

INTRODUCTION

Apache Spark is a cluster computing framework for large-scale data processing. Spark does not use MapReduce as an execution engine; instead, it uses its own distributed runtime for executing work on a cluster.

Spark is closely integrated with Hadoop: it can run on YARN and works with Hadoop file formats and storage backends like HDFS.

Spark is best known for its ability to keep large working datasets in memory between jobs.

This capability allows Spark to outperform the equivalent MapReduce, where datasets are always loaded from disk.

Two styles of application that benefit greatly from Spark's processing model are iterative algorithms and interactive analysis.

Unlike MapReduce, Spark's DAG engine can process arbitrary pipelines of operators and translate them into a single job for the user.

Spark has a rich set of APIs for performing many common data processing tasks, such as joins. Spark provides APIs in three languages: Scala, Java, and Python.

Spark also comes with a REPL (read—eval—print loop) for both Scala and

Python, which makes it quick and easy to explore datasets.

Spark is proving to be a good platform on which to build analytics tools, too, modules for machine learning (MLlib), graph processing (GraphX), stream processing (Spark Streaming), and SQL (Spark SQL).

Installing Spark

Download a stable release of the Spark binary distribution from the downloads page (choose the one that matches the Hadoop distribution you are using), and unpack the tarball in a suitable location:

```
% tar xzf spark-x.y.z-bin-distro.tgz
```

It's convenient to put the Spark binaries on your path as follows:

```
% export SPARK_HOME=~/.sw/spark-x.y.z-bin-distro
```

```
% export PATH=$PATH:$SPARK_HOME/bin
```

Spark Applications, Jobs, Stages, and Tasks

Spark has the concept of a job. A Spark job is more general than a MapReduce job, though, since it is made up of an arbitrary directed acyclic graph (DAG) of stages, each of which is roughly equivalent to a map or reduce phase in MapReduce.

Stages are split into tasks by the Spark runtime and are run in parallel on partitions of an RDD spread across the cluster.

A job always runs in the context of an application (represented by a SparkContext instance) that serves to group RDDs and shared variables.

An application can run more than one job, in series or in parallel, and provides the mechanism for a job to access an RDD that was cached by a previous job in the same application.

A Scala Standalone Application

Packaging a Scala shell program into a self-contained application that can be run more than once.

```
import org.apache.spark.SparkContext._
import org.apache.spark.{SparkConf, SparkContext}

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

    sc.textFile(args(0))
      .map(_._split("\t"))
      .filter(rec => (rec(1) != "9999" && rec(2).matches("[01459]")))
      .map(rec => (rec(0).toInt, rec(1).toInt))
      .reduceByKey((a, b) => Math.max(a, b))
      .saveAsTextFile(args(1))
  }
}
```

```
% spark-submit --class MaxTemperature --master local \
spark-examples.jar input/ncdc/micro-tab/sample.txt output
% cat output/part-*
(1950,22)
(1949,111)
```

```

public class MaxTemperatureSpark {

    public static void main(String[] args) throws Exception {
        if (args.length != 2) {
            System.err.println("Usage: MaxTemperatureSpark <input path> <output path>");
            System.exit(-1);
        }

        SparkConf conf = new SparkConf();
        JavaSparkContext sc = new JavaSparkContext("local", "MaxTemperatureSpark", conf);
        JavaRDD<String> lines = sc.textFile(args[0]);
        JavaRDD<String[]> records = lines.map(new Function<String, String[]>() {
            @Override public String[] call(String s) {
                return s.split("\t");
            }
        });
        JavaRDD<String[]> filtered = records.filter(new Function<String[], Boolean>() {
            @Override public Boolean call(String[] rec) {
                return rec[1] != "9999" && rec[2].matches("[01459]");
            }
        });
        JavaPairRDD<Integer, Integer> tuples = filtered.mapToPair(
            new PairFunction<String[], Integer, Integer>() {
                @Override public Tuple2<Integer, Integer> call(String[] rec) {
                    return new Tuple2<Integer, Integer>(
                        Integer.parseInt(rec[0]), Integer.parseInt(rec[1]));
                }
            }
        );
        JavaPairRDD<Integer, Integer> maxTemps = tuples.reduceByKey(
            new Function2<Integer, Integer, Integer>() {
                @Override public Integer call(Integer i1, Integer i2) {
                    return Math.max(i1, i2);
                }
            }
        );
    }
}

```

A Python Example

Spark also has language support for Python, in an API called PySpark. By taking advantage of Python's lambda expressions, we can rewrite the example program in a way that closely mirrors the Scala equivalent.

```

from pyspark import SparkContext
import re, sys

sc = SparkContext("local", "Max Temperature")
sc.textFile(sys.argv[1]) \
    .map(lambda s: s.split("\t")) \
    .filter(lambda rec: (rec[1] != "9999" and re.match("[01459]", rec[2]))) \
    .map(lambda rec: (int(rec[0]), int(rec[1]))) \
    .reduceByKey(max) \
    .saveAsTextFile(sys.argv[2])

```

```
% spark-submit --master local \
  ch19-spark/src/main/python/MaxTemperature.py \
  input/ncdc/micro-tab/sample.txt output
```

Resilient Distributed Datasets

RDDs are at the heart of every Spark program.

Creation

There are three ways of creating RDDs:

- *from an in-memory collection of objects (known as parallelizing a collection),*
- *using a dataset from external storage (such as HDFS), or*
- *transforming an existing RDD.*

The first way is useful for doing CPU-intensive computations on small amounts of input data in parallel.

For example, the following runs separate computations on the numbers from 1 to 10:

```
val params = sc.parallelize(1 to 10)
val result = params.map(performExpensiveComputation)
```

The performExpensiveComputation function is run on input values in parallel. The level of parallelism is determined from the spark.default.parallelism property.

The default values of spark.default.parallelism depends on where the spark job is running:

*spark.default.parallelism = number of cores on the machine, if it is
running locally*

*spark.default.parallelism = number of cores on all executor
machines if it is running on a cluster*

The user can also override the level of parallelism for a particular computation by passing it as the second argument to parallelize():

sc.parallelize(1 to 10, 10)

The second way to create an RDD is by creating a reference to an external dataset.

val text: RDD[String] = sc.textFile(inputPath)

The path may be any Hadoop filesystem path, such as a file on the local filesystem or on HDFS. Internally, Spark uses TextInputFormat from the old MapReduce API to read the file.

This means that the file-splitting behavior is the same as in Hadoop itself, so in the case of HDFS there is one Spark partition per HDFS block.

The default can be changed by passing a second argument to request a particular number of splits:

```
sc.textFile(inputPath, 10)
```

Another variant permits text files to be processed as whole files by returning an RDD of string pairs, where the first string is the file path and the second is the file contents. Since each file is loaded into memory, this is only suitable for small files:

```
val files: RDD[(String, String)] = sc.wholeTextFiles(inputPath)
```

Spark can work with other file formats besides text. For example, sequence files can be read with:

```
sc.sequenceFile[IntWritable, Text](inputPath)
```

For common Writable types, Spark can map them to the Java equivalents, so we could use the equivalent form:

```
sc.sequenceFile[Int, String](inputPath)
```

There are two methods for creating RDDs from an arbitrary Hadoop InputFormat:

hadoopFile() for file-based formats that expect a path, and

hadoopRDD() for those that don't, such as HBase's TableInputFormat.

These methods are for the old MapReduce API; for the new one, use newAPIHadoopFile() and newAPIHadoopRDD(). Here is an example of reading an Avro datafile using the Specific API with a WeatherRecord class:

```
val job = new Job()
AvroJob.setInputKeySchema(job, WeatherRecord.getClassSchema)
val data = sc.newAPIHadoopFile(inputPath,
    classOf[AvroKeyInputFormat[WeatherRecord]],
    classOf[AvroKey[WeatherRecord]], classOf[NullWritable],
    job.getConfiguration)
```

Transformations and Actions

Spark provides two categories of operations on RDDs:

*transformations and
actions.*

A transformation generates a new RDD from an existing one, while an action triggers a computation on an RDD and does something with the results—either returning them to the user or saving them to external storage.

Actions have an immediate effect, but transformations do not—they are lazy, in the sense that they don't perform any work until an action is performed on the transformed RDD.

For example, the following lowercases lines in a text file:

```
val text = sc.textFile(inputPath)
val lower: RDD[String] = text.map(_.toLowerCase())
lower.foreach(println(_))
```

The map() method is a transformation, which Spark represents internally as a function (toLowerCase()) to be called at some later time on each element in the input RDD (text).

The function is not actually called until the foreach() method (which is an action) is invoked and Spark runs a job to read the input file and call toLowerCase() on each line in it, before writing the result to the console.

One way of telling if an operation is a transformation or an action is by looking at its return type: if the return type is RDD, then it's a transformation; otherwise, it's an action.

Spark's library contains a rich set of operators, including transformations for mapping, grouping, aggregating, repartitioning, sampling, and joining RDDs, and for treating RDDs as sets.

There are also actions for materializing RDDs as collections, computing statistics on RDDs, sampling a fixed number of elements from an RDD, and saving RDDs to external storage.

Aggregation transformations

The three main transformations for aggregating RDDs of pairs by their keys are reduceByKey(), foldByKey(), and aggregateByKey(). They work in slightly different ways, but they all aggregate the values for a given key to produce a single value for each key.

The simplest is reduceByKey(), which repeatedly applies a binary function to values in pairs until a single value is produced. For example:

```
val pairs: RDD[(String, Int)] =
    sc.parallelize(Array(("a", 3), ("a", 1), ("b", 7), ("a", 5)))
val sums: RDD[(String, Int)] = pairs.reduceByKey(_+_ )
assert(sums.collect().toSet === Set(("a", 9), ("b", 7)))
```

The order and grouping of the operations should not matter; in this case, the aggregation could be $5 + (3 + 1)$, or $3 + (1 + 5)$, which both return the same result.

The triple equals operator (===) used in the assert statement is from

ScalaTest, and provides more informative failure messages than using the regular == operator.

we would perform the same operation using foldByKey():

```
val sums: RDD[(String, Int)] = pairs.foldByKey(0)(_+_ ) assert(sums.collect().toSet === Set(("a", 9), ("b", 7)))
```

Notice that this time we had to supply a zero value, which is just 0 when adding integers, but would be something different for other types and operations. This time, values for a are aggregated as $((0 + 3) + 1) + 5 = 9$ (or possibly some other order, although adding to 0 is always the first operation). For b it is $0 + 7 = 7$.

Using foldByKey() is no more or less powerful than using reduceByKey(). In particular, neither can change the type of the value that is the result of the aggregation. For that we need aggregateByKey(). For example, we can aggregate the integer values into a set:

```
val sets: RDD[(String, HashSet[Int])] =
    pairs.aggregateByKey(new HashSet[Int])(_+=_, _+=_)
assert(sets.collect().toSet === Set(("a", Set(1, 3, 5)), ("b", Set(7))))
```

Persistence

A transformed RDD can be persisted in memory so that subsequent operations on it are more efficient.

We can cache the intermediate dataset of year-temperature pairs in memory. Consider the same example for finding out temperature with the following based on quality code:

```
scala> tuples.cache()
```

```
res1: tuples.type = MappedRDD[4] at map at <console>:18
```

Calling cache() does not cache the RDD in memory straightaway. Instead, it marks the RDD with a flag indicating it should be cached when the Spark job is run. So, let's first force a job run:


```
scala> tuples.reduceByKey((a, b) => Math.max(a, b)).foreach(println(_))
INFO BlockManagerInfo: Added rdd_4_0 in memory on 192.168.1.90:64640
INFO BlockManagerInfo: Added rdd_4_1 in memory on 192.168.1.90:64640
(1950,22)
(1949,111)
```

The log lines for BlockManagerInfo show that the RDD's partitions have been kept in memory as a part of the job run. The log shows that the RDD's number is 4 (this was shown in the console after calling the cache() method), and it has two partitions labeled 0 and 1. If we run another job on the cached dataset, we'll see that the RDD is loaded from memory. This time we'll compute minimum temperatures:

```
scala> tuples.reduceByKey((a, b) => Math.min(a, b)).foreach(println(_))
INFO BlockManager: Found block rdd_4_0 locally
INFO BlockManager: Found block rdd_4_1 locally
(1949,78)
(1950,-11)
```

Persistence levels

Calling cache() will persist each partition of the RDD in the executor's memory. If an executor does not have enough memory to store the RDD partition, the computation will not fail, but instead the partition will be recomputed as needed. For complex programs with lots of transformations, this may be expensive, so Spark offers different types of persistence behavior that may be selected by calling persist() with an argument to specify the StorageLevel.

By default, the level is MEMORY_ONLY, which uses the regular in-memory representation of objects. A more compact representation can be used by serializing the elements in a partition as a byte array. This level is MEMORY_ONLY_SER; it incurs CPU overhead compared to MEMORY_ONLY, but is worth it if the resulting serialized RDD partition fits in memory when the regular in-memory representation doesn't.

MEMORY_ONLY_SER also reduces garbage collection pressure, since each RDD is stored as one byte array rather than lots of objects.

By default, regular Java serialization is used to serialize RDD partitions, but Kryo serialization is normally a better choice, both in terms of size and speed. Further space savings can be achieved by compressing the serialized partitions by setting the spark.rdd.compress property to true, and optionally setting spark.io.compression.codec.

Serialization

There are two aspects of serialization to consider in Spark: serialization of data and serialization of functions (or closures).

DATA SERIALIZATION

By default, Spark will use Java serialization to send data over the network from one executor to another, or when caching (persisting) data in serialized form.

A better choice for most Spark programs is Kryo serialization. Kryo is a more efficient general-purpose serialization library for Java. In order to use Kryo serialization, set the `spark.serializer` as follows on the `SparkConf` in your driver program:

```
conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer")
```

Kryo does not require that a class implement a particular interface (like `java.io.Serializable`) to be serialized, so plain old Java objects can be used in RDDs without any further work beyond enabling Kryo serialization.

It is much more efficient to register classes with Kryo before using them. This is because Kryo writes a reference to the class of the object being serialized (one reference is written for every object written), which is just an integer identifier if the class has been registered but is the full classname otherwise.

Registering classes with Kryo is straightforward. Create a subclass of `KryoRegistrar`, and override the `registerClasses()` method:

```
class CustomKryoRegistrar extends KryoRegistrar {  
  override def registerClasses(kryo: Kryo) {  
    kryo.register(classOf[WeatherRecord])  
  }  
}
```

Finally, in the driver program, set the `spark.kryo.registrator` property to the fully qualified classname of your `KryoRegistrar` implementation:

```
conf.set("spark.kryo.registrator", "CustomKryoRegistrar")
```

Functions

Generally, serialization of functions will “just work”: in Scala, functions are serializable using the standard Java serialization mechanism, which is what Spark uses to send functions to remote executor nodes.

Spark will serialize functions even when running in local mode, so if you inadvertently introduce a function that is not serializable it is caught early on in the development process.

Shared Variables

In Spark, shared variables are variables that are used by many functions and methods in parallel. They are used in parallel operations and allow for efficient data sharing and consistency across nodes.

Types of Shared Variables:

1. **Broadcast Variable**
2. **Accumulator**

Broadcast Variable:

It is an optimization technique in the Spark SQL engine that lets you keep a cached version of a variable on each machine rather than shuffling it around.

*This reduces the shuffling of data across nodes. Broadcast variables are created from a variable *v* by calling `SparkContext.broadcast(v)`.*

*The broadcast variable is a wrapper around *v*, and its value can be accessed by calling the `value` method. The code below shows this*

```
val lookup: Broadcast[Map[Int, String]] =  
    sc.broadcast(Map(1 -> "a", 2 -> "e", 3 -> "i", 4 -> "o", 5 -> "u"))  
val result = sc.parallelize(Array(2, 1, 3)).map(lookup.value(_))  
assert(result.collect().toSet == Set("a", "e", "i"))
```

As the name suggests, broadcast variables are sent one way, from driver to task—there is no way to update a broadcast variable and have the update propagate back to the driver.

For that, we need an accumulator.

Accumulators

An accumulator is a shared variable that tasks can only add to, like counters in Map-Reduce. After a job has completed, the accumulator's final value can be retrieved from the driver program.

Here is an example that counts the number of elements in an RDD of integers using an accumulator, while at the same time summing the values in the RDD using a `reduce()` action:

```
val count: Accumulator[Int] = sc.accumulator(0)
val result = sc.parallelize(Array(1, 2, 3))
```

```
    .map(i => { count += 1; i })
    .reduce((x, y) => x + y)
assert(count.value == 3)
assert(result == 6)
```

When the result of the Spark job has been computed, the value of the accumulator is accessed by calling value on it.

Anatomy of a Spark Job Run

When we run a Spark job. At the highest level, there are two independent entities:

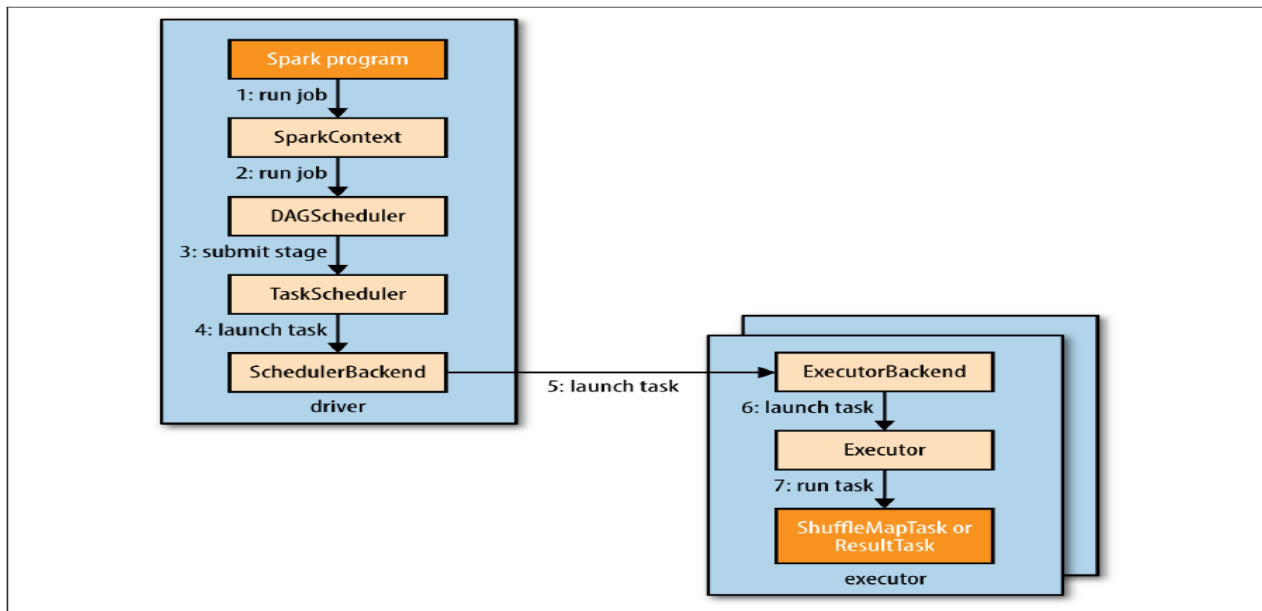
the driver, which hosts the application (SparkContext) and schedules tasks for a job; and

the executors, which are exclusive to the application, run for the duration of the application, and execute the application's tasks.

Usually, the driver runs as a client that is not managed by the cluster manager and the executors run on machines in the cluster.

1. Spark job is submitted automatically when an action (such as count()) is performed on an RDD. Internally, this causes runJob() to be called on the SparkContext which passes the call on to the scheduler that runs as a part of the driver.

2. The scheduler is made up of two parts: a DAG scheduler that breaks down the job into a DAG of stages, and a task scheduler that is responsible for submitting the tasks from each stage to the cluster.



DAG Construction

To understand how a job is broken up into stages, we need to look at the type of tasks that can run in a stage. There are two types:

*shuffle map tasks and
result tasks.*

The name of the task type indicates what Spark does with the task's output:

Shuffle map tasks

As the name suggests, shuffle map tasks are like the map-side part of the shuffle in MapReduce. Each shuffle map task runs a computation on one RDD partition and, based on a partitioning function, writes its output to a new set of partitions, which are then fetched in a later stage. Shuffle map tasks run in all stages except the final stage.

Result tasks

Result tasks run in the final stage that returns the result to the user's program. Each result task runs a computation on its RDD partition, then sends the result back to the driver, and the driver assembles the results from each partition into a final result.

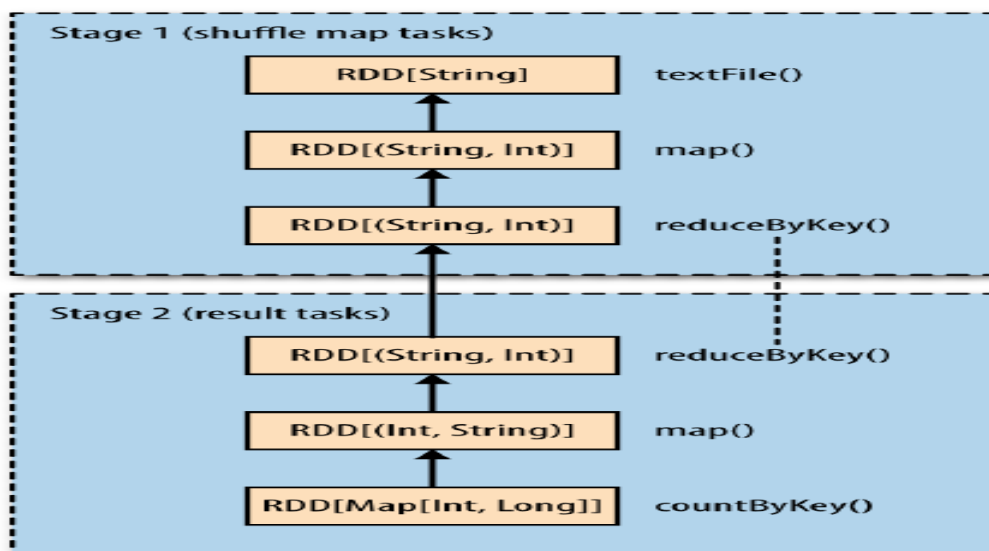
The simplest Spark job is one that does not need a shuffle and therefore has just a single stage composed of result tasks. This is like a map-only job in MapReduce.

More complex jobs involve grouping operations and require one or more shuffle stages.

For example, consider the following job for calculating a histogram of word counts for text files stored in `inputPath` (one word per line):

```
val hist: Map[Int, Long] = sc.textFile(inputPath)
  .map(word => (word.toLowerCase(), 1))
  .reduceByKey((a, b) => a + b)
  .map(_._swap)
  .countByKey()
```

Spark's DAG scheduler turns this job into two stages since the `reduceByKey()` operation forces a shuffle stage.⁴ The resulting DAG is illustrated in the following figure:



If an RDD has been persisted from a previous job in the same application (SparkCon text), then the DAG scheduler will save work and not create stages for recomputing it (or the RDDs it was derived from).

The DAG scheduler is responsible for splitting a stage into tasks for submission to the task scheduler. In this example, in the first stage one shuffle map task is run for each partition of the input file. The level of parallelism for a `reduceByKey()` operation can be set explicitly by passing it as the second parameter

Each task is given a placement preference by the DAG scheduler to allow the task scheduler to take advantage of data locality.

A task that processes a partition of an input RDD stored on HDFS, for example, will have a placement preference for the datanode hosting the partition's block (known as node local), while a task that processes a partition of an RDD that is cached in memory will prefer the executor storing the RDD partition (process local).

Once the DAG scheduler has constructed the complete DAG of stages, I

it submits each stage's set of tasks to the task scheduler (step 3).

Child stages are only submitted once their parents have completed successfully.

Task Scheduling

- *When the task scheduler is sent a set of tasks, it uses its list of executors that are running for the application and constructs a mapping of tasks to executors that takes placement preferences into account.*
- *Next, the task scheduler assigns tasks to executors that have
free cores (this may not be the complete set if another job in the same
application is running), and it continues to assign more tasks as executors
finish running tasks, until the task set is complete.*
- *Each task is allocated one core by default, although this can be
changed by setting `spark.task.cpus`.*
- *Note that for a given executor the scheduler will first assign process-local tasks, then
node-local tasks, then rack-local tasks, before assigning an arbitrary (nonlocal) task, or a
speculative task if there are no other candidates. Assigned tasks are launched through a
scheduler backend which
sends a remote launch task message (step 5) to the executor backend to tell
the executor to run the task (step 6).*
- *Executors also send status update messages to the driver when a task has finished or if a
task fails. In the latter case, the task scheduler will resubmit the task on another executor.
It will also launch speculative tasks for tasks that are running slowly, if this is enabled.*

Task Execution

An executor runs a task as follows (step 7).

- *First, it makes sure that the JAR and file dependencies for the task are up to date. The
executor keeps a local cache of all the dependencies that previous tasks have used, so
that it only downloads them when they have changed.*
- *Second, it deserializes the task code (which includes the user's functions) from the
serialized bytes that were sent as a part of the launch task message.*
- *Third, the task code is executed. Note that tasks are run in the same JVM as the executor,
so there is no process overhead for task launch.*
- *Tasks can return a result to the driver. The result is serialized and sent to the executor
backend, and then back to the driver as a status update message. A shuffle map task
returns information that allows the next stage to retrieve the output partitions, while a
result task returns the value of the result for the partition it ran on, which the driver
assembles into a final result to return to the user's program.*

HBasics

HBase is a distributed column-oriented database built on top of HDFS. HBase is the Hadoop application to use when you require real-time read/write random access to very large datasets.

Although there are countless strategies and implementations for database storage and retrieval, most solutions—especially those of the relational variety—are not built with very large scale and distribution in mind. Many vendors offer replication and partitioning solutions to grow the database beyond the confines of a single node, but these add-ons are generally an afterthought and are complicated to install and maintain. They also severely compromise the RDBMS feature set. Joins, complex queries, triggers,

views, and foreign-key constraints become prohibitively expensive to run on a scaled RDBMS, or do not work at all.

HBase approaches the scaling problem from the opposite direction.

It is built from the ground up to scale linearly just by adding nodes. HBase is not relational and does not support SQL, but given the proper problem space, it is able to do what an RDBMS cannot: host very large, sparsely populated tables on clusters made from commodity hardware.

Backdrop

The HBase project was started toward the end of 2006 by Chad Walters and Jim Kellerman at Powerset. It was modeled after Google's Bigtable, which had just been published.² In February 2007, Mike Cafarella made a code drop of a mostly working system that Jim Kellerman then carried forward.

The first HBase release was bundled as part of Hadoop 0.15.0 in October 2007. In May 2010, HBase graduated from a Hadoop subproject to become an Apache Top Level Project. Today, HBase is a mature technology used in production across a wide range of industries.

Whirlwind Tour of the Data Model

Applications store data in labeled tables. Tables are made of rows and columns. Table cells—the intersection of row and column coordinates—are versioned. By default, their version is a timestamp auto-assigned by HBase at the time of cell insertion. A cell's content is an uninterpreted array of bytes. An example HBase table for storing photos is shown below:

The diagram illustrates an HBase table structure. It features a grid with three rows and two main column families. The first column family is 'contents' with a sub-column 'image', and the second is 'info' with sub-columns 'format' and 'geo'. The rows are indexed by a row key, with values 000001, 000002, and 000003. A vertical arrow on the left indicates 'Increasing row key'. The 'image' column contains icons of a sad face, a stack of happy faces, and a stack of sad faces. The 'format' column contains 'jpeg', 'tiff', and 'jpeg'. The 'geo' column contains '51.5,-0.1' and '51.8,-3.1'. Arrows at the bottom point to the 'Versions' and 'Cells' of the data.

	Column family contents contents:image	Column family info info:format	info:geo
000001		jpeg	51.5,-0.1
000002		tiff	
000003		jpeg	51.8,-3.1

Increasing row key

Versions

Cells

Table row keys are also byte arrays, so theoretically anything can serve as a row key, from strings to binary representations of long or even serialized data structures. Table rows are sorted by row key, that is the table's primary key. The sort is byte-ordered. All table accesses are via the primary key.

Row columns are grouped into column families. All column family members have a common prefix, so, for example, the columns `info:format` and `info:geo` are both members of the `info` column family, whereas `contents:image` belongs to the `contents` family.

The column family prefix must be composed of printable characters. The qualifying tail, the column family qualifier, can be made of any arbitrary bytes. The column family and the qualifier are always separated by a colon character (:).

A table's column families must be specified up front as part of the table schema definition, but new column family members can be added on demand. For example, a new column `info:camera` can be offered by a client as part of an update, and its value persisted, as long as the column family `info` already exists on the table.

Physically, all column family members are stored together on the filesystem. So, although it is described that HBase as a column-oriented store, it would be more accurate if it were described as a column-family-oriented store.

Regions

Tables are automatically partitioned horizontally by HBase into regions. Each region comprises a subset of a table's rows. A region is denoted by the table it belongs to, its first row (inclusive), and its last row (exclusive). Initially, a table comprises a single region, but as the region grows it eventually crosses a configurable size threshold, at which point it splits at a row boundary into two new regions of approximately equal size. Until this first split happens, all loading will be against the single server hosting the original region. As the table grows, the number of its regions grows.

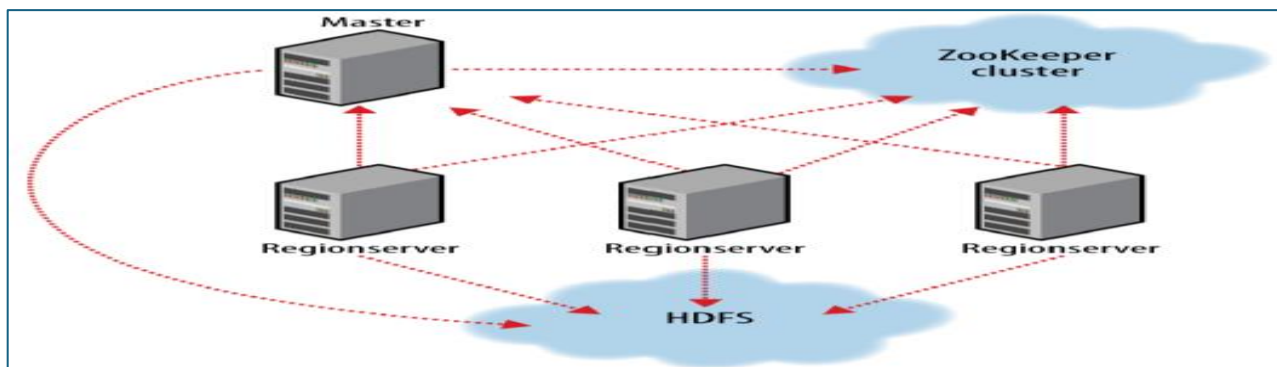
Regions are the units that get distributed over an HBase cluster.

Locking

Row updates are atomic, no matter how many row columns constitute the row-level transaction. This keeps the locking model simple.

Implementation

Just as HDFS and YARN are built of clients, workers, and a coordinating master—the namenode and datanodes in HDFS and resource manager and node managers in YARN—so is HBase made up of an HBase master node orchestrating a cluster of one or more regionserver workers.



The HBase master is responsible for bootstrapping install, for assigning regions to registered regionserver, and for recovering regionserver failures. The master node is lightly loaded.

The regionserver carry zero or more regions and field client read/write requests. They also manage region splits, informing the HBase master about the new daughter regions so it can manage the offlining of parent regions and assignment of the replacement daughters.

HBase depends on ZooKeeper. The ZooKeeper ensemble hosts vitals such as the location of the hbase:meta catalog table and the address of the current cluster master.

Assignment of regions is mediated via ZooKeeper in case participating servers crash midassignment. Hosting the assignment transaction state in ZooKeeper makes it so recovery can pick up on the assignment where the crashed server left off.

Regionserver worker nodes are listed in the HBase conf/regionserver file, as you would list datanodes and node managers in the Hadoopetc/hadoop/slaves file.

Start and stop scripts are like those in Hadoop and use the same SSH-based mechanism for running remote commands.

A cluster's site-specific configuration is done in the HBase

conf/hbase-site.xml and

conf/hbase-env.sh files,

which have the same format as their equivalents in the Hadoop parent project.

HBase persists data via the Hadoop filesystem API. By default HBase writes to the local filesystem.

HBase in operation

Internally, HBase keeps a special catalog table named `hbase:meta`, within which it maintains the current list, state, and locations of all user-space regions afloat on the cluster.

Entries in `hbase:meta` are keyed by region name, where a region name is made up of the name of the table the region belongs to, the region's start row, its time of creation, and finally, an MD5 hash of all of these (i.e., a hash of table name, start row, and creation timestamp).

Here is an example region name for a region in the table `TestTable` whose start row is `xyz`:

`TestTable,xyz,1279729913622.1b6e176fb8d8aa88fd4ab6bc80247ece.`

Commas delimit the table name, start row, and timestamp. The MD5 hash is surrounded by a leading and trailing period.

Writes arriving at a regionserver are first appended to a commit log and then added to an in-memory memstore. When a memstore fills, its content is flushed to the filesystem.

The commit log is hosted on HDFS, so it remains available through a regionserver crash. When the master notices that a regionserver is no longer reachable, usually because the server's znode has expired in ZooKeeper, it splits the dead regionserver's commit log by region. On reassignment and before they reopen for business, regions that were on the

dead regionserver will pick up their just-split files of not-yet-persisted edits and replay them to bring themselves up to date with the state they had just before the failure.

Installation

Download a stable release from an Apache Download Mirror and unpack it on your local filesystem. For example:

```
% tar xzf hbase-x.y.z.tar.gz
```

For convenience, add the HBase binary directory to your command-line path. For example:

```
% export HBASE_HOME=~/.sw/hbase-x.y.z
```

```
% export PATH=$PATH:$HBASE_HOME/bin
```

To get the list of HBase options, use the following:

% hbase


Options:

--config DIR Configuration direction to use. Default: ./conf
--hosts HOSTS Override the list in 'regionserver' file

Commands:

Some commands take arguments. Pass no args or -h for usage.

shell	Run the HBase shell
hbck	Run the hbase 'fsck' tool
hlog	Write-ahead-log analyzer
hfile	Store file analyzer
zkcli	Run the ZooKeeper shell
upgrade	Upgrade hbase
master	Run an HBase HMaster node
regionserver	Run an HBase HRegionServer node
zookeeper	Run a Zookeeper server
rest	Run an HBase REST server
thrift	Run the HBase Thrift server
thrift2	Run the HBase Thrift2 server
clean	Run the HBase clean up script
classpath	Dump hbase CLASSPATH
mapredcp	Dump CLASSPATH entries required by mapreduce
pe	Run PerformanceEvaluation
ltt	Run LoadTestTool
version	Print the version
CLASSNAME	Run the class named CLASSNAME

 BDHS

Test Drive

To administer your HBase instance, launch the HBase shell as follows:

% hbase shell

HBase Shell; enter 'help<RETURN>' for list of supported commands.

Type "exit<RETURN>" to leave the HBase Shell

Version 0.98.7-hadoop2, r800c23e2207aa3f9bddb7e9514d8340bcfb89277, Wed Oct 8

15:58:11 PDT 2014

hbase(main):001:0>

This will bring up a JRuby IRB interpreter that has had some HBase-specific commands added to it. Type help and then press Return to see the list of shell commands grouped into categories.

To create a table named test with a single column family named data using defaults for table and column family attributes, enter:

```
hbase(main):001:0> create 'test', 'data'  
0 row(s) in 0.9810 seconds
```

To prove the new table was created successfully, run the list command. This will output all tables in user space:

```
hbase(main):002:0> list
TABLE
test
1 row(s) in 0.0260 seconds
```

To insert data into three different rows and columns in the data column family, get the first row, and then list the table content, do the following:

```
hbase(main):003:0> put 'test', 'row1', 'data:1', 'value1'
hbase(main):004:0> put 'test', 'row2', 'data:2', 'value2'
hbase(main):005:0> put 'test', 'row3', 'data:3', 'value3'
hbase(main):006:0> get 'test', 'row1'
COLUMN                                CELL
data:1                                timestamp=1414927084811, value=value1
1 row(s) in 0.0240 seconds
hbase(main):007:0> scan 'test'
ROW                                    COLUMN+CELL
row1                                   column=data:1, timestamp=1414927084811, value=value1
```

```
row2                                   column=data:2, timestamp=1414927125174, value=value2
row3                                   column=data:3, timestamp=1414927131931, value=value3
3 row(s) in 0.0240 seconds
```

Notice how we added three new columns without changing the schema.

To remove the table, you must first disable it before dropping it:

```
hbase(main):009:0> disable 'test'
0 row(s) in 5.8420 seconds
hbase(main):010:0> drop 'test'
0 row(s) in 5.2560 seconds
hbase(main):011:0> list
TABLE
0 row(s) in 0.0200 seconds
```

Shut down your HBase instance by running:

```
% stop-hbase.sh
```

Clients

There are a number of client options for interacting with an HBase cluster.

Java – using java API

MapReduce – just as we process HDFS – HBASE tables act as either source or sink.

REST and Thrift - HBase ships with REST and Thrift interfaces. These are useful when the interacting

application is written in a language other than Java. In both cases, a Java server hosts an instance of the HBase client brokering REST and Thrift application requests into and out of the HBase cluster.

Building a query system

The following steps are used to query HBase data in Apache Drill.

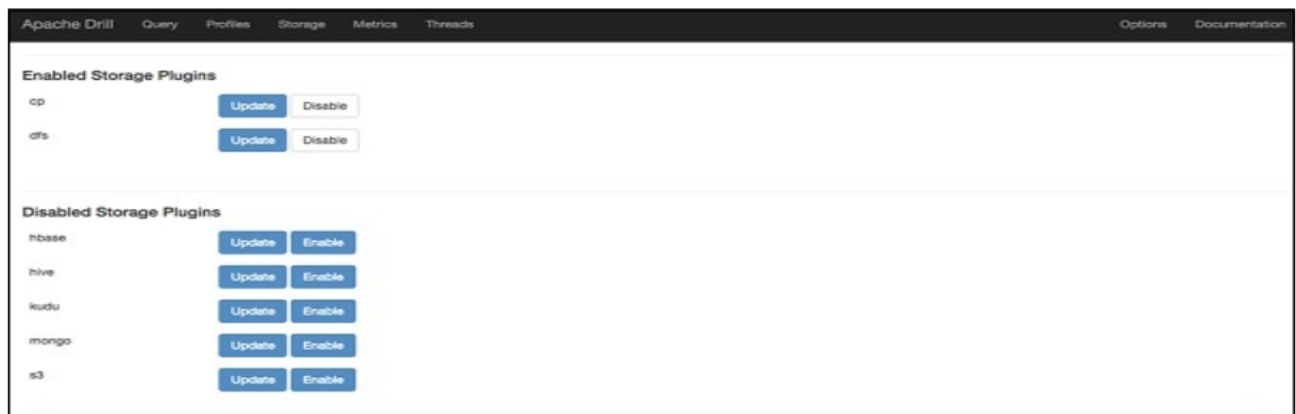
Step 1: Prerequisites

Before moving on to querying HBase data, you must need to install the following –

- Java installed version 1.7 or greater
- Hadoop
- HBase

Step 2: Enable Storage Plugin

After successful installation navigate to Apache Drill web console and select the storage menu option as shown in the following screenshot.



Then choose HBase Enable option, after that go to the update option and now you will see the response as shown in the following program.

```
{
  "type": "hbase",
  "config": {
    "hbase.zookeeper.quorum": "localhost",
    "hbase.zookeeper.property.clientPort": "2181"
  },
  "size.calculator.enabled": false,
  "enabled": true
}
```

Step 3: Start Hadoop and HBase

After enabling the plugin, first start your Hadoop server then start HBase.

Create a Table

Pipe the following commands to the HBase shell to create a “customer” table.

Query

```
hbase(main):001:0> create 'customers','account','address'
```

Load Data into the Table

Create a simple text file named “hbase-customers.txt” as shown in the following program.

Example

```
put 'customers','Alice','account:name','Alice'  
put 'customers','Alice','address:street','123 Ballmer Av'
```

Now, issue the following command in hbase shell to load the data into a table.

Query

```
hbase(main):001:0> cat ../drill_sample/hbase/hbase-customers.txt | bin/hbase
```

Query

Now switch to Apache Drill shell and issue the following command.

```
0: jdbc:drill:zk = local> select * from hbase.customers;
```

Result

row_key	account	address
416C696365	{"name":"QWxpY2U="}	{"state":"Q0E=","street":"MTIzIEJhbGxtZXIgc0E="}
426F62	{"name":"Qm9l"}	{"state":"Q0E=","street":"MSBJbmZpbml0ZSBMb29l"}
4672616E6B	{"name":"RnJhbms="}	{"state":"Q0E=","street":"NDM1IFdhdG9lIG91dG8="}
4D617279	{"name":"TWfyeQ="}	{"state":"Q0E=","street":"NTYgU291dGhlcmlk"}

Apache Drill fetches the HBase data as a binary format, which we can convert into readable data using **CONVERT_FROM** function available in drill. Check and use the following query to get proper data from drill.

Query

```
0: jdbc:drill:zk = local> SELECT CONVERT_FROM(row_key, 'UTF8') AS customer_id
. . . . . > CONVERT_FROM(customers.account.name, 'UTF8') AS custo
. . . . . > CONVERT_FROM(customers.address.state, 'UTF8') AS cust
. . . . . > CONVERT_FROM(customers.address.street, 'UTF8') AS cus
. . . . . > CONVERT_FROM(customers.address.zipcode, 'UTF8') AS cu
. . . . . > FROM hbase.customers;
```

Result

```
+-----+-----+-----+-----+-----+
| customer_id | customers_name | customers_state | customers_street | customers_zipcode |
+-----+-----+-----+-----+-----+
| Alice       | Alice         | CA              | 123 Ballmer Av  | 12345             |
| Bob         | Bob           | CA              | 1 Infinite Loop | 12345             |
| Frank       | Frank         | CA              | 435 Walker Ct   | 12345             |
| Mary        | Mary          | CA              | 56 Southern Pkwy | 12345             |
+-----+-----+-----+-----+-----+
```

HBase Versus RDBMS

Both RDBMS and HBase, both are database management systems. RDBMS uses tables to represent data and their relationships. HBase is a column-oriented dbms and it works on top of Hadoop Distributed File System (HDFS).

Following are the important differences between RDBMS and HBase.

Sr. No.	Key	RDBMS	HBase
1	Definition	RDBMS stands for Relational DataBase Management System.	HBase has no full form.
2	SQL	RDBMS requires SQL, Structured Query Language.	HBase does not need SQL.
3	Schema	RDBMS has a fixed schema.	HBase has no fixed schema.
4	Orientation	RDBMS is row oriented.	HBase is column oriented.
5	Scalability	RDBMS faces problems in scalability.	HBase is highly scalable.
6	Nature	DBMS is static in nature.	HBase is dynamic in nature.
7	Data Retrieval	RDBMS data retrieval is slow.	HBase data retrieval is fast.
8	RULE	RDBMS follws ACID(Atomicity, Consistency, Isolation and Durability) Rule.	HBase follows CAP(Consistency, Availability, Partition-tolerance) Rule.
9	Data structure	RDBMS handles structural data.	HBase handles structural, non-structural and semi-structural data.
10	Sparse Data Handling	Sparse data handling is not present.	Sparse data handling is present.

Praxis

Some of the common issues users run into when running an HBase cluster under load.

HBase's use of HDFS is very different from how it's used by MapReduce. In MapReduce, generally, HDFS files are opened with their content streamed through a map task and then closed. In HBase, datafiles are opened on cluster startup and kept open so that we avoid paying the costs associated with opening files on each access. Because of this, HBase tends to see issues not normally encountered by MapReduce clients:

Running out of file descriptors

Because we keep files open, on a loaded cluster it doesn't take long before we run into system- and Hadoop-imposed limits. For instance, say we have a cluster that has three nodes, each running an instance of a datanode and a regionserver, and we're running an upload into a table that is currently at 100 regions and 10 column families. Allow that each column family has on average two flush files. Doing the math, we can have $100 \times 10 \times 2$, or 2,000, files open at any one time. Add to this total other miscellaneous descriptors consumed by outstanding scanners and Java libraries. Each open file consumes at least one descriptor over on the remote data-node.

The default limit on the number of file descriptors per process is 1,024. When we exceed the filesystem ulimit, we'll see the complaint about "Too many open files" in logs, but often we'll first see indeterminate behavior in Hbase.

To fix this the solution is to increase the upper limit on the number of file descriptors. Usually, this value is 10,240.

UI

HBase runs a web server on the master to present a view on the state of your running cluster. By default, it listens on port 60010. The master UI displays a list of basic attributes such as software versions, cluster load, request rates, lists of cluster tables, and participating regionservers.

Click on a regionserver in the master UI, and you are taken to the web server running on the individual regionserver. It lists the regions this server is carrying and basic metrics such as resources consumed and request rates.

Metrics

Hadoop has a metrics system that can be used to emit vitals over a period to a context. Enabling Hadoop metrics, will give you views on what

is happening on your cluster, both currently and in the recent past. HBase also adds metrics of its own—request rates, counts of vitals, resources used. These will be available in

hadoopmetrics2-hbase.properties

under the HBase conf directory.