**Indian Institute of Information Technology-Allahabad**



PROJECT

SEMESTER – VIII

**Distributed framework for back-testing and research analytics**

Submitted By:

Jai Prakash Yanamadala

IIT2013068

**CANDIDATES’ DECLARATION**

I hereby certify that the work which is being presented in the B.Tech. Project Report entitled “**Distributed framework for back-testing and research analytics**”,being submitted asa part of Semester Project Evaluation to the Department of Information Technology of Indian Institute of Information Technology, Allahabad, is an authenticated record of my original work in **Edelweiss Financial Services Ltd** from February 2017 to July 2017.

**Date:** 25th July 2017

**Place**: IIIT-Allahabad

Jai Prakash Yanamadala

**Table of Content**

1. Abstract …………………………………………………………………………
2. Introduction
3. Problem Definition

**ABSTRACT**

The financial sector has gone through unprecedented change in the last few years, primarily with advent of Big Data. Big Data, in the perspective of financial sector, can be viewed as quantitative methods that refer to the collection and analysis of massive amounts of information.

Big data is especially promising and differentiating for financial services companies. With no physical products to manufacture, data is one of arguably their most important assets. The business of banking and financial management is rife with transactions, conducting hundreds of millions daily, each adding another row to the industry’s immense and growing ocean of data. So the question for many of these firms remains how to harvest and leverage this information to gain a competitive advantage.

These firms, especially firms dealing in market trading heavily rely on back testing and analytics, for the creation of strategies and testing these strategies for authenticity and profit making ability. The performance and the ability to judge the optimal strategy greatly increase, as the dataset utilized in the build and back test strategies increase. Thus, handling Big Data is of at most importance for these firms.

The traditional methods of data collection and analytics fail to compete with the sheer velocity, volume and variety of this data, thus it is required to design a system that can handle these humongous amounts of data. The best and optimal solution to tackle this problem is the creation of distributed systems that are fast, fault-tolerant and resilient. This, project aims to build such a system of distributed network, where in humungous amounts of data can be stored, processed and analysed.

**INTRODUCTION**

At Global Markets we deal with various kinds of Algorithmic and Systematic Trading. Systematic trading is a way of defining trade goals, risk controls and rules that can make investment and trading decisions in a methodical way. Algorithmic trading is pre-programmed trading instructions to buy/sell shares to achieve maximum profits. Algorithmic and Systematic Trading identify futures by analysing models, trends, patterns across days or months or years, in various forms of trading like high-frequency trading, day trading, low-frequency trading etc . This requires deep analysis of large historical and live streaming data, which comprises of variables like date, product type, stock symbol etc.

The analysis of such large data can't be achieved on a single system. It requires the use of a distributed system.

A distributed system is a model in which components located on networked computers communicate and coordinate their actions by passing messages. The components interact with each other in order to achieve a common goal.

This project aims to design and implement such a distributed system which can be effectively utilized for trade research analytics and back testing of trade strategies.

The Distributed system will utilize various big data technologies such as Hadoop, HDFS(Hadoop Distributed File System)YARN, HIVE, Spark, Zeppelin etc., for the purpose of back testing and research analytics. The project also involves the creation of various task specific applications to be run on top of this distributed system, rapid data transfer rates among nodes and fault-tolerance that allows the system to continue operating in case of a node failure, thus rapidly increasing the business scalability.

**PROBLEM DEFINITION**

The aim of this project is to create a distributed system framework for back testing and research analytics. This project utilizes the various concepts of Hadoop framework like map-reduce, HDFS(Hadoop distributed file system),SPARK, TEZ, data warehousing, prestoDB ,LLAP etc. to effectively and optimally run thousands of terabytes of data on thousands of commodity hardware nodes, producing results in the shortest amount of time possible and handle node failure.

**METHODOLOGY**

**The project can be broadly divided into four stages, namely:**

1. Setting up the software for creating the framework
2. Creating a Distributed Data warehouse.
3. Distributed Framework to run simulations and strategies.
4. Data Visualization for research analytics and result verification.
5. **Hadoop and Yarn:**

The entire frame work was built on top of Hadoop and Yarn was used as the resource manager, as they provide means of scalability and mechanisms to handle huge sets of data.

* 1. **Hadoop**:

Hadoop is an open source programming framework that supports the processing and storage of extremely large data sets in a distributed computing environment.

Hadoop runs applications using the Map Reduce algorithm, where the data is processed in parallel on different CPU nodes. In short, Hadoop framework is capable enough to develop applications capable of running on clusters of computers and they could perform complete statistical analysis for huge amounts of data.

* + 1. **Map-Reduce:**

Hadoop Map-Reduce is a software framework for easily writing applications which process vast amounts of data (multi-terabyte data-sets) in-parallel on large clusters (thousands of nodes) of commodity hardware in a reliable, fault-tolerant manner.

As illustrated in the **Figure 1** Map-Reduce job usually splits the input data-set into independent chunks which are processed by the map tasks in a completely parallel manner. The framework sorts the outputs of the maps, which are then input to the reduce tasks. Between the map and reduce stages, usually the map-reduce goes through a shuffle stage, where the mapped data is further mapped into, various sets depending on the job at hand, this is done to achieve faster reduce and optimal usage of parallelism.

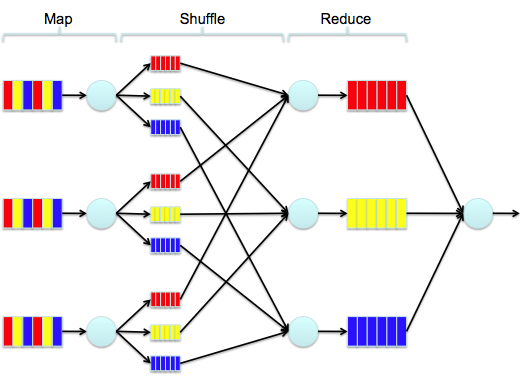


Figure 1

* 1. **YARN**:

YARN is a resource manager, it is the architectural centre of Hadoop that allows multiple data processing engines such as interactive SQL, real-time streaming, data science and batch processing to handle data stored in a single platform. The Figure 2, illustrates Yarn,

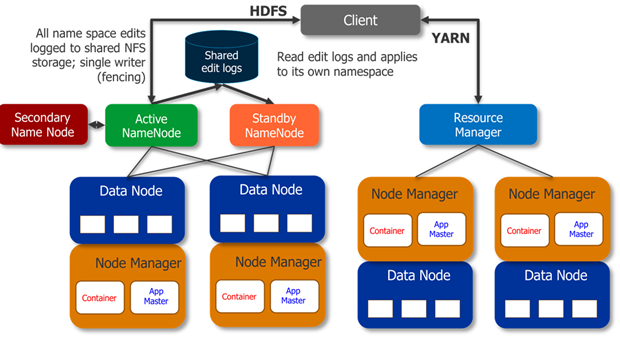


Figure 2

1. **Creating a Distributed Data warehouse:**

The first step of the project was to create a central data warehouse, where huge amounts of data (terabytes of data) can be stored in a distributed fashion. Different forms of data have to be accumulated into a single distributed framework, where in the data can be queried effectively and in the shortest amount of time, without the loss of any data.

Various forms of data like tbtfeed (tick by tick data of trades), snapshot data(data aggregated in snapshots of time), the data generated by the various simulation and strategies , was stored using HDFS file system and hive framework.

HDFS (Hadoop distributed file system) is a scalable, fault-tolerant, distributed storage system data storage system based on java. HDFS uses a client/server architecture. A HDFS cluster consists of two masters: Name node and Resource manager, multiple Data nodes and is accessed by many clients.

* 1. **Client:**

A client is an api of applications. It communicates with the Name node because of metadata and after receiving them, it directly runs operations on the Data nodes. If the operation is a MapReduce operation, the client creates a job and sends it to the queue. The Job Tracker handles this queue.

* 1. **Name node**:

Name node is the master server which maintains all file system metadata like the namespace, access control information, the mapping from files to blocks and the current location of blocks. Block locations are not stored on the Name node permanently, it collects by asking Data nodes while starting up or when a new Data node is connected to the cluster .Basing on the system resources and the input file size the Name node decides which Data nodes the clients should connect and responds this information to the Client.

* 1. **DataNode**:

A Data node doesn’t store all files in the same directory, it uses a heuristic to calculate which number of files is best for the local file system and creates subdirectories suitably. Secondary Name node Modifications to the file system are stored as a log file by the Name node.

Apache Hive is a data warehousing solution for Hadoop which provides data summarization, query, and ad-hoc analysis. It is used to process structured and semi-structured data in Hadoop. It is a layer between the user and HDFS, through which the user can store and query results.

It stores the data in a RDMS kind of fashion and uses a query language similar to SQL.

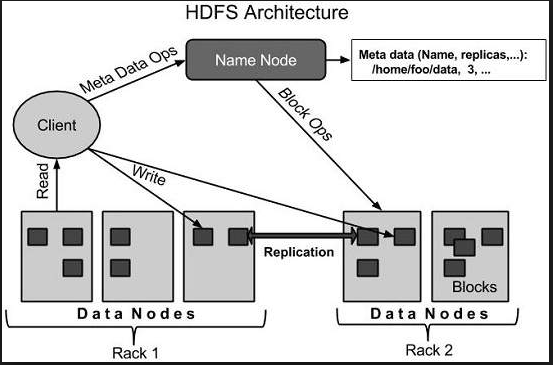


Figure 3

At its core it uses map-reduce algorithms to make effective data storage and retrieval. The hive architecture can be described from the image below:



Figure 4

Hive can be broadly divided into the following components, the metastore, the compiler, the driver and the execution engine.

The metastore is an abstraction of data, where in it contains the various aspects of data, the file format the data is stored, the compression format of the data, the way the data is partitioned, the fields of the data, the data types etc.

The driver interacts with the UI and breaks down the query submitted into a map reduce plan. It then sends the plan to the complier. The compiler analyses the plan and retrieves the required metastore information and sends it back to the driver.

The execution engine implements the plan. The execution engine is configurable and various map-reduce paradigms like TEZ, Spark etc. can be used as execution engine.

The driver program uses various algorithms to achieve effecting map-reduce plan and joins. Hive uses various forms of aggregated mapping before proceeding to reduce stage. We used map-side for handling various aggregations and group by.

Unlike traditional map-reduce approach where in the entire data read by the map is stored and a join key is passed down, mapSide moves the data into a hash table file to the Hadoop distributed cache, which populates these files to each mapper’s local disk. So all the mappers can load this persistent hash table file back into the memory and do the join work as before. The execution flow of the optimized map join is shown in the figure below. After optimization, the small table needs to be read just once. Also if multiple mappers are running on the same machine, the distributed cache only needs to push one copy of the hash table file to this machine.

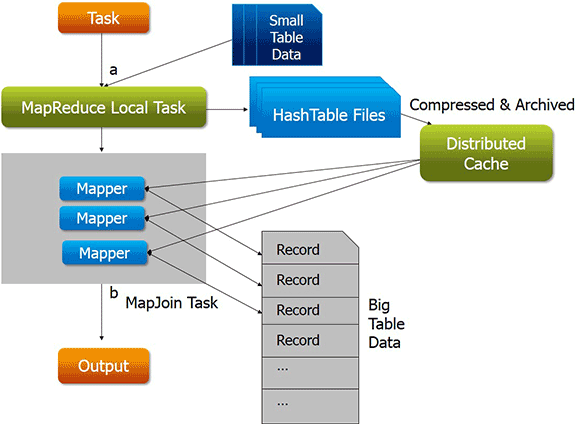
­

Figure 5

The fig.6, illustrate the different ways in which map-side was utilized:

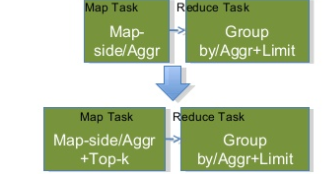
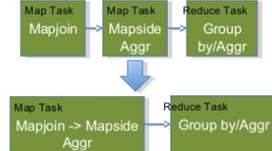


Figure 6

The hive execution engine is responsible for implementing the map-reduce plan that the driver designs, initially traditional map-reduce was used as the execution engine for hive.

Unlike Spark, Tez is not limited to simply map and reduce, but could be used to develop complex networks of Map and Reduce tasks. Ideas like a set of job tasks that do Map then Reduce and Reduce again could not be expressed previously. More importantly, Tez allowed for the intermediate results of tasks to go directly to the next task skipping the dreaded write to disk step that was so costly when processing big data. Subsequent repeated identical queries would run dramatically faster since they did not incur the cost of a new container launch, as TEZ could handle container delay. Containers were reused allowing shared access to data along with lower latency queries again due to the lack of container setup time. These changes lead to a significant improvement in hive query performance, as compared to traditional map-reduce, which have been illustrated in the fig.7.

The performance of hive also depends upon the partitioning of hive table and the file formats used,

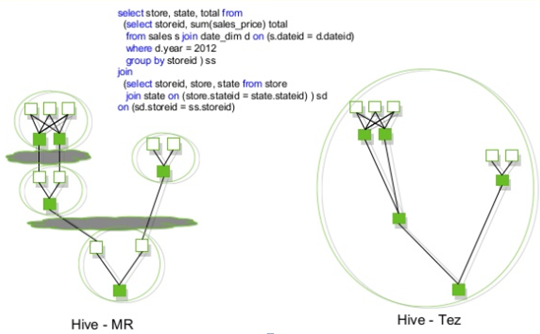


Figure 7

In Hive’s partitioning method, all the data present in a table is divided into multiple partitions. Each partition corresponds to a specific value(s) of partition column(s) and is kept as a sub-record inside the table’s record present in the HDFS. So on querying a particular table, appropriate partition of the table is queried which contains the query value. Hence it decreases the I/O time required by the query which increases the performance speed.

Hive supports various file formats, the file formats and the partitions where decided depending upon the data that was to be stored.

For tick-by-tick data, where in every trade that occurs in a day was stored the data format that was used was ORC, file format as it was useful in aggregations and range selections, the partition used was the date as most of the analytics are to be done, grouping the data into dates.

The orc data has been illustrated shows, the stripe footer contains a directory of stream locations. Row data is used in table scans. Index data includes min and max values for each column and the row positions within each column. (A bit field or bloom filter could also be included.) Row index entries provide offsets that enable seeking to the right compression block and byte within a decompressed block. Note that ORC indexes are used only for the selection of stripes and row groups and not for answering queries.

Having relatively frequent row index entries enables row-skipping within a stripe for rapid reads, despite large stripe sizes. By default, every 10,000 rows can be skipped.

With the ability to skip large sets of rows based on filter predicates, you can sort a table on its secondary keys to achieve a big reduction in execution time. For example, if the primary partition is transaction date, the table can be sorted on state, zip code, and last name. Then looking for records in one state will skip the records of all other states.

Orc data has the features of projection pushdown, filter pushdown and aggregate pushdown that enable effective and fast aggregate queries.

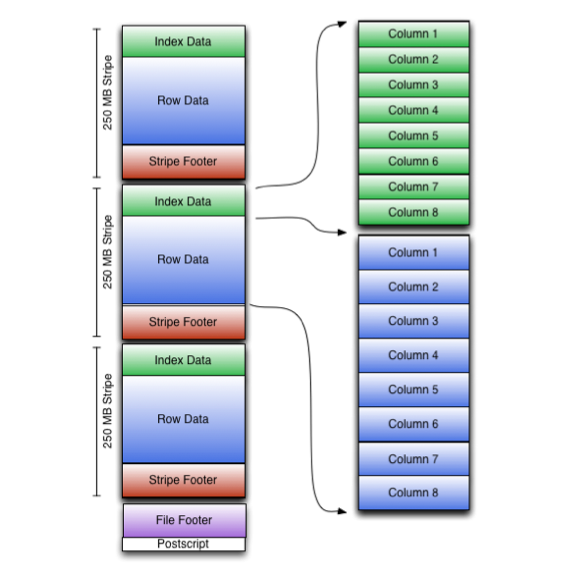


Figure 8

For the simulation data where in the parameters of the simulation can vary according to each data making it a dynamically irregular data, the parameters and the results where mapped and a map based columnar approach was used, this can be illustrated by the fig 8.

The file format used for this purpose was parquet.

Parquet is a columnar data structure, where in instead of just storing rows of data adjacent to one another you also store column values adjacent to each other. So datasets are partitioned both horizontally and vertically. This is particularly useful in or case as the entire map column can be stored at a singular cluster, thus leading to a better query speed.

The pictorial representation of parquet file format has been illustrated in fig 9.

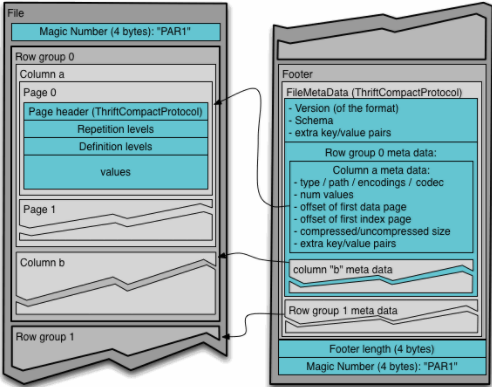


Figure 9

The performance measure of parquet upon other file formats, for a map column based data, has been illustrated in the <fig no:><to add>

Of the different compression formats available, we used AVRO based compression as it helps in speeding up the serialization and de-serialization of data. Thus, reducing the data retrieval time.

The performance measure of AVRO upon various file compressions has been illustrated in the <fig no:><to add>

To sped up the interactive queries that are frequently queried across researchers continuously, we used LLAP as a middle layer. Where in traditionally, the map reduce reads from the disk, LLAP acts as middle layer buffer, where in it stores the data, thus reducing I?O calls and thus increasing the speed.

As, the volume of data increased, the query speed started to reduce as it had to search a much larger data store, the difference of performance has been illustrate in <fig no:><to add>

To tackle this problem, the different partitions of data had to be parallel fetched , to enable this we used presto on top of hive, which enables us to execute the query parallel across clusters.

The DAG between a traditional hive query and a presto query has been illustrated in fig <fig no:><to add>.

1. **Distributed Framework to run Simulations and Strategies:**

After the creation of the central data base the next step was to create a distributed framework, where in different simulation processes and Strategies could be plugged in.

For this purpose we had to design and build a map-reduce plan, which could then be deployed onto the Hadoop distributed system. We have utilized java with spark to design and implement this.

* 1. **Spark:**

Spark is a fast, in-memory data processing engine with elegant and expressive development APIs to allow data workers to efficiently execute streaming, machine learning or SQL workloads that require fast iterative access to datasets. With Spark running on Apache Hadoop YARN, developers everywhere can now create applications to exploit Spark’s power, derive insights, and enrich their data science workloads within a single, shared dataset in Hadoop.

Spark enables us to use distributed collection data-structure, know as Resilient distributed datasets (RDD), to make references to data, which can then be used in map-reduce jobs. Resilient Distributed Datasets (RDD) is a fundamental data structure of Spark. It is an immutable distributed collection of objects. Each dataset in RDD is divided into logical partitions, which may be computed on different nodes of the cluster. RDDs can contain any type of Python, Java, or Scala objects, including user-defined classes. There are two ways to create RDDs − parallelizing an existing collection in your driver program, or referencing a dataset in an external storage system, such as a shared file system, HDFS, HBase, or any data source offering a Hadoop Input Format.

Traditionally the map-reduce jobs are slow on interactive and iterative queries, due to replication, serialization and disk I/O, RDD solves this bottle necks and speeds up the process by 90%. The difference of approach between traditional map-reduce approach and the approach through spark RDD have been illustrated in the below figures10 and 11

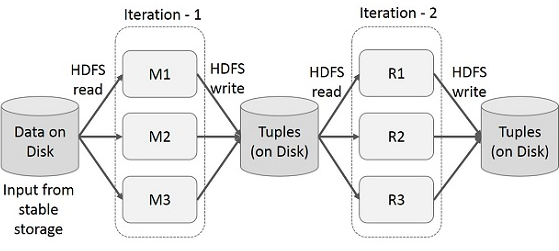


Figure 10

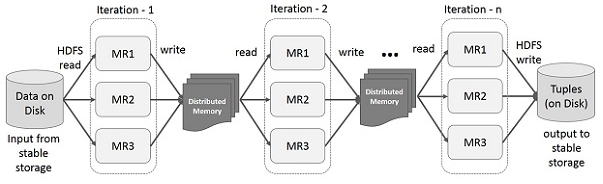


Figure 11

RDD enables to store the data on a Distributed Memory, instead of frequently calling the dataset, this distributed memory, it stores this state of memory as an object across the jobs and the object is sharable between those jobs. If the data cannot be stored in a Distributed data of the clusters ram, then the additional data is spilled on to the disk space. We can also persist the RDD in memory thus saving re-computation time.

* 1. The stages of an RDD can be broadly divided into two categories:
     1. Transformations.
     2. Actions.

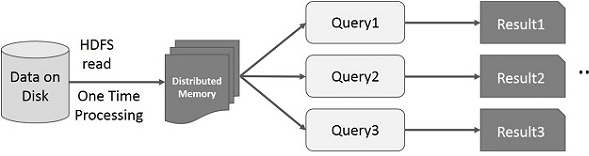
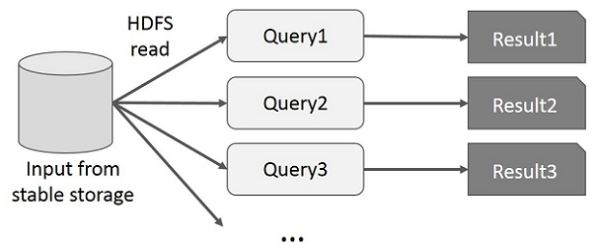


Figure 12

Transformations are the process that transform RDD from one form to another, for example if we have a RDD which is a un-partitioned dataset and we apply a process on it that transforms into a RDD with a Tuple, where in the tuple consists of pair corresponding to the data and a index, then we say that the RDD has been transformed from a un-partitioned RDD to a RDD of tupels.

RDD transformations returns pointer to new RDD and allow you to create dependencies between RDDs. Each RDD in dependency chain (String of Dependencies) has a function for calculating its data and has a pointer (dependency) to its parent RDD.

RDD transformation is not a set of data but is a step in a program (might be the only step) telling Spark how to get data and what to do with it.

Actions are the reduce part of the RDD, they are generally the last steps in a specific map-reduce RDD task, once spark reaches this stage, it completes the creation of plan and starts the execution. From the fig.13, we can see that there can be several number of transformations before an action takes place.

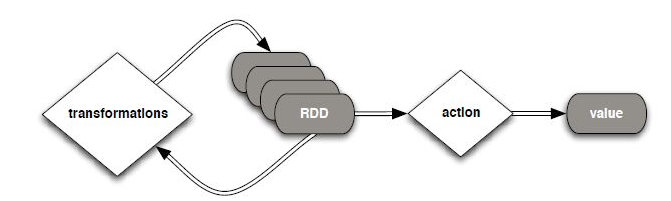


Figure 13

Spark is based on lazy programme principle, where in there is no execution unless an action is stated.

This means that the programme doesn’t start to execute as soon as it is started, the Spark first scans through the entire programme and then takes into account all the transformations that have been called, it then creates a plan of execution in form of a DAG. For, example for the following code snippet in fig<fig no:>, the resulting plan of execution has been shown in fig<fig no:>

The driver is the process that runs the user code that creates RDDs, and performs transformation and action, and also creates SparkContext. When the Spark Shell is launched, this signifies that we have created a driver program. On the termination of the driver, the application is finished.

The driver program splits the Spark application into the task and schedules them to run on the executor. The task scheduler resides in the driver and distributes task among workers. The two main key roles of drivers are:

* Converting user program into the task.
* Scheduling task on the executor.

The structure of Spark program at a higher level is: RDDs are created from some input data, derive new RDD from existing using various transformations, and then after it performs an action to compute data. In Spark Program, the DAG (directed acyclic graph) of operations are created implicitly. And when the driver runs, it converts that Spark DAG into a physical execution plan. The figure 14 shows the timeline view of the various tasks that the spark executes.

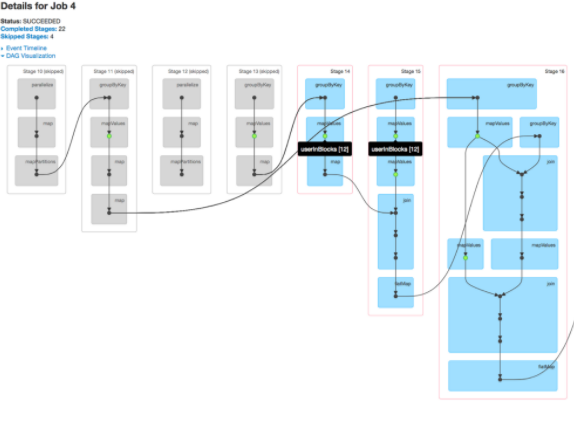


Figure 14

At high level, when any action is called on the RDD, Spark creates the DAG and submits it to the DAG scheduler. The DAG scheduler divides operators into stages of tasks. A stage is comprised of tasks based on partitions of the input data. The DAG scheduler pipelines operators together. For e.g. Many map operators can be scheduled in a single stage. The final result of a DAG scheduler is a set of stages.

The Stages are passed on to the Task Scheduler. The task scheduler launches tasks via cluster manager (Yarn/Standalone). The task scheduler doesn't know about dependencies of the stages.

The Worker executes the tasks on the Slave.

At high level, there are two transformations that can be applied onto the RDDs, namely narrow transformation and wide transformation. Wide transformations basically result in stage boundaries.

Narrow transformation - doesn't require the data to be shuffled across the partitions. for example, Map, filter etc..

wide transformation - requires the data to be shuffled for example, reduceByKey etc..

For example the following code snippet in fig 15, that counts the logs based on severity level is executed as follows :

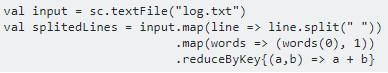


Figure 15

This sequence of commands implicitly defines a DAG of RDD objects (RDD lineage) that will be used later when an action is called. Each RDD maintains a pointer to one or more parents along with the metadata about what type of relationship it has with the parent. For example, when we call val b = a.map() on a RDD, the RDD b keeps a reference to its parent a, that's a lineage.

To display the lineage of an RDD, Spark provides a debug method toDebugString(). For example executing toDebugString() on the splitedLines RDD, will output the following, in fig 16.

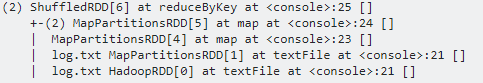


Figure 16

The first line (from the bottom) shows the input RDD. We created this RDD by calling sc.textFile().

Once the DAG is build, the Spark scheduler creates a physical execution plan. As mentioned above, the DAG scheduler splits the graph into multiple stages, the stages are created based on the transformations. The narrow transformations will be grouped (pipe-lined) together into a single stage. So for our example, Spark will create two stage execution as follows, fig 17.

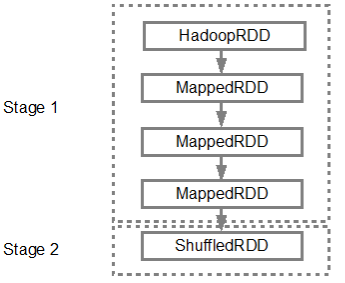


Figure 17

The DAG scheduler will then submit the stages into the task scheduler. The number of tasks submitted depends on the number of partitions present in the textFile. Fox example consider we have 4 partitions in this example, then there will be 4 set of tasks created and submitted in parallel provided there are enough slaves/cores. Below diagram illustrates this in more detail, fig 18:

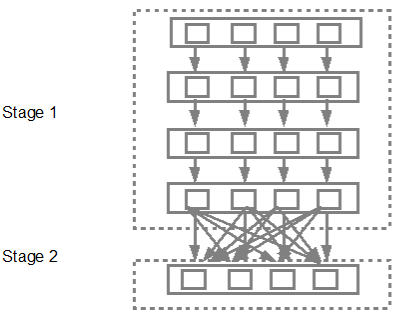


Figure 18

As shown in the above example spark can be used to design an composite map-reduce task, which can be used for various purposes.

The framework to be designed had to have the following functionalities:

* It has to use the data in the central data repository, earlier created and distribute it across various tasks
* Various simulations and strategies can be plugged in, thus creating a generic framework across all simulation and strategies.

The frame work was built using java and spark, initially the data was quired from within spark using hiveQL, then the data was transformed into a RDD of a custom class type, where in the class had the required data representation.

This was the raw data across the required days, this data was then transformed into a RDD of type tuple. Where in, the tuple represented was a pair of key/value, the key begin the timestamp of the data and the value begin the data itself. This transformed data was then mapped into a javaRDD Pair, for this a custom map partition function was developed.

On this RDD various combinations are to be run, to simulate a random data inflow. Each such simulation depends only on the data it takes in and has no other dependency on any other data, so each such data could be mapped as a separate process.

The data from the central data base was distributed into various RDD, based on the test cases and parameters, the test cases where generated through simulations, on which the multiple strategies where run across multiple test cases in parallel.

Traditionally, if there are n number of simulations to be run, the base code of simulation would be run in a loop for n times , changing the parameters of simulation in each iteration. The aim was to achieve this through parallelism rather than iterations.

When the driver initiates a map-job, the map-job is distributed into various clusters(slave nodes), each such executer then assigns various tasks, which are then run parallel, if the number of clusters are smaller than the number of executors, then each cluster is assigned more than one executer.

This can be further illustrated by fig 19.

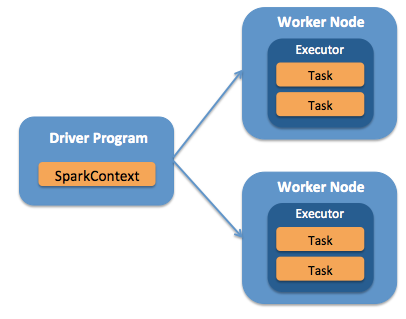


Figure 19

The problem with this approach was, that the executors under a cluster aka worker node, can only performs a single task, as the resources are assigned to the executor, which it then distributes to the tasks, iteratively.

Say, for example we have to run 10 simulations across 3 clusters, with 5 combinations of parameters P1, P2, P3, P4 and P5. When the map-reduce gets executed, let’s say 4 executors were created, now each cluster gets one executor and one cluster gets two executors, under each executor there are a number of tasks, each task inside an executor has to wait till the task before it completes.

This, leads to considerable delay in processing, it was required to speed up this process. To achieve massive parallelization, the tasks inside an executor that where running sequentially had to parallelized.

To achieve this, we introduced the concept of resource sharing inside the executor of a cluster.

Each executor is given a set of resources, which are utilized by the task, in sequential order, these resources are to be put in a common resource sharing paradigm, where in all the tasks inside a executor, can access them parallel.

This, way of sharing resources ensures that, the tasks are not run sequentially, but can be run parallel. This will lead to greater reduction in the time taken to achieve the tasks. This has been illustrated in the fig,<fig No:><to be added> . This parallelization, was achieved by multi-threading the tasks inside the executor, the performance increments achieved, have been illustrated in the fig,<fig No:><to be added>

The next bottle neck, was the I/O calls required to access the data, this was taking a considerable time, as the number of combinations, required for the simulations increased, the number of I/O calls Increased, leading to increase in the time required to run the simulations.

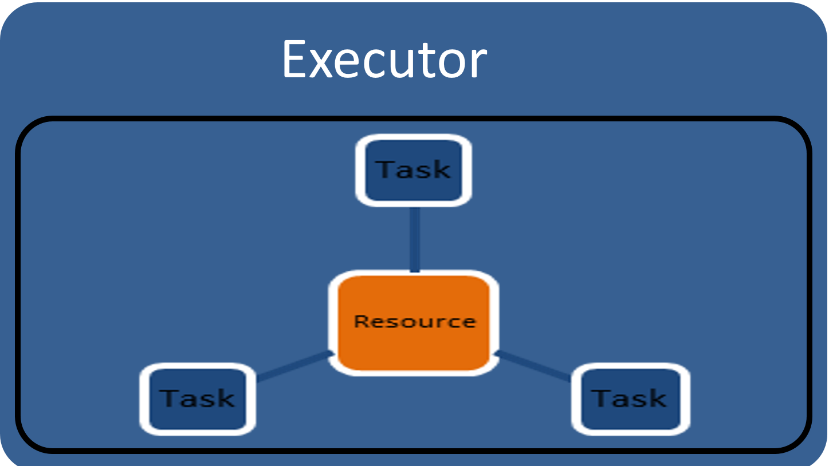


Figure 20

If the parameters of the simulation are same, but the number of combinations of those parameters keep changing, then the data acquired for one simulation, could be used across all the simulations with similar parameters. Hence, a cache layer was created, between the central database and framework, where in the parametric data common across a set of simulations is brought into.

The cache layer was dynamically divided into different partitions, depending upon the set of simulations being run in parallel. Thus, the number of I/O, for n combinations of p parameters, has been reduce from n to 1, as only once the data had to be fetched and cached. If the data is larger than the cache memory, then the data gets spilled on to the disk.

Using all the above mentioned principles and changes to the existing system, an robust, optimal back testing framework was built. The data generated from these simulations was then stored in the central database created partitioned by the date the simulation run, the researcher id of the simulation and was stored in map-columnar format, as mentioned in <Index No:><to be added>

The entire workflow and design of algorithm has been illustrated in the fig<fig No:><to be added>

1. **Data Visualization for research analytics and result verification:**

After building, the framework the next step was to create a visualization framework for research analytics and result verification.

The frame work should have the following functionalities:

* Should be able to retrieve data from the central data base
* Should have the functionality to utilize the data to create various graphs and visualization.
* Should be fast and error free.

For, creating this visualization framework zeppelin and d3 were used.

Zeppelin is a multipurpose incubating web-based notebook, it can be used for running various programs aimed towards data integration, analysis and visualization.

Zeppelin uses a unique approach of interpreter addition, various interpreters can be added to zeppelin, which can then be called on to the notebooks. In a notebook, we can create various paragraphs.

Each such paragraph, can be run on a different interpreter, and the data can be flown from one para to another, thus increasing the flexibility of an user. For, example in the fig<fig No:><to be added>, we can see that one paragraph has the interpreter as hive (on the left, represented by %jdbc(hive)), and the other para has the interpreter as spark, the data has been propagated using zeppelin context.

Zeppelin context is object creation mechanism, where in an object created can be used across various interpreters. To create a zeppelin context object we use, the identifier ‘z’, as illustrated in the figure z.select create the object and z.get gets the object, we can also use z.load() to load objects. They are various other functions that can be used, to propagate and create zeppelin contexts.

Custom interpreters for hive, prestoDB and spark where created and added to zeppelin’s interpreter pool. These, interpreters where then used across notebooks, in various paragraphs to fetch data.

Once the data has been queried, we used were required to provide with the functionality to visualize it. We used Helium for this purpose.

Helium is a visualization resource manager, where in we can plug in are visualization, which the can be integrated into zeppelin, once the visualization had been scripted it can then been plugged into helium and used as a functionality in zeppelin, as shown in the figure<to be added>.

To create various visualizations, D3 was used.

D3 is a data visualization library, short for Data Driven Documents. D3 allows you to bind arbitrary data to a Document Object Model (DOM), and then apply data-driven transformations to the document. For example, you can use D3 to generate an HTML table from an array of numbers. Or, use the same data to create an interactive SVG bar chart with smooth transitions and interaction. The Document Object Model (DOM) is a programming API for HTML and XML documents. It defines the logical structure of documents and the way a document is accessed and manipulated.

As, illustrated by the code snippet<to be added>, d3 was used to create various visualizations that where then integrated into zeppelin, to be used for result verification and data analytics.

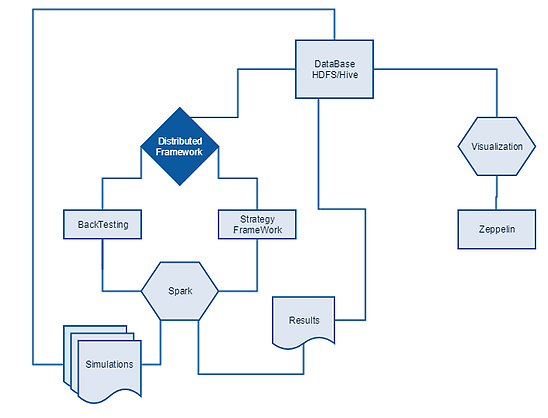


Figure 21

Thus, the entire framework for back-testing and analytics was built.

**Results and Discussion**

