Database optimization on Spark

1. Introduction

1.1 Background

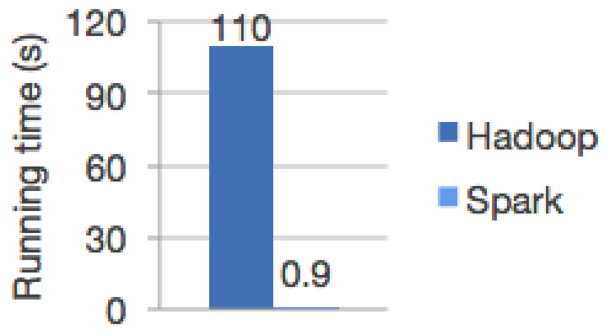
Since nowadays the amount of the data on the internet grows much larger than before, it is necessary to use some corresponding platform to compute and process such big data.

Spark is such a platform for large-scale data processing. It’s much faster than the former used engines and is a very general tool which can be used for database, machine learning, image processing.

Similarly, our Acemap also need to handle big data, therefore our Database group began to construct a Spark ecosystem, aiming to firstly using Spark to accelerate the querying speed of database and then further using it for machine learning.

1.2 Spark and Hadoop

Spark learned from the Mapreduce of Hadoop(another platform which is widely used before), borrowing the advantages of its distributed parallel computing. However, Spark performs much better than Hadoop, because it puts the intermediate data in the memory instead of the disks to largely increase the efficiency of data processing. It also gives much more operations for data set while Hadoop only has ‘map’ and ‘reduce’, so that it can be used more generally and easily.



1.3 Spark and MySQL

MySQL doesn’t perform well in many tasks because it uses only one CPU core for a single query. It means that though you have many fast CPU cores and a big data set, you can’t fully use their computing ability. Spark on the contrary can use all those cores. I will talk about how I fully use those cores later.

1.4 Goal

Our goal is to test the performance of Spark in accelerating querying speed and do some optimizations. A further goal is to try machine learning using Spark.

2. Environment

2.1 Hadoop HDFS

Though we don’t use Hadoop for data processing, we use the HDFS of

Hadoop. It’s a distributed file system which is safe and easy to use. We firstly migrate some table to the HDFS from the database, and then using Spark to query the data from HDFS.

2.2 Spark

We use two components of Spark in this project:

1. SparkSQL, a SQL component of Spark, which uses the techniques such as In-Memory Columnar Storage, bytecode generation to enhance the querying performance.
2. MLLib, a machine learning library on Spark.

Spark is a distributed system so we should install and configure Spark on each server. The installation and configuration steps are :

1. Install Scala.
2. Compile the source code of Spark or simply use the pre-build Spark.
3. Generate SSH key pairs and send public key to each other server. So that each server can communicate between each other.
4. Install Spark on the master server, and two slave servers.
5. Do some configurations for example the scala path and hadoop address on the master server, and copy the configuration files to other servers by SSH.
6. Start Spark and use ‘jps’ to monitor the processes.

With above steps we can construct and run a basic Spark system and advanced configurations will be talked later.

3. Pre-work of data migration

3.1 Brief introduction of sqoop

Sqoop is a tool designed to transfer data between Hadoop and relational databases or mainframes. You can use Sqoop to import data from a relational database management system (RDBMS) such as MySQL or Oracle or a mainframe into the Hadoop Distributed File System (HDFS), transform the data in Hadoop MapReduce, and then export the data back into an RDBMS.

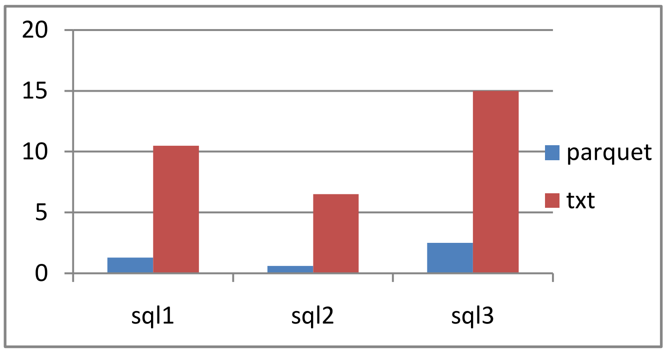
Since we need firstly migrate some tables of the database from MySQL to Hadoop, sqoop is a good tool to help us.

3.2 Parquet and txt

Sqoop supports migrating data from RDBMS to HDFS as different file formats. While we use different types of files as the source data for Spark, the performance varies. So in order to choose the best file format, I checked some information and had some experiments. I finally chose parquet file for following reasons:

1. SparkSQL is much quicker using parquet than txt.

As the user’s guide of spark shows, using parquet can increase tenfold the querying performance of using txt. I tried a few querying to test and verify this result.



2. Less disk space.
   Parquet with compression capabilities can reduce data storage by an average of 75%.

4. Implementation

This part discusses the implementation of SparkSQL in two different methods. The optimization is also included.

4.1 Using operations for RDD

As I mentioned before, Spark supports a lot of operations on its data set. For example, ‘map’ and ‘filter’ can be used to build a new data set from an old one, and ‘sample’, ‘groupbykey’ and some other operations can be used to process a data set.

There are some simple examples:

```
// Select people older than 21
df.filter($"age" > 21).show()
// +---+----+
// |age|name|
// +---+----+
// | 30|Andy|
// +---+----+

// Count people by age
df.groupBy("age").count().show()
// +----+-----+
// | age|count|
// +----+-----+
// |  19|    1|
// |null|    1|
// |  30|    1|
// +----+-----+
```

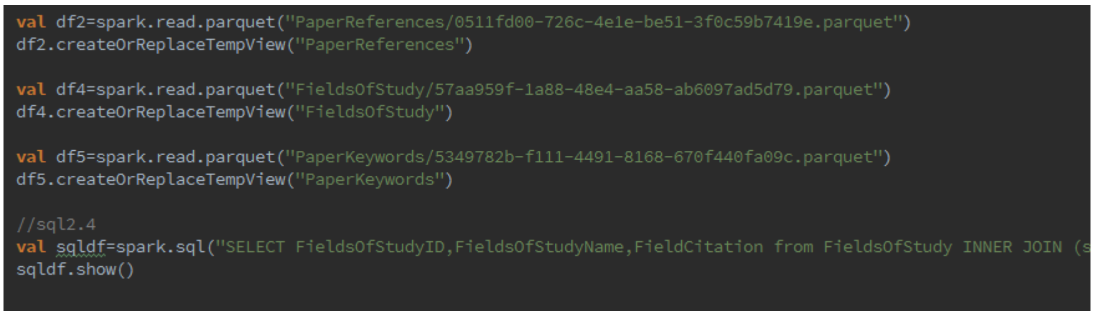
One query in Acemap for finding the paper that references an author and the number of times the paper references the author can be implemented as:

```
df1.union(df2.filter("AuthorID=>AuthorID='0C7733DB'").count()
```

4.2 Using translator

SparkSQL has a translator for SQL sentences, which can help translate SQL into Spark language. With this feature, users will be free from learning those operations for Spark and finding the best operations for a query. So anyone who is familiar with SQL can easily use the SparkSQL, though it will take a little time for translating.

The implementation is quite simple:



```
val df2=spark.read.parquet("PaperReferences/0511fd00-726c-4e1e-be51-3f0c59b7419e.parquet")
df2.createOrReplaceTempView("PaperReferences")

val df4=spark.read.parquet("FieldsOfStudy/57aa959f-1a88-48e4-aa58-ab6097ad5d79.parquet")
df4.createOrReplaceTempView("FieldsOfStudy")

val df5=spark.read.parquet("PaperKeywords/5349782b-f111-4491-8168-670f440fa09c.parquet")
df5.createOrReplaceTempView("PaperKeywords")

//sql2.4
val sqldf=spark.sql("SELECT FieldsOfStudyID,FieldsOfStudyName,FieldCitation from FieldsOfStudy INNER JOIN (s
sqldf.show()
```

4.3 Run spark project

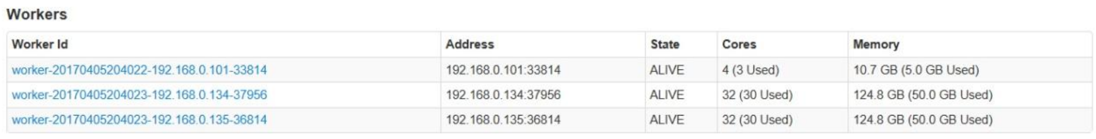
1. spark-shell: A shell interface for users to test their codes. In this mode, project is running stand-alone.

2. spark-submit: It helps us submit the project to cluster, which means the project is running on all the servers.

4.4 Optimization

We talked about some configurations for building a basic Spark system, but still there are some advanced configurations to reach a better performance. I also did some experiments to decide the how to configure.

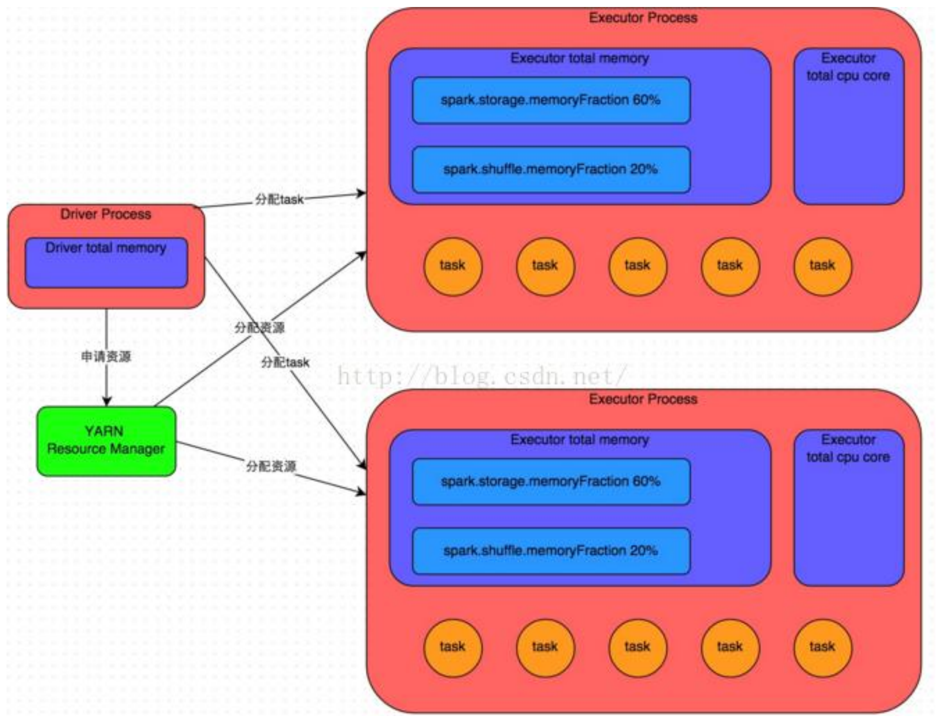
The resources of our servers(4 cores and 10.7GB memory for the master server, 32 cores and 124.8GB memory for two slave servers):



Workers

Worker Id	Address	State	Cores	Memory
worker-20170405204022-192.168.0.101-33814	192.168.0.101:33814	ALIVE	4 (3 Used)	10.7 GB (5.0 GB Used)
worker-20170405204023-192.168.0.134-37956	192.168.0.134:37956	ALIVE	32 (30 Used)	124.8 GB (50.0 GB Used)
worker-20170405204023-192.168.0.135-36814	192.168.0.135:36814	ALIVE	32 (30 Used)	124.8 GB (50.0 GB Used)

When we use ‘spark-submit’ to submit a Spark project, a driver process is started, it will start some executor processes. In a Spark job, driver process responsible for resources allocation and executor processes perform the tasks. The number of executors and how many cores a single executor has, are determined by our configuration.



1. number of executors & executor memory

   The number of executors is a very important parameter. If the number is too small, we can't fully use our resources, and on the other hand, some executors may not have enough resources if the number is too big. Executor memory influences the performance of a single executor. When we allocating the memory, we should make sure the total executor memory doesn't go beyond the total memory of the server.

   I finally choose to set 20 executors and 10GB memory for each executor.
2. executor cores

   Executor cores determine the ability for an executor to process tasks parallel. The more cores an executor has, the less time it will take to finish its own task.
3. parallelism

   Parallelism determines the tasks number. It has direct influence on the performance of your Spark project, because if the number is two small, many executor will not get a task, so no matter how will you configure the number of executors and the executor cores, your efforts are useless.

   I finally set the number as 200, so that those executors and CPU cores are fully used.

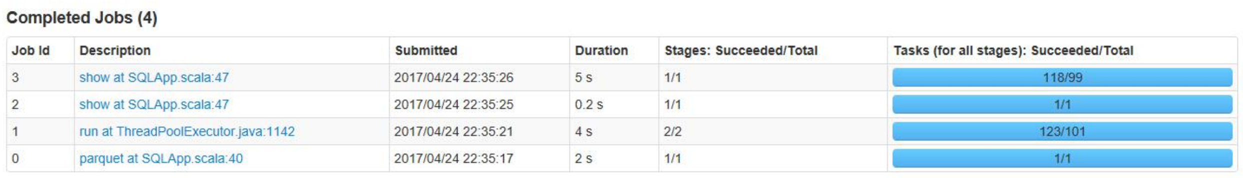
5 Results & Analysis

5.1 Results

Because MySQL performs well itself on simple querying, and Spark is designed to process big data sets, our optimization mainly focused on complicated querying which either uses very large data sets or need to do difficult computing.

I optimized these 5 querying:

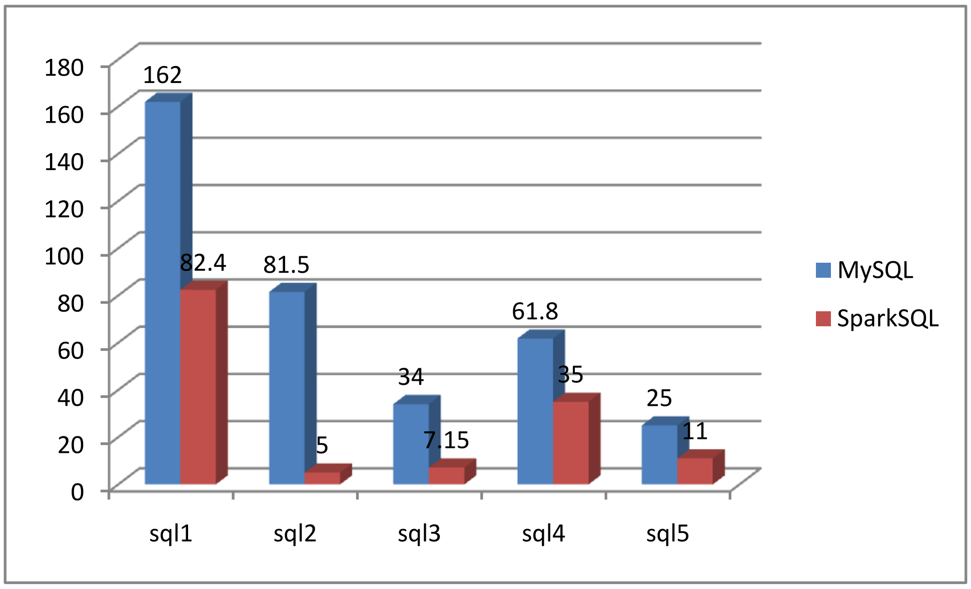
1. sql1: find the papers that reference an author and the number of times the papers reference the author
2. sql2: find 1000 most relevant study fields to a certain field, and show their relationships.
3. sql3: find out 50 of the most cited authors in a field.
4. sql4: find out the numbers of a paper be sited in each study field.
5. sql5: Shows the number of papers of an author, and the total number of citations.



Completed Jobs (4)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
3	show at SQLApp.scala:47	2017/04/24 22:35:26	5 s	1/1	118/99
2	show at SQLApp.scala:47	2017/04/24 22:35:25	0.2 s	1/1	1/1
1	run at ThreadPoolExecutor.java:1142	2017/04/24 22:35:21	4 s	2/2	123/101
0	parquet at SQLApp.scala:40	2017/04/24 22:35:17	2 s	1/1	1/1

	MySQL	SparkSQL	SparkSQL-optimized
Sql1	162	120	82.4
Sql2	81.5	25	5
Sql3	34	18	7.1
Sql4	61.8	44	35
Sql5	25	16	11



5.2 Analysis

For those simple querying, MySQL itself performs well enough that users will not feel those querying are slow. Spark performs equivalent or sometimes little slower with those querying, because Spark may cost some time to apply resources and allocate tasks. However, when we deal with large data sets, Spark enhances its performance impressively. As the result show above, for those complicated querying Spark is average 5 times quicker, thanks to those features, for example, putting intermediate data in memory and fully using the cores of CPU to compute.