TUTORIAL: BIG DATA ANALYTICS USING APACHE SPARK

Sugimiyanto Suma
Yasir Arfat

Supervisor:
Prof. Rashid Mehmood
Outline

- Big Data
- Big Data Analytics
- Problem
- Basics of Apache Spark
- Practice basic examples of Spark
- Building and Running the Spark Applications
- Spark's Libraries
- Practice Data Analytics using applications.
- Performance tuning of Apache Spark: Cluster Resource Utilization
- Advance features of Apache Spark: When and How to use
Tutorial objective

- Giving Spark’s concepts
- Build and Run Spark Application
- Utilization of Apache Spark Standalone Cluster
- Analytics using Apache Spark with applications
‘Big data’ is defined by IBM as any data that cannot be captured, managed and/or processed using traditional data management components and techniques.

Source: https://www.slideshare.net/AndersQuitzaulbm/big-data-analyticsin-energy-utilities
Big data analytic

- Big data analytics examines large amounts of data to uncover hidden patterns, correlations and other insights.

- With today’s technology, it’s possible to analyze our data and get answers from it almost immediately – an effort that's slower and less efficient with more traditional business intelligence solutions.

Source: sas.com
Cost reduction. Big data technologies such as Hadoop and cloud-based analytics bring significant cost advantages when it comes to storing large amounts of data – plus they can identify more efficient ways of doing business.

Faster, better decision making. With the speed of Hadoop and in-memory analytics, combined with the ability to analyze new sources of data, businesses are able to analyze information immediately – and make decisions based on what they’ve learned.

New products and services. With the ability to gauge customer needs and satisfaction through analytics comes the power to give customers what they want. Davenport points out that with big data analytics, more companies are creating new products to meet customers’ needs.

Problem

- Data growing faster than processing speeds.
- The only solution is to parallelize the data on **large clusters**.
  - Wide use in both enterprises and web industry.

Problem

- Data growing faster than processing speeds.
- The only solution is to parallelize the data on large clusters.
  - Wide use in both enterprises and web industry.

How do we program these things?

Apache Spark
History of Spark

- Started in 2009 as a research project at UC Berkeley
- Hadoop was too inefficient for iterative type algorithms and interactive analysis that the researchers required – Spark was initially the result of creating a platform to address those needs
- At inception, it was already providing speed gains over Hadoop on Map Reduce jobs
- Open sourced in 2010
- Spark Streaming was incorporated as a core component in 2011
- In 2013, Spark was transferred to the Apache Software foundation
- Spark 1.0 released in 2014, where it became the most actively developed and fastest growing Apache project in Apache’s history
- July 26th, 2016 Spark 2.0 released with major improvements in every area over the 1.x branches

Source: https://spark.apache.org/docs/latest/
Spark Community

- Most active open source community in big data 200+ developers,
- 50+ companies contributing

Source: https://training.databricks.com/
What is spark?

- Apache Spark is a fast and general-purpose cluster computing system.
  - It provides high-level APIs in Java, Scala, Python and R, and an optimized engine that supports general execution graphs.
  - It also supports a rich set of higher-level tools including Spark SQL for SQL and structured data processing, MLlib for machine learning, GraphX for graph processing, and Spark Streaming. It eases developer to build an application to process and analyze big data in parallel on top of a cluster.
  - It is an alternative of Hadoop Map Reduce to process big data in parallel.

Source: https://spark.apache.org/docs/latest/
Sequential VS Parallel

We utilize more cores/processors/executors

Why spark?

- **Speed**: Run programs up to 100x faster than Hadoop MapReduce in memory, or 10x faster on disk.
- **Ease of Use**: Supports different languages for developing applications using Spark.
- **Generality**: Combine SQL, streaming, and complex analytics into one platform.
- **Runs Everywhere**: Spark runs on Hadoop, Mesos, standalone, or in the cloud.

Dealing with huge data and parallelize it (speed up)

**Source**: http://edureka.co

Tutorial: Big data analytics using Apache Spark

- Sugimiyanto Suma et al.
Why spark? (cont.)

Hadoop execution flow

Spark execution flow

Source: http://www.wiziq.com/blog/hype-around-apache-spark/
Spark Stack and what you can do

- **Spark SQL**
  - For SQL and unstructured data processing

- **MLlib**
  - Machine Learning Algorithms

- **GraphX**
  - Graph Processing

- **Spark Streaming**
  - stream processing of live data streams

Source: http://spark.apache.org

Computation in parallel
Execution Flow of Spark

Source: http://spark.apache.org/docs/latest/cluster-overview.html
Terminology

- **Spark Context**
  - Represents the connection to a Spark cluster, and can be used to create RDDs, accumulators and broadcast variables on the cluster.

- **Driver Program**
  - The process running the main() function of the application and creating the SparkContext

- **Cluster Manager**
  - An external service to manage resources on the cluster (standalone manager, YARN, Apache Mesos)

- **Deploy Mode**
  - Distinguishes where the driver process runs.
    - "cluster" mode, the framework launches the driver inside of the cluster.
    - "client" mode, the submitter launches the driver outside of the cluster.

Source: [http://spark.apache.org/docs/](http://spark.apache.org/docs/)
Terminology (cont.)

- **Worker Node**: Nodes that run the application program in cluster
- **Executor**
  - Process launched on a worker node, which runs the Tasks
  - Keep data in memory or disk storage
- **Task**: A unit of work that will be sent to executor
- **Job**
  - Consists multiple tasks
  - Created based on an Action
- **Stage**: Each Job is divided into smaller set of tasks called Stages that is sequential and depend on each other

Source: [http://spark.apache.org/docs/](http://spark.apache.org/docs/)
Resilient Distributed Dataset (RDD)

- **RDD** stands for resilient distributed dataset
- **Resilient** - if data is lost, data can be recreated
- **Distributed** - stored in nodes among the cluster
- **Dataset** - initial data comes from a file or can be created programmatically
- Partitioned collection of records
- Spread across the cluster
- Read-only
- Caching dataset in memory
  - different storage levels available

Source: [http://spark.apache.org/docs/](http://spark.apache.org/docs/)
Resilient Distributed Dataset (Cont.)

- Resilient Distributed Dataset (RDD) is a basic Abstraction in Spark
- **Immutable**, Partitioned collection of elements that can be operated in parallel
- Simply, RDD is a data type which can be parallelized.
- **RDD main characteristics:**
  - A list of partitions
  - A function for computing each split
  - A list of dependencies on other RDDs
  - Optionally, a partitioner for key-value RDDs (e.g. to say that the RDD is hash-partitioned)
  - Optionally, a list of preferred locations to compute each split on (e.g. block locations for an HDFS file)
- Custom RDD can also be implemented (by overriding functions)

Source: [http://spark.apache.org/docs/](http://spark.apache.org/docs/)
How basic operations works

RDD computations model

3 main steps:
- Create RDD
- Transformation
- Actions

Type of Operations

- **Transformations**
  - map
  - filter
  - flatMap
  - union, etc.

- **Actions**
  - collect
  - count
  - take
  - foreach, etc.
Working with RDDs

```
linesWithSpark = textFile.filter(lambda line: "Spark" in line)
```

```
textFile = sc.textFile("SomeFile.txt")

linesWithSpark.count()  
74

linesWithSpark.first()  
# Apache Spark
```

Source: http://10minbasics.com/what-is-apache-spark/
Creating RDDs

- Two ways creating an RDD
  - Initialize a collection of values
    ```scala
    val rdd = sc.parallelize(Seq(1,2,3,4))
    ```
  - Load data file(s) from fileSystem, HDFS, etc.
    ```scala
    val rdd = sc.textFile("file://anyText.txt")
    ```
How the parallelization works

➢ Suppose we are working with a standalone node with 4 cores.
➢ We want to increment by 1 each element in a list/array.
➢ We are running the job in sequential mode.

```scala
val data = Seq(1, 2, 3, 4)
var res[] = {}
for(i <- 1 to 4){
  res[i] = data[i] + 1
}

res = (2, 3, 4, 5) Execution time: 4 millisecond
```
Suppose we are working with a standalone node with 4 cores.
- We want to increment by 1 each element in a list/array.
- We are going to parallelize the data among 2 cores.

```scala
val rdd = sc.parallelize(Seq(1, 2, 3, 4))
val rddAdd = rdd.map(i => i + 1)
```

```
1 2 3 4
+1 +1 +1 +1

res: rddAdd = (2, 3, 4, 5)   Execution time : 2 millisecond
```
How the parallelization works (cont.)

Suppose we are working with a standalone node with 4 cores.
We are going to parallelize the data with core-based.

```scala
val rdd = sc.parallelize(Seq(1,2,3,4))
val rddAdd = rdd.map(i => i+1)
```

```
1
+1

2
+1

3
+1

4
+1
```

```
res: rddAdd = (2,3,4,5)  Execution time : 1 millisecond
```
Spark Web UI

Details for Stage 2 (Attempt 0)

Total Time Across All Tasks: 6 ms
Locality Level Summary: Process local: 1

- DAG Visualization
- Show Additional Metrics
- Event Timeline

Enable zooming

<table>
<thead>
<tr>
<th>Task Type</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scheduler Delay</td>
<td>800</td>
</tr>
<tr>
<td>Task Deserialization Time</td>
<td>850</td>
</tr>
<tr>
<td>Shuffle Read Time</td>
<td>900</td>
</tr>
<tr>
<td>Executor Computing Time</td>
<td>950</td>
</tr>
<tr>
<td>Getting Result Time</td>
<td>000</td>
</tr>
</tbody>
</table>

Stage details event timeline showing the timeline of task execution. Notice that only task is executed sequentially. There is only one task at a time.
Spark Web UI (cont.)

Details for Stage 0 (Attempt 0)

- Total Time Across All Tasks: 0 ms
- Locality Level Summary: Process local: 2

- DAG Visualization
- Show Additional Metrics
- Event Timeline
- Enable zooming

Stage details event timeline showing the timeline of task execution. Notice that there are two tasks executed in parallel at a time.
Spark Web UI (cont.)

Stage details event timeline over four cores, parallelism has increased when we use more cores. It shows that it is running four tasks in parallel at a time.
Cluster Deployment

- Local Node
- Standalone Deploy Mode
- Apache Mesos
- Hadoop YARN
- Amazon EC2
## Cluster Deployment

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local</td>
<td>Run Spark locally with one worker thread (no parallelism)</td>
</tr>
<tr>
<td>Local[K]</td>
<td>Run Spark locally with K worker threads (ideally, set this to the number of cores on your machine).</td>
</tr>
<tr>
<td>Local[*]</td>
<td>Run Spark locally with as many worker threads as logical cores on your machine.</td>
</tr>
<tr>
<td>Spark://HOST:PORT</td>
<td>Connect to a Spark standalone cluster; PORT depends on config (7077 by default)</td>
</tr>
<tr>
<td>Mesos://HOST:PORT</td>
<td>Connect to a Mesos cluster; PORT depends on config (5050 by default)</td>
</tr>
<tr>
<td>Class</td>
<td>The entry point for your application (e.g. org.apache.spark.examples.SparkPi)</td>
</tr>
<tr>
<td>Master</td>
<td>The master URL for the cluster (e.g. spark://23.195.26.187:7077)</td>
</tr>
<tr>
<td>Deploy-mode</td>
<td>Whether to deploy your driver on the worker nodes (cluster) or locally as an external client (client) (default: client)</td>
</tr>
<tr>
<td>Conf</td>
<td>Arbitrary Spark configuration property in key=value format. For values that contain spaces wrap “key=value” in quotes (as shown)</td>
</tr>
<tr>
<td>Application-jar</td>
<td>Path to a bundled jar including your application and all dependencies. The URL must be globally visible inside of your cluster, for instance, an hdfs:// path or a file:// path that is present on all nodes.</td>
</tr>
<tr>
<td>Application arguments</td>
<td>Arguments passed to the main method of your main class, if any</td>
</tr>
</tbody>
</table>

Source: spark.apache.org/docs/latest/submitting-applications.html
Two ways working with spark

- Interactively (Spark-shell)
- Standalone application (Spark-submit)
Two ways working with spark (cont.)

- Interactively (spark-shell)

```bash
hduser@sugi-elitelOne:~$ spark-shell
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel).
17/04/15 23:37:42 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
17/04/15 23:37:48 WARN SparkContext: Use an existing SparkContext, some configuration may not take effect.
Spark context available as 'sc' (master = spark://hadoop-master:7077, app id = application-2017041515233744-0046).
Spark session available as 'spark'.
Welcome to

```
```
version 2.0.1

Using Scala version 2.11.8 (OpenJDK 64-Bit Server VM, Java 1.8.0_121)
Type in expressions to have them evaluated.
Type :help for more information.

scala> ```
Two ways working with spark (cont.)

- Standalone application (Spark-submit)

```
spark-submit \
   --class company.division.yourClass \ 
   --master spark://<host>:7077 \ 
   --name "countingCows" 
CountingCows-1.0.jar
```

```
spark-submit --class company.division.yourClass --master spark://<host>:7077 --name "countingCows" CountingCows-1.0.jar
```
Simple example: Using Interactive Mode

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

val conf = new SparkConf().setAppName("ToyApp")
val sc = new SparkContext(conf)

val rdd = sc.parallelize(Seq(10,20,30,40))
val rddAdd = rdd.map(i => i+1)
rddAdd.collect()
sc.stop()
```

Create an RDD  
Transformation  
Action
Building And Running a filter application

```scala
cd /home/hduser/Tutorial/SimpleApp/Filter
1. Create main class directory : mkdir -p src/main/scala
2. Create script file : vim src/main/scala/Filter.scala
2.1 Use this code :
   import org.apache.spark._
   object Filter{
     def main(args: Array[String]){  
       val conf = new SparkConf().setAppName("Filter")
       val sc = new SparkContext(conf)
       val x = sc.parallelize(1 to 100000, 2)
       val y = x.filter(e => e%2==1)
       y.foreach(println(_))
       println("Done.")
       sc.stop()
     }
   }
```
Building And Running the filter application

3. Create an sbt file: `vim build.sbt`
4. Use this application properties:
   ```scala
   name := "Filter"
   version := "1.0"
   scalaVersion := "2.11.8"
   libraryDependencies += "org.apache.spark" %% "spark-core" % "2.2.0"
   ``
5. Compile and generate *.jar file:
   ```bash
   sbt compile
   sbt package
   ```
6. Spark-submit:
   ```bash
   spark-submit --master local --executor-memory 5g --driver-memory 5g target/scala-2.11/filter_2.11-1.0.jar
   ```
Word count example

- Working with `spark-submit` with a million records
  - Case: count word appearance in whole document (commonly used to generate burst words)
  - Submitting the application to single core (sequential)
  - We will see the speedup when submitting to multi-cores (parallel)
Word count example (cont.)

```
import org.apache.spark._

object Toy{
  def main(args: Array[String]){
    val conf = new SparkConf().setAppName("ToyApp")
    val sc = new SparkContext(conf)

    val rdd = sc.textFile("file:///home/hduser/Tutorial/SimpleApp/WordCount big.txt")

    val start = System.nanoTime
    val rddLower = rdd.map(i => i.toLowerCase)
        .replaceAll("[^a-zA-Z0-9_ ]","")
        .flatMap(i => i.split(" "))
        .map(word => (word,1)).reduceByKey(_ + _)

    rddLower.coalesce(1).saveAsTextFile("file:///home/hduser/Tutorial/big_clean.txt")
    val duration = (System.nanoTime - start) / 1e9d
    println("Done.")
    println("Execution time: "+duration)
    sc.stop()
  }
}
```

Create an RDD

Transformation

Action
Building Word Count application with SBT

cd /home/hduser/Tutorial/SimpleApp/WordCount

1. Create main class directory : mkdir -p src/main/scala

2. Create script file : vim src/main/scala/WordCount.scala

3. Use this code :

```scala
import org.apache.spark._
object WordCount{
  def main(args: Array[String]){
    val conf = new SparkConf().setAppName("Word Count")
    val sc = new SparkContext(conf)
    val rdd = sc.textFile("file:///home/hduser/Tutorial/SimpleApp/WordCount/big.txt")
    val start = System.nanoTime
    val rddLower = rdd.map(i => i.toLowerCase().replaceAll("[^a-zA-Z0-9_\ ]",""))
      .flatMap(i => i.split(" ")).map(word => (word,1)).reduceByKey(_ + _)
    rddLower.coalesce(1).saveAsTextFile("file:///home/hduser/Tutorial/SimpleApp/WordCount/Result/wordCount")
    val duration = (System.nanoTime - start) / 1e9d
    println("Done.")
    println("Execution time: "+duration)
    sc.stop()
  }
}
```

4. Building Word Count application with SBT
Building And Running the Word Count application

3. Create an sbt file:
   
   `vim build.sbt`

4. Use this application properties:
   
   ```
   name := "WordCount"
   version := "1.0"
   scalaVersion := "2.11.8"
   libraryDependencies += "org.apache.spark" %% "spark-core" % "2.2.0"
   ```

5. Compile and generate *.jar file:
   
   ```
   sbt compile
   sbt package
   ```

6. Submit to one core:
   
   ```
   spark-submit --master local target/scala-2.11/wordcount_2.11-1.0.jar
   ```

7. Submit to all cores:
   
   Remove the previous result:
   
   ```
   rm -r /home/hduser/Tutorial/SimpleApp/WordCount/Result/wordcount
   ```

   ```
   spark-submit --master local[*] target/scala-2.11/wordcount_2.11-1.0.jar
   ```
Submitting application with spark-submit

```
hduser@sugi-ideapad:~/Tutorial/SimpleApp
hduser@sugi-ideapad:~/Tutorial/SimpleApp$ spark-submit -master local target/scala-2.11/toy_2.11-1.0.jar
17/11/20 16:16:13 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
17/11/20 16:16:13 WARN Utils: Your hostname, sugi-ideapad resolves to a loopback address: 127.0.0.1; using 10.6.73.100 instead (on interface enp3s0)
17/11/20 16:16:13 WARN Utils: Set SPARK_LOCAL_IP if you need to bind to another address
Done.
Execution time: 15.815824361
```

Tutorial: Big data analytics using Apache Spark
- Sugimiyanto Suma et al.
Submitting application with spark-submit (cont.)

```bash
hduser@sugi-idepad:~/Tutorial/SimpleApp$ spark-submit --master local[*] target/scala-2.11/toy_2.11-1.0.jar
17/11/20 16:18:28 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
17/11/20 16:18:28 WARN Utils: Your hostname, sugi-idepad resolves to a loopback address: 127.0.0.1; using 10.6.73.100 instead (on interface enp3s0)
17/11/20 16:18:28 WARN Utils: Set SPARK_LOCAL_IP if you need to bind to another address
Done.
Execution time: 9.240921633
hduser@sugi-idepad:~/Tutorial/SimpleApp$ 
```
Submitting application with spark-submit (cont.)
Questions?
Break

30 minutes
Welcome Back

TUTORIAL: BIG DATA ANALYTICS USING APACHE SPARK

Sugimiyanto Suma
Yasir Arfat

Supervisor:
Prof. Rashid Mehmood
Outline

- Big Data
- Big Data Analytics
- Problem
- Basics of Apache Spark
- Practice basic examples of Spark
- Building and Running the Spark Applications
- Spark’s Libraries
- Practice Data Analytics using applications.
- Performance tuning of Apache Spark: Cluster Resource Utilization
- Advance features of Apache Spark: When and How to use
Big Data Analytics Component

(Gandomi & Haider 2015)
Big Data Analytics Using Apache Spark

Big data Processes

Data Management
- Acquisition and Recording
- Extraction, Cleaning and Annotation
- Integration, Aggregation and Representation

Analytics
- Modeling and Analysis
- Interpretation

Tutorial: Big data analytics using Apache Spark
- Sugimiyanto Suma et al.
Analysis Techniques

- Natural Language Processing (NLP)
- Statistical Analysis
- Machine Learning
- ...
In 1959, Arthur Samuel, a pioneer in the field of computer gaming and artificial intelligence defined machine learning as:

"the field of study that gives computers the ability to learn without being explicitly programmed"

- Finding a pattern from the given dataset (training)
- Estimate the value of unseen data
Machine Learning (cont.)

- A general machine learning pipeline

(Rajdeep et al. 2017)
Unsupervised learning: A model does not require labelled data. No labels of outputs are given to the learning system. It finds structure on its own from the inputs given to. E.g. clustering, dimensionality reduction, and some forms of feature extraction, such as text processing.

Supervised learning: These types of models use labelled data to learn. The system is presented with inputs and desired outputs by a human and the goal is to learn a model to map inputs to outputs. E.g. Recommendation engines, regression, and classification.
**Machine Learning (cont.)**

- **Classification** generally refers to classifying things into distinct categories or classes.
- We wish to assign classes based on a set of features.
- The features might represent variables related to an item or object, and event or context, or some combination of these.
  - **Binary classification** (1 or 0). E.g. email spam classifier (spam or non-spam).
  - Multiclass classification (0 … n). E.g. sentiment analysis (positive, negative, neutral).
Spark built-in libraries
Spark built-in libraries

- MLlib
- Spark SQL
Machine Learning Library (MLlib)

- Apache Spark's scalable machine learning library.
- Its goal is to make practical machine learning scalable and easy.

Ease of use

data = spark.read.format("libsvm")
.load("hdfs://...")

model = KMeans(k=10).fit(data)

Calling MLlib in Python

Performance (high-quality algorithms, 100x faster than MapReduce)

Logoic regression in Hadoop and Spark

Source: spark.apache.org/mllib/
MLlib (cont.)

- MLlib contains many algorithms and utilities.
- ML algorithms include:
  - Classification: logistic regression, naive Bayes,...
  - Regression: generalized linear regression, survival regression,...
  - Clustering: K-means, Gaussian mixtures (GMMs),...
- ML workflow utilities include:
  - Feature transformations: standardization, normalization, hashing,...
  - Model evaluation and hyper-parameter tuning
- Other utilities include:
  - Distributed linear algebra: SVD, PCA,...
  - Statistics: summary statistics, hypothesis testing,...

For the complete list visit: https://spark.apache.org/mllib/

Source: spark.apache.org/mllib/
Spark built-in libraries

- MLlib
- Spark SQL
Spark SQL

- Apache Spark's module for working with structured data
- Integrated
  - Seamlessly mix SQL queries with Spark programs

```python
context = HiveContext(sc)
results = context.sql(
    "SELECT * FROM people")
names = results.map(lambda p: p.name)
```

Apply functions to results of SQL queries.

Source: spark.apache.org/sql/
Spark SQL (cont.)

- **Performance & Scalability**
  - Spark SQL includes a cost-based optimizer, columnar storage and code generation to make queries fast.
  - At the same time, it scales to thousands of nodes and multi hour queries using the Spark engine, which provides full mid-query fault tolerance.

- **Uniform Data Access**
  - Connect to any data source the same way

```java
context.jsonFile("s3n://...")
.registerTempTable("json")
results = context.sql(
  """"SELECT *
  FROM people
  JOIN json ...
"
)
```

Query and join different data sources.

*Source: spark.apache.org/sql/*
Spark SQL (cont.)

- **Standard Connectivity**
  - Connect through JDBC or ODBC.

Source: https://spark.apache.org/sql/
Demo: Data Analytics Applications

- Spark SQL and MLlib
Detecting occurring events are essential in finding out what is happening in the city for decision-making or future planning purposes. It is a benefit for stakeholders such as government.

Social media event detection could be an additional detection apart from sensor-based event detection, where the detection coverage is not limited by the number of installed sensor as in sensor-based detection.
Demo: Data Analytics application

Analyzing Twitter data to detect traffic event

- Crawling twitter data
- Preprocessing
  - Data Cleansing & Transformation
  - Tokenization
  - Stop Word removal
- Supervised learning: binary classification (traffic, non-traffic), using > 1000 labeled data
  - Feature Extraction (Bag-of-words representation) using Spark’s HashingTF
  - Logistic Regression with default threshold 0.5 (adjustable)
- Evaluation Metrics
  - Well-known and common (prediction accuracy, Precision, Recall, F1-Measure)
cd /home/hduser/Tutorial/Analytic/EventDetection/App

spark-submit --master spark://sugi-ideapad:7077 --class Tutor_DataExtraction --executor-memory 8g --driver-memory 8g --conf "spark.debug.maxToStringFields=5000" --num-executors 7 --executor-cores 7 target/scala-2.11/tutorial_2.11-1.0.jar

spark-submit --master spark://sugi-ideapad:7077 --class Tutor_Classification --executor-memory 8g --driver-memory 8g --conf "spark.debug.maxToStringFields=5000" --num-executors 7 --executor-cores 7 target/scala-2.11/tutorial_2.11-1.0.jar
Performance Tuning of Apache Spark

- Cluster Resource Utilization
- Spark Web UI
- Partitioning
- Profiler
Spark executor in an application has the same fixed number of cores and same fixed heap size.

The number of cores can be specified with the `--executor-cores` flag when invoking `spark-submit`, `spark-shell`, and `pyspark` from the command line, or by setting the `spark.executor.cores` property in the `spark-defaults.conf` file or on a SparkConf object.

Similarly, the heap size can be controlled with the `--executor-memory` flag or the `spark.executor.memory` property.

```
spark-shell --master spark://<host>:7077 --executor-cores 17 --executor-memory 5g
spark-submit --class company.division.yourClass --master spark://<host>:7077 --executor-cores 17 --executor-memory 5g --name "countingSheep" CountingSheep-1.0.jar
```
Performance Tuning of Apache Spark

- Cluster Resource Utilization
- Spark Web UI
- Partitioning
- Profiler
Spark Web UI

Details for Job 0

Status: SUCCEEDED
Completed Stages: 3

Completed Stages (3)

<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>2</td>
<td>reduce at &lt;console&gt;:31</td>
<td>2015/11/26 14:57:56</td>
<td>0.2 s</td>
<td>2/2</td>
<td></td>
<td></td>
<td>625.7 KB</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>mapPartitions at VertexRDD:scala:358</td>
<td>2015/11/26 14:57:55</td>
<td>0.7 s</td>
<td>2/2</td>
<td>2.7 MB</td>
<td></td>
<td>319.4 KB</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>mapPartitions at GraphInput:scala:235</td>
<td>2015/11/26 14:57:55</td>
<td>0.7 s</td>
<td>2/2</td>
<td>2.7 MB</td>
<td></td>
<td>206.2 KB</td>
<td></td>
</tr>
</tbody>
</table>

A key part of understanding the performance of your job is the amount of data read in and out as well as the amount of data sent between shuffle boundaries.

Each stage of the job is listed with important execution statistics.

Source: (Malak & East 2016)
Performance Tuning of Apache Spark

- Cluster Resource Utilization
- Spark Web UI
- Partitioning
- Profiler
Partitioning (to utilize all CPU cores)

- Using the parameters to spark-shell or spark-submit, we can ensure that memory and CPUs are available on the cluster for our application. But that **doesn’t guarantee that all the available memory or CPUs will be used.**

- Spark processes a stage by processing each partition separately.

- In fact, **only one executor can work on a single partition**, so if the number of partitions is less than the number of executors, the stage won’t take advantage of the full resources available.

- Call RDD.partitions.size to find out how many partition your RDD has. The default partition size can be retrieved using `sc.defaultParallelism`.

- One way to resolve this problem is to use the *repartition method* on the RDD to supply a recommended new number of partitions:

  ```scala
def recommendNewPartitionNumber(rdd: RDD[Int], number: Int): RDD[Int] = 
    rdd
      .repartition(sc.defaultParallelism) 
      .mapPartitions { it: Array[Int] => it.take(number) }
```

  ```
  Val rdd = ...
  rdd.repartition(10)
  ```

**Source:** (Malak & East 2016)
Performance Tuning of Apache Spark

- Cluster Resource Utilization
- Spark Web UI
- Partitioning
- Profiler
YourKit Profiler

Call tree - All threads together

Call tree - By thread

Hot spots

Method list

Java EE statistics

Database

JSPs and Servlets

INDI

CPU usage telemetry

Tutorial: Big data analytics using Apache Spark

- Sugimiyanto Suma et al.
YourKit Profiler (cont.)
Advance Features of Apache Spark
Advance Features of Apache Spark

- Shared Variables
  - Accumulators
  - Broadcast Variable

- RDD Persistence
Shared Variables

- Normally, when a function passed to a Spark operation (such as `map` or `reduce`) is executed on a remote cluster node, it works on separate copies of all the variables used in the function.

- These variables are copied to each machine, and no updates to the variables on the remote machine are propagated back to the driver program.

- Supporting general, read-write shared variables across tasks would be inefficient. However, Spark does provide two limited types of shared variables for two common usage patterns: broadcast variables and accumulators.

  - **Broadcast variables** – used to efficiently, distribute large values.
  - **Accumulators** – used to aggregate the information of particular collection.

Source: https://www.tutorialspoint.com/apache_spark/advanced_spark_programming.htm
Accumulators

- Accumulators are variables that are only “added” to through an associative and commutative operation and can therefore be efficiently supported in parallel.

- They can be used to implement **counters** (as in MapReduce) or **sums**.

- Spark natively supports accumulators of numeric types, and programmers can add support for new types.

Source: [https://www.tutorialspoint.com/apache_spark/advanced_spark_programming.htm](https://www.tutorialspoint.com/apache_spark/advanced_spark_programming.htm)
Accumulators (cont.)

scala> val accum = sc.longAccumulator("My Accumulator")
accum: org.apache.spark.util.LongAccumulator = LongAccumulator(id: 0, name: Some(My Accumulator), value: 0)

scala> sc.parallelize(Array(1, 2, 3, 4)).foreach(x => accum.add(x))
...
10/09/29 18:41:08 INFO SparkContext: Tasks finished in 0.317106 s

scala> accum.value
res2: Long = 10
Accumulators (cont.)

### Accumulators

<table>
<thead>
<tr>
<th>Accumulable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>counter</td>
<td>45</td>
</tr>
</tbody>
</table>

### Tasks

<table>
<thead>
<tr>
<th>Index</th>
<th>ID</th>
<th>Attempt</th>
<th>Status</th>
<th>Locality Level</th>
<th>Executor ID / Host</th>
<th>Launch Time</th>
<th>Duration</th>
<th>GC Time</th>
<th>Accumulators</th>
<th>Errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>SUCCESS</td>
<td>PROCESS_LOCAL</td>
<td>driver / localhost</td>
<td>2016/04/21 10:10:41</td>
<td>17 ms</td>
<td></td>
<td>counter: 1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>SUCCESS</td>
<td>PROCESS_LOCAL</td>
<td>driver / localhost</td>
<td>2016/04/21 10:10:41</td>
<td>17 ms</td>
<td></td>
<td>counter: 2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0</td>
<td>SUCCESS</td>
<td>PROCESS_LOCAL</td>
<td>driver / localhost</td>
<td>2016/04/21 10:10:41</td>
<td>17 ms</td>
<td></td>
<td>counter: 7</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>0</td>
<td>SUCCESS</td>
<td>PROCESS_LOCAL</td>
<td>driver / localhost</td>
<td>2016/04/21 10:10:41</td>
<td>17 ms</td>
<td></td>
<td>counter: 5</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>0</td>
<td>SUCCESS</td>
<td>PROCESS_LOCAL</td>
<td>driver / localhost</td>
<td>2016/04/21 10:10:41</td>
<td>17 ms</td>
<td></td>
<td>counter: 6</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>0</td>
<td>SUCCESS</td>
<td>PROCESS_LOCAL</td>
<td>driver / localhost</td>
<td>2016/04/21 10:10:41</td>
<td>17 ms</td>
<td></td>
<td>counter: 7</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>0</td>
<td>SUCCESS</td>
<td>PROCESS_LOCAL</td>
<td>driver / localhost</td>
<td>2016/04/21 10:10:41</td>
<td>17 ms</td>
<td></td>
<td>counter: 7</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>0</td>
<td>SUCCESS</td>
<td>PROCESS_LOCAL</td>
<td>driver / localhost</td>
<td>2016/04/21 10:10:41</td>
<td>17 ms</td>
<td></td>
<td>counter: 17</td>
<td></td>
</tr>
</tbody>
</table>

*Source: (Malak & East 2016)*
Advance Features of Apache Spark

- Shared Variables
  - Accumulators
  - Broadcast Variable

- RDD Persistence
Broadcast Variables

- Broadcast variables allow the programmer to keep a read-only variable cached on each machine rather than shipping a copy of it with tasks.

- They can be used, for example, to give every node a copy of a large input dataset in an efficient manner.

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

Source: https://spark.apache.org/docs/2.2.0/rdd-programming-guide.html#broadcast-variables
Broadcast Variables (cont.)

```scala
scala> val broadcastVar = sc.broadcast(Array(1, 2, 3))
broadcastVar: org.apache.spark.broadcast.Broadcast[Array[Int]] = Broadcast(0)

scala> broadcastVar.value
res0: Array[Int] = Array(1, 2, 3)
```

Source: https://spark.apache.org/docs/2.2.0/rdd-programming-guide.html#broadcast-variables
Advance Features of Apache Spark

- Shared Variables
  - Accumulators
  - Broadcast Variable

- RDD Persistence
One of the most important capabilities in Spark is persisting (or caching) a dataset in memory across operations.

When you persist an RDD, each node stores any partitions of it, that it computes in memory and reuses them in other actions on that dataset (or datasets derived from it).

This allows future actions to be much faster (often by more than 10x).

Caching is a key tool for iterative algorithms and fast interactive use.

Spark’s cache is fault-tolerant – if any partition of an RDD is lost, it will automatically be recomputed using the transformations that originally created it.

An RDD can be persisted using:

- **Persist()**: each persisted RDD can be stored using a different storage level, allowing you, for example, to persist the dataset on disk, persist it in memory but as serialized Java objects (to save space), replicate it across nodes.

- **Cache()**: a shorthand for using the default storage level, which is **StorageLevel.MEMORY_ONLY** (store deserialized objects in memory)
The different storage level is set by passing a `StorageLevel` object (Scala, Java, Python) to `persist()`.

Full set of storage level:

<table>
<thead>
<tr>
<th>Storage Level</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEMORY_ONLY</td>
<td>Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, some partitions will not be cached and will be recomputed on the fly each time they're needed. This is the default level.</td>
</tr>
<tr>
<td>MEMORY_AND_DISK</td>
<td>Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed.</td>
</tr>
<tr>
<td>MEMORY_ONLY_SER (Java and Scala)</td>
<td>Store RDD as serialized Java objects (one byte array per partition). This is generally more space-efficient than deserialized objects, especially when using a fast serializer, but more CPU-intensive to read.</td>
</tr>
<tr>
<td>MEMORY_AND_DISK_SER (Java and Scala)</td>
<td>Similar to MEMORY_ONLY_SER, but spill partitions that don't fit in memory to disk instead of recomputing them on the fly each time they're needed.</td>
</tr>
<tr>
<td>DISK_ONLY</td>
<td>Store the RDD partitions only on disk.</td>
</tr>
<tr>
<td>MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc.</td>
<td>Same as the levels above, but replicate each partition on two cluster nodes.</td>
</tr>
<tr>
<td>OFF_HEAP (experimental)</td>
<td>Similar to MEMORY_ONLY_SER, but store the data in off-heap memory. This requires off-heap memory to be enabled.</td>
</tr>
</tbody>
</table>

Source: [https://spark.apache.org/docs/2.2.0/rdd-programming-guide.html#rdd-persistence](https://spark.apache.org/docs/2.2.0/rdd-programming-guide.html#rdd-persistence)
References

blog.cloudera.com


https://www.tutorialspoint.com/apache_spark/advanced_spark_programming.htm

https://spark.apache.org/docs/2.2.0/rdd-programming-guide.html#rdd-persistence

https://spark.apache.org/docs/2.2.0/rdd-programming-guide.html#rdd-persistence

https://spark.apache.org/docs/2.2.0/rdd-programming-guide.html#broadcast-variables


Malak, M.S. & East, R., 2016. Spark GraphX In action, Manning Publications.

Useful Resources

- Learning Spark
  Holden Karau, Andy Konwinski, Patrick Wendell & Matei Zaharia

- Advanced Analytics with Spark
  Sandy Ryza, Uri Laserson, Sean Owen & Josh Wills

- Spark GraphX in Action
  Michael S. Malek, Robin East

Tutorial: Big data analytics using Apache Spark
- Sugimiyanto Suma et al.
Thank you for attending HPCSaudi2018 Tutorial
Contact

Prof. Rashid Mehmood
Director of Research, Training, and Consultancy
HPC Center, King Abdul Aziz University, Jeddah, Saudi Arabia
Email: RMehmood@kau.edu.sa