr>
<tr>
<td><strong># Nodes</strong></td>
<td>2100</td>
<td>206</td>
<td>190</td>
</tr>
<tr>
<td><strong># Cores</strong></td>
<td>50400 physical</td>
<td>6592 virtualized</td>
<td>6080 virtualized</td>
</tr>
<tr>
<td><strong>Cluster disk throughput</strong></td>
<td>3150 GB/s (est.)</td>
<td>618 GB/s</td>
<td>570 GB/s</td>
</tr>
<tr>
<td><strong>Sort Benchmark Daytona Rules</strong></td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td><strong>Network</strong></td>
<td>dedicated data center, 10Gbps</td>
<td>virtualized (EC2) 10Gbps network</td>
<td>virtualized (EC2) 10Gbps network</td>
</tr>
<tr>
<td><strong>Sort rate</strong></td>
<td>1.42 TB/min</td>
<td>4.27 TB/min</td>
<td>4.27 TB/min</td>
</tr>
<tr>
<td><strong>Sort rate/node</strong></td>
<td>0.67 GB/min</td>
<td>20.7 GB/min</td>
<td>22.5 GB/min</td>
</tr>
</tbody>
</table>

Work by Databricks engineers: Reynold Xin, Parviz Deyhim, Xiangrui Meng, Ali Ghodsi, Matei Zaharia
Startup Crunches 100 Terabytes of Data in a Record 23 Minutes

By Klint Finley 10.13.14 | 2:36 PM | PERMALINK

Databricks demolishes big data benchmark to prove Spark is fast on disk, too

by Derrick Herrs Oct. 10, 2014 - 1:49 PM PST
WHY SORTING?

- Stresses “shuffle” which underpins everything from SQL to Mllib
- Sorting is challenging b/c there is no reduction in data
- Sort 100 TB = 500 TB disk I/O and 200 TB network

Engineering Investment in Spark:
- Sort-based shuffle (SPARK-2045)
- Netty native network transport (SPARK-2468)
- External shuffle service (SPARK-3796)

Clever Application level Techniques:
- GC and cache friendly memory layout
- Pipelining
TECHNIQUE USED FOR 100 TB SORT

- Intel Xeon CPU E5 2670 @ 2.5 GHz w/ 32 cores
- 244 GB of RAM
- 8 x 800 GB SSD and RAID 0 setup formatted with /ext4
- ~9.5 Gbps (1.1 GBps) bandwidth between 2 random nodes

- Each record: 100 bytes (10 byte key & 90 byte value)
- OpenJDK 1.7
- HDFS 2.4.1 w/ short circuit local reads enabled
- Apache Spark 1.2.0
- Speculative Execution off
- Increased Locality Wait to infinite
- Compression turned off for input, output & network
- Used Unsafe to put all the data off-heap and managed it manually (i.e. never triggered the GC)

EC2: i2.8xlarge
(206 workers)
- 32 slots per machine
- 6,592 slots total
groupByKey = sortByKey
reduceByKey
spark.shuffle.spill=false

(Affects reducer side and keeps all the data in memory)
EXTERNAL SHUFFLE SERVICE

- Worker JVM serves files

- Node Manager serves files

- Must turn this on for dynamic allocation in YARN
OLD TECHNIQUE FOR SERVING MAP OUTPUT FILES

- Was slow because it had to copy the data 3 times
- Uses a technique called zero-copy
- Is a map-side optimization to serve data very quickly to requesting reducers
Hash Based Shuffle

- Entirely bounded by I/O reading from HDFS and writing out locally sorted files
- Mostly network bound

Notice that map has to keep 3 file handles open

Map()  Map()  Map()  Map()  Reduce()  Reduce()  Reduce()  Reduce()  TimSort

File = 5 blocks
SORT BASED SHUFFLE

(28,000 unique blocks)  
RF = 2

- Only one file handle open at a time

= 3.6 GB

250,000+ reducers!
SORT BASED SHUFFLE

- 5 waves of maps
- 5 waves of reduces

RF = 2

(28,000 unique blocks)

(MergeSort)

HADOOP HDFS

250,000+ reducers!

HADOOP HDFS

RF = 2

(RF = 2)
Sustaining 1.1GB/s/node during shuffle

- Actual final run
- Fully saturated the 10 Gbit link
[SPARK-3613] Record only average block size in MapStatus for large st...
<table>
<thead>
<tr>
<th>UserID</th>
<th>Name</th>
<th>Age</th>
<th>Location</th>
<th>Pet</th>
</tr>
</thead>
<tbody>
<tr>
<td>28492942</td>
<td>John Galt</td>
<td>32</td>
<td>New York</td>
<td>Sea Horse</td>
</tr>
<tr>
<td>95829324</td>
<td>Winston Smith</td>
<td>41</td>
<td>Oceania</td>
<td>Ant</td>
</tr>
<tr>
<td>92871761</td>
<td>Tom Sawyer</td>
<td>17</td>
<td>Mississippi</td>
<td>Raccoon</td>
</tr>
<tr>
<td>37584932</td>
<td>Carlos Hinojosa</td>
<td>33</td>
<td>Orlando</td>
<td>Cat</td>
</tr>
<tr>
<td>73648274</td>
<td>Luis Rodriguez</td>
<td>34</td>
<td>Orlando</td>
<td>Dogs</td>
</tr>
</tbody>
</table>
SchemaR DD

- RDD of Row objects, each representing a record
- Row objects = type + col. name of each
- Stores data very efficiently by taking advantage of the schema
- SchemaRDDs are also regular RDDs, so you can run transformations like map() or filter()
- Allows new operations, like running SQL on objects
Introducing DataFrames in Spark for Large Scale Data Science

February 17, 2015 | by Reynold Xin, Michael Armbrust and Davies Liu

Today, we are excited to announce a new DataFrame API designed to make big data processing even easier for a wider audience.

When we first open sourced Spark, we aimed to provide a simple API for distributed data processing in general-purpose programming languages (Java, Python, Scala). Spark enabled distributed data processing through functional transformations on distributed collections of data (RDDs). This was an incredibly powerful API: tasks that used to take thousands of lines of code to express could be reduced to just a few.

# sc is an existing SparkContext.
from pyspark.sql import SQLContext, Row
sqlContext = SQLContext(sc)

# Load a text file and convert each line to a Row.
lines = sc.textFile("examples/src/main/resources/people.txt")
parts = lines.map(lambda l: l.split(","))
people = parts.map(lambda p: Row(name=p[0], age=int(p[1])))

# Infer the schema, and register the SchemaRDD as a table.
schemaPeople = sqlContext.inferSchema(people)
schemaPeople.registerTempTable("people")

# SQL can be run over SchemaRDDS that have been registered as a table.
teens = sqlContext.sql("SELECT name FROM people WHERE age >= 13 AND age <= 19")

# The results of SQL queries are RDDs and support all the normal RDD operations.

for teenName in teens.collect():
    print(teenName)
# Import SQLContext and data types
from pyspark.sql import *

# sc is an existing SparkContext.
sqlContext = SQLContext(sc)

# Load a text file and convert each line to a tuple.
lines = sc.textFile("examples/src/main/resources/people.txt")
parts = lines.map(lambda l: l.split(","))
people = parts.map(lambda p: (p[0], p[1].strip()))

# The schema is encoded in a string.
schemaString = "name age"

fields = [StructField(field_name, StringType(), True) for field_name in schemaString.split()]
schema = StructType(fields)

# Apply the schema to the RDD.
schemaPeople = sqlContext.applySchema(people, schema)

# Register the SchemaRDD as a table.
schemaPeople.registerTempTable("people")

# SQL can be run over SchemaRDWs that have been registered as a table.
results = sqlContext.sql("SELECT name FROM people")

# The results of SQL queries are RDWs and support all the normal RDD operations.
names = results.map(lambda p: "Name: " + p.name)
for name in names.collect():
    print name
# sqlContext from the previous example is used in this example.

schemaPeople # The SchemaRDD from the previous example.

# SchemaRDDs can be saved as Parquet files, maintaining the schema information.
schemaPeople.saveAsParquetFile("people.parquet")

# Read in the Parquet file created above. Parquet files are self-describing so the schema is preserved.
# The result of loading a parquet file is also a SchemaRDD.
parquetFile = sqlContext.parquetFile("people.parquet")

# Parquet files can also be registered as tables and then used in SQL statements.
parquetFile.registerTempTable("parquetFile");
teenagers = sqlContext.sql("SELECT name FROM parquetFile WHERE age >= 13 AND age <= 19")
teenNames = teenagers.map(lambda p: "Name: " + p.name)

for teenName in teenNames.collect:
    print teenName
Configuration of Parquet can be done using the `setconf` method on `SQLContext` or by running `set` key-value commands using SQL.

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Default</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>spark.sql.parquet.binaryAsString</code></td>
<td>false</td>
<td>Some other Parquet-producing systems, in particular Impala and older versions of Spark SQL, do not differentiate between binary data and strings when writing out the Parquet schema. This flag tells Spark SQL to interpret binary data as a string to provide compatibility with these systems.</td>
</tr>
<tr>
<td><code>spark.sql.parquet.cacheMetadata</code></td>
<td>true</td>
<td>Turns on caching of Parquet schema metadata. Can speed up querying of static data.</td>
</tr>
<tr>
<td><code>spark.sql.parquet.compression.codec</code></td>
<td>gzip</td>
<td>Sets the compression codec use when writing Parquet files. Acceptable values include: uncompressed, snappy, gzip, lzo.</td>
</tr>
<tr>
<td><code>spark.sql.parquet.filterPushdown</code></td>
<td>false</td>
<td>Turn on Parquet filter pushdown optimization. This feature is turned off by default because of a known bug in Parquet 1.8.0rc3 (PARQUET-136). However, if your table doesn't contain any nullable string or binary columns, it's still safe to turn this feature on.</td>
</tr>
<tr>
<td><code>spark.sql.hive.convertMetastoreParquet</code></td>
<td>true</td>
<td>When set to false, Spark SQL will use the Hive SerDe for parquet tables instead of the built-in support.</td>
</tr>
</tbody>
</table>
An RDD of `Row` objects that has an associated schema. In addition to standard RDD functions, SchemaRRDs can be used in relational queries, as shown in the examples below.

Importing a SQLContext brings an implicit into scope that automatically converts a standard RDD whose elements are `scala.collection.immutable.Map[String, Any]` into a SchemaRDD. This conversion can also be done explicitly using the `createSchemaRDD` function on a SQLContext.

A SchemaRDD can also be created by loading data in from external sources. Examples are loading data from Parquet files by using the `parquetFile` method on `org.apache.spark.sql.SQLContext` and loading JSON datasets by using `jsonFile` and `jsonRDD` methods on `org.apache.spark.sql.SQLContext`.

### SQL Queries

A SchemaRDD can be registered as a table in the `SQLContext` that was used to create it. Once an RDD has been registered as a table, it can be used in the FROM clause of SQL statements.

```scala
val sc: SparkContext // An existing spark context.
val sqlContext = new SQLContext(sc)

// Importing the SQL context gives access to all the SQL functions and implicit conversions.
import sqlContext._

val rdd = sc.parallellize((1 to 100).map(i => Record(i, s"val_$i")))

// Any RDD containing case classes can be registered as a table. The schema of the table is
// automatically inferred using scala reflection.
rdd.registerTempTable("records")

val results: SchemaRDD = sql("SELECT * FROM records")

Language Integrated Queries
```
TwitterUtils.createStream(...) 
  .filter(_.getText.contains("Spark")) 
  .countByWindow(Seconds(5))
Complex algorithms can be expressed using:
- Spark transformations: *map()*\(), *reduce()*\(), *join()*\(), etc
- MLlib + GraphX
- SQL
One unified API
Tathagata Das (TD)

- Lead developer of Spark Streaming + Committer on Apache Spark core
- Helped re-write Spark Core internals in 2012 to make it 10x faster to support Streaming use cases
- On leave from UC Berkeley PhD program
- Ex: Intern @ Amazon, Intern @ Conviva, Research Assistant @ Microsoft Research India
- 1 guy; does not scale

- Scales to 100s of nodes
- Batch sizes as small at half a second
- Processing latency as low as 1 second
- Exactly-once semantics no matter what fails
USE CASES

Page views → Kafka for buffering → Spark for processing

(live statistics)
USE CASES (Anomaly Detection)

Join 2 live data sources

Spark STREAMING

Smart meter readings

Live weather data
Batch interval = 5 seconds

Input DStream

T = 5
Block #1  Block #2  Block #3

RDD @ T=5

T = +5
Block #1  Block #2  Block #3

RDD @ T=+5

One RDD is created every 5 seconds

DSTREAM
(Discretized Stream)
TRANSFORMING DSTREAMS

- linesDStream
- wordsRDD
-flatMap()
- Part. #1
- Part. #2
- Part. #3
- 5 sec
- Materialize!
from pyspark import SparkContext
from pyspark.streaming import StreamingContext

# Create a local StreamingContext with two working thread and batch interval of 1 second
sc = SparkContext("local[2]", "NetworkWordCount")
ssc = StreamingContext(sc, 5)

# Create a DStream that will connect to hostname:port, like localhost:9999
linesDStream = ssc.socketTextStream("localhost", 9999)

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

# Count each word in each batch
pairsDStream = wordsDStream.map(lambda word: (word, 1))
wordCountsDStream = pairsDStream.reduceByKey(lambda x, y: x + y)

# Print the first ten elements of each RDD generated in this DStream to the console
wordCountsDStream.pprint()

ssc.start()  # Start the computation
ssc.awaitTermination()  # Wait for the computation to terminate
$ nc -lk 9999

hello world

$ ./network_wordcount.py localhost 9999

...  
--------------------------
--------------------------
(hello, 2)  
(world, 1)
Batch interval = 600 ms
Batch interval = 600 ms
Ex RDD, P1

Driver

RDD, P1

block, P1

SSD

OS Disk

Internal Threads

Ex RDD, P2

block, P2

SSD

OS Disk

Internal Threads

Ex RDD, P3

block, P3

SSD

OS Disk

Internal Threads

Batch interval = 600 ms

200 ms later

block, P2

Internal Threads

Ex

block, P2

Internal Threads

Ex

block, P2

Internal Threads

Ex

block, P2

Internal Threads
Batch interval = 600 ms

W

Ex

RDD, P1
RDD, P2
RDD, P1
RDD, P2
RDD, P1
RDD, P3

Internal Threads

Driver

OS Disk SSD SSD

Ex

RDD, P1
RDD, P2
RDD, P1
RDD, P3

Internal Threads

W

Ex

RDD, P2

Internal Threads

W

OS Disk SSD SSD

Batch interval = 600 ms
Batch interval = 600 ms
Streaming

Time since start: 27 minutes 20 seconds
Network receivers: 1
Batch interval: 1 second
Processed batches: 1641
Waiting batches: 0

Statistics over last 100 processed batches

Receiver Statistics

<table>
<thead>
<tr>
<th>Receiver</th>
<th>Status</th>
<th>Location</th>
<th>Records in last batch</th>
<th>Minimum rate</th>
<th>Median rate</th>
<th>Maximum rate</th>
<th>Last Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>TwitterReceiver 0</td>
<td>ACTIVE</td>
<td>localhost</td>
<td>39</td>
<td>0</td>
<td>61</td>
<td>151</td>
<td>-</td>
</tr>
</tbody>
</table>

Batch Processing Statistics

<table>
<thead>
<tr>
<th>Metric</th>
<th>Last batch</th>
<th>Minimum</th>
<th>25th percentile</th>
<th>Median</th>
<th>75th percentile</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processing Time</td>
<td>31 ms</td>
<td>5 ms</td>
<td>39 ms</td>
<td>56 ms</td>
<td>457 ms</td>
<td>2 seconds 269 ms</td>
</tr>
<tr>
<td>Scheduling Delay</td>
<td>0 ms</td>
<td>0 ms</td>
<td>0 ms</td>
<td>0 ms</td>
<td>1 ms</td>
<td>803 ms</td>
</tr>
<tr>
<td>Total Delay</td>
<td>31 ms</td>
<td>21 ms</td>
<td>40 ms</td>
<td>57 ms</td>
<td>490 ms</td>
<td>2 seconds 289 ms</td>
</tr>
</tbody>
</table>
Batch interval = 600 ms

2 input DStreams
Batch interval = 600 ms
Batch interval = 600 ms

Materialize!
Batch interval = 600 ms

Union!
BASIC
- File systems
- Socket Connections
- Akka Actors

Sources directly available in `StreamingContext` API

ADVANCED
- Kafka
- Flume
- Twitter

Requires linking against extra dependencies

CUSTOM
- Anywhere

Requires implementing user-defined receiver
Spark Streaming + Flume Integration Guide

Apache Flume is a distributed, reliable, and available service for efficiently collecting, aggregating, and moving large amounts of log data. Here we explain how to configure Flume and Spark Streaming to receive data from Flume. There are two approaches to this.

Approach 1: Flume-style Push-based Approach

Flume is designed to push data between Flume agents. In this approach, Spark Streaming essentially sets up a receiver that acts as an Avro agent for Flume, to which Flume can push the data. Here are the configuration steps.

General Requirements

Choose a machine in your cluster such that

- When your Flume + Spark Streaming application is launched, one of the Spark workers must run on that machine.
- Flume can be configured to push data to a port on that machine.

Due to the push model, the streaming application needs to be up, with the receiver scheduled and listening on the chosen port, for Flume to be able push data.

Configuring Flume

Configure Flume agent to send data to an Avro sink by having the following in the configuration file.

agent.sinks = avrosink
Spark Streaming + Kafka Integration Guide

Apache Kafka is publish-subscribe messaging rethought as a distributed, partitioned, replicated commit log service. Here we explain how to configure Spark Streaming to receive data from Kafka.

1. Linking: In your SBT/Maven project definition, link your streaming application against the following artifact (see Linking section in the main programming guide for further information).

   ```
   groupId = org.apache.spark
   artifactId = spark-streaming-kafka_2.10
   version = 1.2.0
   ```

2. Programming: In the streaming application code, import `kafkastream` and create input DStream as follows.

   ```scala
   import org.apache.spark.streaming.kafka._
   ```

   ```java
   ```
TRANSFORMATIONS ON DSTREAMS

- map($f(x)$)
- flatMap($f(x)$)
- filter($f(x)$)
- repartition(numPartitions)
- union(otherStream)
- reduceAByKey($f_{(x)}$, numTasks)
- join(otherStream, numTasks)
- transform($f_{(x)}$)
- cogroup(otherStream, numTasks)
- count()
- countByValue()
updateStateByKey( \( f(x) \) )

allows you to maintain arbitrary state while continuously updating it with new information.

To use:

1) Define the state
   (an arbitrary data type)

2) Define the state update function
   (specify with a function how to update the state using the previous state and new values from the input stream)

* Requires a checkpoint directory to be configured

To maintain a running count of each word seen in a text data stream (here running count is an integer type of state):

```python
def updateFunction(newValues, runningCount):
    if runningCount is None:
        runningCount = 0
    return sum(newValues, runningCount)  # add the new values with the previous running count
                                      # to get the new count
```

runningCounts = pairs.updateStateByKey(updateFunction)
TRANSFORMATIONS ON DSTREAMS

For example:

- Functionality to join every batch in a data stream with another dataset is not directly exposed in the DStream API.

- If you want to do real-time data cleaning by joining the input data stream with pre-computed spam information and then filtering based on it.

```python
spamInfoRDD = sc.pickleFile(...)  # RDD containing spam information

# join data stream with spam information to do data cleaning
cleanedDStream = wordCounts.transform(lambda rdd:
    rdd.join(spamInfoRDD).filter(...))
```

OR

MLlib  GraphX
Window Length: *3 time units
Sliding Interval: *2 time units
* Both of these must be multiples of the batch interval of the source DStream

WindowedWordCounts = pairs.reduceByKeyAndWindow(lambda x, y: x + y, lambda x, y: x - y, 30, 10)
COMMON WINDOW OPERATIONS

window(windowLength, slideInterval)

countByValueAndWindow(windowLength, slideInterval, [numTasks])

countByWindow(windowLength, slideInterval)

reduceByWindow(f, windowLength, slideInterval)

reduceByKeyAndWindow(f, windowLength, slideInterval, [numTasks])

reduceByKeyAndWindow(f, windowLength, slideInterval, [numTasks])
OUTPUT OPERATIONS ON DSTREAMS

print()

saveAsTextFile((prefix, suffix))

foreachRDD( $f$ )

saveAsObjectFiles((prefix, suffix))

saveAsHadoopFiles((prefix, suffix))
Next slide is only for on-site students...
Post DevOps Spark Class Survey

High quality slides, labs and instructors are very important to us at Databricks. Please take a few minutes to complete this anonymous survey. The results will be shared with the entire class in 5 minutes via a now link the instructor will give you.

* Required

How would you rate today's class overall? *

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class was really bad</td>
<td>Class was awesome</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

How would you rate the instructor's effectiveness at teaching? *

Was the instructor easy to understand? Was he knowledgeable about Spark?

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>He was horrible</td>
<td>He was great</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

How did you find the difficulty level and pacing of the class? *

You can pick one or more below... or just type in a free-form response...

- Class moved a bit too fast
- It was just the right pace