****

Tutorial

**BIG DATA** for Developers

Parte 1: Introducción a Apache Hadoop

El advenimiento de una marea de nuevas fuentes de datos denominado *Big Data* está haciendo que las compañías se pregunten: ¿de qué forma puedo aprovechar esta información?

Es en este punto en donde generalmente comenzamos a hablar de **Apache Hadoop**.

Apache Hadoop es un proyecto open-source gestionado por la fundación Apache, originalmente implementado por Yahoo! en 2006 quién se basó para crearlo en especificaciones publicadas por Google en 2003 y 2004. Hadoop cuenta hoy con aportes de diferentes organizaciones como Facebook, Cloudera o Microsoft.

Apache Hadoop ha crecido para convertirse hoy en una *plataforma de datos* que no sólo procesa un inmenso volumen de datos estáticos sino que además soporta nuevas modalidades de carga de trabajo adicionales a la meta inicial del proyecto como consultas interactivas, proceso en tiempo real, almacenamiento de datos super escalable NoSQL y almacenamiento de datos en memoria – entre otras características.

En este tutorial de Hadoop para desarrolladores vamos a enfocarnos en cuestiones básicas como:

* Conceptos fundamentales de Apache Hadoop
* Escribir un programa MapReduce

### El núcleo de Apache Hadoop

* Hadoop Distributed File System (HDFS)
* MapReduce

Se denomina Hadoop Cluster al grupo de máquinas corriendo HDFS y MapReduce. Las máquinas individuales son denominadas nodos. Un cluster puede contener desde un sólo nodo a varios miles de ellos. Para la mayoría de los escenarios de aplicabilidad Hadoop es horizontalmente escalable, lo que significa que se puede esperar una mejor performance del sistema simplemente agregándole nodos.

### MapReduce

MapReduce es un método para distribuir tareas a través de múltiples nodos de forma que cada nodo sea el que procese la información que contiene.

Un job MapReduce consiste en varias fases del tipo Map -> Sort -> Shuffle -> Reduce

Las principales ventajas de abstraer estos trabajos en la forma MapReduce, corriendo sobre una infraestructura distribuida de CPUs y dispositivos de almacenamiento son:

* Paralelización y distribución de datos automática en bloques a través de una infraestructura distribuida totalmente escalable.
* Tolerancia a fallos de almacenamiento, cómputo e infraestructura de red.
* Despliegue, monitoreo y capacidad de asegurar la infraestructura.
* Abstracción transparente para los programadores.

La mayoría de los programas MapReduce están escritos en Java aunque pueden también escribirse en otros lenguajes usando la API de streaming de Hadoop. MapReduce abstrae el trabajo de implementación de la tecnología de forma que el desarrollador puede concentrarse en escribir las funciones para Map y Reduce.

**Conceptos y terminología MapReduce**

Los trabajos MapReduce están controlados por un software llamado demonio (o servicio) conocido como JobTracker. El demonio JobTracker reside en un *nodo* maestro; los clientes envían trabajos MapReduce al JobTracker. El JobTracker asigna tareas Map y Reduce a los nodos del cluster.

Cada uno de estos nodos corre un demonio llamado TaskTracker . El demonio TaskTracker es el responsable de instanciar una tarea Map o Reduce e informar el progreso de la misma de vuelta al JobTracker.

Un job es un programa con la capacidad de ejecución de *Mappers* y *Reducers* sobre un set de datos. Una task representa la ejecución de un único Mapper o Reducer en una porción de información.

Siempre habrá al menos la misma cantidad de intentos de tareas que equiparen a las tareas pendientes. Si un intento de tarea falla, otro será iniciado por el JobTracker. La ejecución especulativa de tareas puede resultar en más intentos de ejecución de tareas que en tareas completadas.

#### **MapReduce: el mapeador (the mapper)**

Hadoop siempre intenta asegurarse que los Mappers corren en los nodos que contienen la porción de datos requeridos localmente para minimizar el tráfico de red. Múltiples mapeadores corren en paralelo, cada uno procesando una porción de los datos provistos.

El Mapper lee los datos en la forma de pares key/value y devuelve cero u otros pares key/value: map(in\_key, in\_value) -> (inter\_key, inter\_value) list

El Mapper puede usar o completamente ignorar la llave provista. Por ejemplo, un patrón estándar es cuando leemos una línea de un archivo a la vez. La llave es el offset del byte dentro del archivo que marca el inicio de la línea. El valor es el contenido de la línea en sí misma. Típicamente la llave es considerada irrelevante. Si el Mapper devuelve algo, la salida debe ser en la forma de pares key/value.

**MapReduce: el reductor (the reducer)**

Luego que ha concluido la fase Map, todos los valores intermedios para una key intermedia son combinados dentro de una lista. Esta lista se pasa al Reducer. Puede haber un sólo Reducer o múltiples Reducers, esto se especifica en la configuración del *job.* El total de los valores asociados a una key intermedia en particular están garantizados de ser procesados por el mismo Reducer.

Las key intermedias y sus listas de valores son pasadas al Reducer ordenados por key. Este paso es conocido como el 'ordenamiento y la mezcla' (shuffle and sort). El Reducer devuelve cero o más pares key/value finales; los mismos son escritos al HDFS. En la práctica, el Reducer generalmente emite un sólo par key/value para cada key de entrada.

Es posible que algunas tareas Map necesiten más tiempo para completarse que otras, generalmente dado a fallas de software o falta de capacidad de procesamiento. Esto puede causar un cuello de botella ya que todos los mappers necesitan finalizar antes que cualquier reducer pueda comenzar. Hadoop usa ejecución especulativa para mitigar el impacto en estas situaciones.

Si un Mapper parece estar corriendo más lento que el resto, una nueva instancia será iniciada en otra máquina operando sobre la misma información. El resultado del Mapper que termine antes es el que será utilizado. Hadoop matará entonces el Mapper que todavía esté corriendo.

### Escribiendo un programa MapReduce

En esta sección mostraremos cómo usar la API de Hadoop para escribir un programa MapReduce en Java.

Cada una de las partes de un programa MapReduce (RecordReader, Mapper, Partitioner, Reducer, etc.) es creada por un developer; se espera que el desarrollador escriba al menos el Mapper, el Reducer y el código de control.

**Ejemplos de MapReduce**

**WordCount** es un programa de ejemplo que lee archivos de texto y cuenta con qué frecuencia aparecen las palabras. La entrada son archivos de texto y la salida son archivos del mismo tipo, cada línea de los cuáles contiene una palabra y la cuenta de cuán frecuentemente apareció – separados por una tabulación.

Cada mapper toma una línea como entrada y la quiebra en palabras. Luego, emite un par key/value de la palabra y 1. Cada reducer suma la cuenta por cada palabra y emite un único par key/value con la palabra y la suma.

Para optimizar el proceso, el reducer también es usado como un combinador en las salidas del map. Esto reduce la cantidad de datos enviados a través de la red combinando cada palabra en un registro único.

Para correr este ejemplo la sintaxis sería:

hadoop jar hadoop-\*-examples.jar wordcount [-m <#maps>] [-r <#reducers>] <in-dir> <out-dir>

Todos los archivos en el directorio de entrada son leídos y la cuenta de las palabras escrita en un directorio de salida. Se asume que tanto las entradas más salidas son alamacenadas en HDFS. Si los datos de entrada no se encuentran en formato HDFS - pero por el contrario se encuentran en un sistema de archivos local en cualquier otro sitio – es necesario copiar la información dentro del HDFS usando un comando similar a este:

hadoop dfs -copyFromLocal <local-dir> <hdfs-dir>

A continuación se muestra una aplicación *wordcount* típica implementada en Java:

package org.myorg;  
  
 import java.io.IOException;  
 import java.util.\*;  
  
 import org.apache.hadoop.fs.Path;  
 import org.apache.hadoop.conf.\*;  
 import org.apache.hadoop.io.\*;  
 import org.apache.hadoop.mapreduce.\*;  
 import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;  
 import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;  
 import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;  
 import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;  
  
 public class WordCount {  
  
 public static class Map extends Mapper<LongWritable, Text, Text, IntWritable> {  
 private final static IntWritable one = new IntWritable(1);  
 private Text word = new Text();  
  
 public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {  
 String line = value.toString();  
 StringTokenizer tokenizer = new StringTokenizer(line);  
 while (tokenizer.hasMoreTokens()) {  
 word.set(tokenizer.nextToken());  
 context.write(word, one);  
 }  
 }  
 }   
  
 public static class Reduce extends Reducer<Text, IntWritable, Text, IntWritable> {  
  
 public void reduce(Text key, Iterable<IntWritable> values, Context context)   
 throws IOException, InterruptedException {  
 int sum = 0;  
 for (IntWritable val : values) {  
 sum += val.get();  
 }  
 context.write(key, new IntWritable(sum));  
 }  
 }  
  
 public static void main(String[] args) throws Exception {  
 Configuration conf = new Configuration();  
  
 Job job = new Job(conf, "wordcount");  
  
 job.setOutputKeyClass(Text.class);  
 job.setOutputValueClass(IntWritable.class);  
  
 job.setMapperClass(Map.class);  
 job.setReducerClass(Reduce.class);  
  
 job.setInputFormatClass(TextInputFormat.class);  
 job.setOutputFormatClass(TextOutputFormat.class);  
  
 FileInputFormat.addInputPath(job, new Path(args[0]));  
 FileOutputFormat.setOutputPath(job, new Path(args[1]));  
  
 job.waitForCompletion(true);  
 }  
  
 }

Cada job MapReduce consiste de cuatro partes:

* El código controlador
* El código que corre en el cliente que configura y envía el job
* El Mapper
* El Reducer

Antes de profundizar en el código necesitamos cubrir algunos conceptos básicos sobre la API de Hadoop.

**Mapper: leyendo información desde el HDFS**

El InputFormat especifica la información suministrada al Mapper. A su vez, el InputFormat es especificado en el código controlador. El InputFormat define la ubicación de los datos de entrada como por ejemplo un archivo o directorio en HDFS. También determina cómo dividir la información suministrada en partes (**input splits**).

Cada Mapper procesa un único input split (tajada de información). InputFormat es la factoría desde la cual la interfaz RecordReader extrae los registros en la forma (key, value) de la fuente de datos de entrada.

FileInputFormat es la clase base usada para todos los InputFormat basados en archivos. TextInputFormat es el FileInputFormat predeterminado; TextInputFormat trata cada línea en los archivos terminada en \n como un valor. La Key es el offset del byte dentro del archivo que contiene la línea. KeyValueTextInputFormat mapea las líneas terminadas en \n como ‘key SEP value’. Para separar los datos se utiliza de forma predeterminada las tabulaciones. SequenceFileInputFormat es un archivo binario de pares (key, value) con metadata adicional. El objeto SequenceFileAsTextInputFormat es similar aunque mapea (key.toString(), value.toString()).

Las keys y values en Hadoop son objetos. Los values son objetos que implementan la interfaz writable. Las keys son objetos que implementan writableComparable.

*Writable*

Hadoop define sus propias clases ‘box’ para cadenas, enteros, etc.:

* IntWritable para ints
* LongWritable para longs
* FloatWritable para flotantes
* DoubleWritable para dobles
* Text para cadenas

La interfaz writable hace que la serialización sea rápida y fácil para Hadoop. No importa cual sea el type para value, siempre debe implementar la interfaz writable.

*WritableComparable*

Un writableComparable es un Writable que además es Comparable. Dos writableComparables pueden compararse entre sí para determinar su ‘orden’. Las keys deben ser writableComparables porque son suministradas al Reducer en forma ordenada.

Es importante notar que al margen de sus nombres, todas las clases box en Hadoop implementan tanto writable como writableComparable; por ejemplo intwritable es en realidad un writableComparable.

*Driver*

El código de control corre en la máquina cliente; configura el job y luego lo envía al cluster.

### Streaming API

Muchas organizaciones cuentan con desarrolladores hábiles en otros lenguajes que Java, como por ejemplo:

* C#
* Ruby
* Python
* Perl

La API de Streaming permite que los desarrolladores utilicen el lenguaje que deseen para escribir Mappers y Reducers siempre y cuando el lenguaje pueda leer desde el *standard input* y escribir al *standard output*. La gran ventaja que representa la API de Streaming es que no hay necesidad para los programadores de otros lenguajes de aprender Java, resultando en un tiempo de desarrollo más rápido y en la habilidad de poder utilizar librerías de código existentes.

*Cómo funciona el Streaming*

Para implementar el streaming se debe escribir programas Mapper y Reducer por separado en el lenguaje de elección. Estos programas recibirán la información via stdin (standard input) y deberán escribir su resultado al stdout (standard output).

Si se usa la clase TextInputFormat (predeterminada) el streaming Mapper recibe cada línea desde el archivo en stdin en donde no se haya pasado una key. Tanto el streaming Mapper como la salida del streaming Reducer deberían ser enviados al stdout en la forma ‘key (tab) value (newline)’ -- también pueden emplearse otros Separators en vez de las tabulaciones.

En Java todos los valores asociados a una key se pasan al Reducer como un iterador. Usando Hadoop Streaming , el Reducer es alimentado en la forma de pares (key, value), uno por línea desde el standard input (stdin).

El código escrito deberá llevar el registro de la *key* de forma que pueda detectar cuándo los valores de una nueva key aparecen ejecutando un Streaming Job.

A modo de ejemplo, para iniciar un Streaming Job se podría utilizar:

hadoop jar $HADOOP\_HOME/contrib/streaming/hadoop-streaming\*.jar \  
-input mylnputDirs \ -output myOutputDir \  
-mapper myMap.py \  
-reducer myReduce.py \ -file myMap.py \ -file myReduce.py

### Hive y Pig

El código MapReduce se escribe generalmente en Java, sin embargo puede ser escrito en otros lenguajes utilizando Hadoop Streaming lo que requiere un programador que entienda cómo pensar en términos de MapReduce, que entienda el problema que se trata de resolver y que cuente con suficiente tiempo para escribir y testear el código.

Muchas organizaciones cuentan con unos pocos desarrolladores que pueden escribir código MapReduce de calidad.

Mientras tanto hay mucha otra gente desea analizar la información:

* Analistas de datos
* Analistas de negocios
* Científicos de datos / Estadísticos

Es evidente entonces que necesitamos una capa de abstracción superior encima de MapReduce que provea la habilidad de consultar la información sin necesidad de conocer MapReduce íntimamente. Hive y Pig atienden esta necesidad.

*Hive*

Hive es una infraestructura de almacenamiento de datos basada en Hadoop. Hadoop provee escalamiento masivo y características de tolerancia a fallos para el almacenamiento y procesamiento de datos gracias el paradigma de programación MapReduce utilizando hardware genérico.

Hive está diseñado para facilitar el resúmen de datos, realizar consultas ad-hoc (es decir con un fin definido) y analizar largos volúmenes de datos. Provee un lenguaje de consultas simple llamado Hive QL basado en SQL y que permite a usuarios familiarizados con SQL realizar estas tareas de forma sencilla. Al mismo tiempo Hive QL permite a los programadores de MapReduce conectar sus mappers y reducers personalizados para realizar análisis más sofisticado que puede no ser soportado por las capacidades propias de Hive QL.

*Pig*

Pig es una plataforma de alto nivel para crear programas MapReduce usados en Hadoop. El lenguaje que utiliza la plataforma se llama Pig Latin. Pig Latin abstrae la programación de Java MapReduce en una notación más accesible que transforma a MapReduce en un lenguaje de alto nivel similar al SQL que utilizan los sistemas de gestión de bases de datos relacionales (RDBMS). Pig Latin puede extenderse utilizando UDF (User Defined Functions) que el usuario puede escribir en Java, Python, JavaScript, Ruby or Groovy y luego llamar directamente desde el lenguaje.