International Journal of Computing and Digital Systems
ISSN (2210-142X)
Int. J. Com. Dig. Sys. 8, No.1 (Jan-2019)

http://dx.doi.org/10.12785/ijcds/080105

Framework for Parallel Processing of Very Large Volumes of Data

ARIDJ Mohammed [1]

[1] *Dept.Computer of Hassiba Benbouali University Chlef Algeria*

Received 12 Nov. 2018, Revised 7 Dec. 2018, Accepted 10 Dec. 2018, Published 1 Jan. 2019

Abstract: Many scientific fields are now facing a data deluge. One of the approaches proposed to allow the processing of such volumes is the programming paradigm MapReduce introduced by Google in 2004. This very simple implementation pattern is divided into two phases, map and reduce, between which a phase of massive exchange of data takes place among the machines running the application.
In this article, we propose the integration of the dispensing algorithm at intervals (Distributed Range Partitioning) in the MapReduce paradigm. The schema obtained is called MR^{2}P*. This new approach aims at dealing with dynamic scheduling of data and shuffle phase optimization (the intermediary phase between map and reduce).
The experiments show that our approach produces performance within a very interesting run-time execution and a better transition to scale (Scalability)

Keywords: MapReduce, Distributed file system, Big Data, Cloud Computing, Data distribution, Structures of distributed and scalable Data

1. INTRODUCTION

Currently, we are witnessing the era of mass production of data. On the one hand, applications generate data from logs, sensor networks, transaction reporting, GPS tracklog, etc. and on the other hand, individuals produce data such as photographs, videos, music or even dataset on the health status (heart rate, pressure etc.).

A problem then arises for data storage and analysis. The storage capacity of hard disk drives and the CPU speed are increasing but they remain insufficient to respond to the current applications. It is then necessary to parallelize data processing by storing on multiple hard disk drives. MapReduce is a Framework that responds to these issues.

The MapReduce paradigm is a mechanism for task partitioning in view of an execution distributed over a large number of servers. The principle is simple: it is to break up a task into smaller tasks, or more precisely to divide a task of larger-data volume into identical tasks of subsets of these data. The tasks (and their data) are then dispatched on different servers, and then the results are recovered and consolidated. The upstream phase of the breakdown of tasks, is the map part, while in the downstream phase, the consolidation of the results is the "Reduce" part [6].

MapReduce applies the principle which stipulates that "moving the calculation is less expensive than moving data" and is trying to schedule the map tasks in machines close to the input data that they process, in order to maximize data locality. The locality of the data is desirable because it reduces the amount of data transferred through the network, which reduces consumption of energy, as well as network traffic of data centers. However, in the scheduling of tasks reduce, the locality of the data is not at all taken into account. Some works [4],[16] have demonstrated that transfers through the network can install a considerable additional MapReduce work execution. Accordingly, several optimizations have been suggested in order to reduce the transfer of data among mappers and reducers. We have covered some of those proposals, from the intelligent planning of reduce tasks [4],[14] to the dynamic assignment of intermediary Keys, to reduce tasks at the time of planning [5**]**. However, all these approaches are restricted by the way the key-value pairs intermediaries

Email: m.aridj@uni-chlef.dz

are distributed over the outputs of the map phase. If the data associated with an intermediate key data are presented in the outputs of all map tasks, the pairs in the whole set of nodes, except one, must still be transferred through the network.

In this paper, we propose a technique, called MR^{2}P* which aims at optimizing the data transfer among mappers and reducers in the shuffle phase of MapReduce. In order to achieve this, we integrate a partitioning function based on the SDDS-RP * [1],[2].

Section 2 briefly recalls the principle of MapReduce. Section 3 presents the SDDS RP *. The principle and the algorithms of the new technique MR^{2}P* are developed in section 4. Section 5 is devoted to performance measures in the whole method by comparing it with basic MapReduce. Finally, we conclude this article in section6.

2. MAPREDUCE

MapReduce refers to both the programming model and the Framework originally developed by Google in 2004 for parallel data processing on a large scale. It is designed to automatically manage data partitioning, consistency and replication, as well as task distribution, planning, load balancing, fault tolerance, data distribution and load distribution.

The terms "Map " and " Reduce ", are borrowed from functional programming languages. The functions Map and reduce are defined on the key-value pairs. The Map function consumes input key-value pairs and produces a list of intermediate key-value pairs. And then, the framework brings together intermediate pairs by key and issues each group (the key and all the associated values) to the reduce function, which produces in its turn a list of key pairs- output value (Fig 1)

In a MapReduce job, the input is divided into M parts (splits), which are consumed by M map tasks, one for each part. The output of Map tasks is partitioned according to the intermediate key in R fragments by using a partitioning function, by default (hash= k mod R), which are then processed by R Reduce tasks.

When an operation is launched, the master partitions the entry into M fragments. The MapReduce tasks are then allocated to workers as soon as they become vacant, first the Map tasks, then the Reduce tasks, once all Map tasks are finished. The output of Map tasks is divided into R fragments according to the intermediate key and stored on the local workers hard drives. The Reduce tasks take these outputs and sort them by key so that all the values of a given intermediate key are dealt with jointly by the Reduce function. Once all the MapReduce tasks are finished, the user will be notified. The intermediate phase of a MapReduce job, when the intermediate keys are partitioned, sorted and transferred to the nodes that are running the Reduce tasks is known as the shuffle.

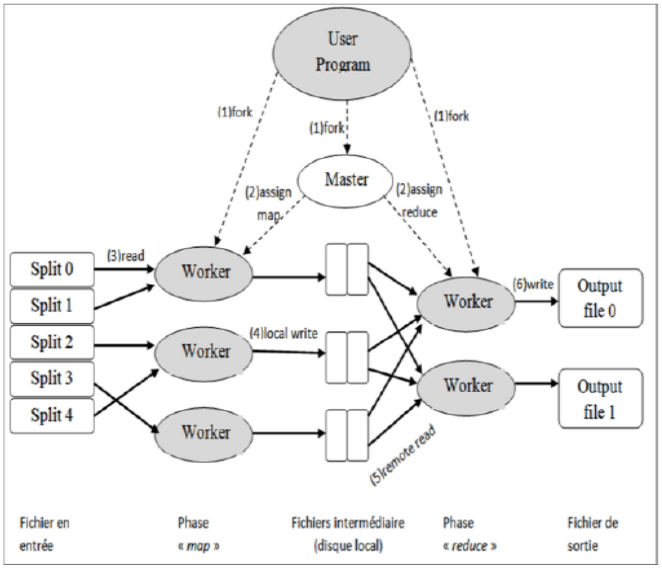

Figure 1. Principle of MapReduce

The MapReduce programming model consists of two map and reduce functions () defined on the key-value pairs:

The Map function (): consumes key-value pairs of input and produces a (possibly empty) list of intermediate key-value pairs.
Formally:
Map: (K1, V1) → stl (K2, V2)

```
Map (void * document){
  Int keys = 1;
  For each word m in document
Intermediate Calculation (m,keys); }
```

The Reduce function () : receives a list of intermediate pairs and then combines all the values corresponding to the same key to a unique pair (key, value).
Formally:

Reduce: (K2, list(V2)) → list(K3, V3)

```
Reduce(integer keys, Iterator values){
  Int result = 0;
  For each  v in values
    Result += v;}
```

3. THE SDDS RP*

The Scalable Distributed Data Structures (SDDS) are a class of data structure proposed in 1993 to support parallel processing of multiple computers and ensure access scalability. [13].

Historically, several SDDS have been suggested. The family of SDDS called LH*[13], develops distributed version of the linear hashing [9], [15], and the dynamic hashing [8]. The family called RP * (Range Partitioning) presents an extension to the clustered workstation technologies are expected data structures which reserve the order (Shaft-B Shaft or binary) [7],[10].Either types of SDDS have been proposed for the multi-key access and the high availability [11], [12].

An SDDS file RP * consists of records, each one identified by a primary key. The records on each server are stored in a memory space called check bucket. The keys of an RP file * are totally ordered. All the buckets correspond to a Partition Key Space. Each element of the partition corresponds to a bucket contains at maximum $b >> 1$ records having their key within an interval.

Clients access the file by sending search queries about a key or interval-queries. An interval-query refers to a set of records with their key in a given interval.

According to the addressing mode used at the client level, we distinguish three diagrams in RP * : RP * N, RP * C and RP * S. A client of RP file * N sends the queries to the servers using only the multicast messages. A file RP * C is an RP* file N with a partial index constructed from the IAM, at the level of each client. This index is an image generally partial of the structure of the distributed file. Each element of the index represents the interval and the address of a bucket already accessed by the client. The client uses the unicast messages when he addresses to a known check bucket of the image. The servers use the multicast messages to redirect queries in case of addressing error. Finally RP * S added to RP * C an index distributed at the level of servers indexing all the buckets. The queries and redirections are sent by unicast messages.

The interval $]\lambda, \Lambda]$ of a bucket is called *rank* .The parameter is λ the minimum key and Λ is called the maximum key of the bucket . A key record *c* belongs to the bucket of interval$]\lambda, \Lambda]$ such that $\lambda < c <= \Lambda$. For any *c*, there can be only a single check bucket in the RP* file to which *c* may belong. An RP* file is initially composed of a single bucket that bears the number zero (bucket *0*), with $\lambda = -\infty$ and $\Lambda = +\infty$. All inserts will initial in bucket 0. It erupted when it overflows. This results in the creation of a new bucket which bears the number a (*bucket i*). His rank is $]cm, +\infty]$, where *cm* is the key to the middle of the burst of the check bucket . The bucket that has exploded view starts with a new rank$]-\infty, c_m]$. This process is repeated for any bucket that overflows. The file can be extended to any number of sites of the multi-computer.

The internal organization of a bucket RP * is presented in detail in [8]. The storage space of a bucket is divided into three consecutive zones Fig 2:

1. The header. It includes the rank of the check bucket, the address of the root of the index associated with the check bucket, the current size of the bucket, the current number of records stored and finally the current size of the index.
2. The index. It is a variant of Shaft-B+ to 3 levels and without the sheets.
3. The data. This zone contains records and the sheets of the index.

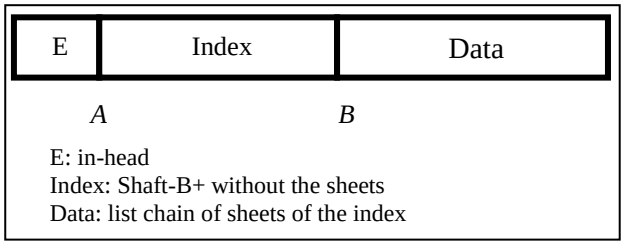

Figure 2. Structure of a bucket SDDS RP* .

The index is hierarchical and consists of nodes linked between them also by pointers, Fig 3. The root is located on the first level and used as an index to a second level. This second level itself is used to index a third level. This last level reference lists of records that correspond to the sheets. The sheets are linked between them using pointers. Each node may contain up to *n elements* of the index that are couples (*key, pointer*). Each pointer refers to a node on the next level. Each node must have at least n */2* elements except the root. This is the only node of the index, which may have at least one element.

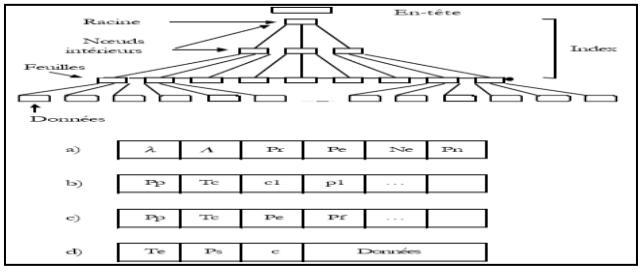

Figure 3. Detailed Structure of the bucket with a Shaft- B+ at three levels.

The SDDS file is handled through the following queries:

– **Simple queries**

A simple query is searching, inserting, modifying or deleting a given key record. Such a query is sent using a unicast or multicast message depending on the type of

client (*N or RP *C). A server *S* interval]λ, Λ] who receives a simple query of key *c* proceed as follows:

A) If $c \in]\lambda, \Lambda]$ then *S* runs the query and then sends a reply to the client with a IAM, if necessary. The reply is sent using a unicast message.

B) If $c \notin]\lambda, \Lambda]$ and if the query is sent using a unicast message then the issuer is a client RP *C. *S* considers that this is an addressing error. It inserts its interval and its IP address within the message and redirects it towards the other servers by using a multicast message. Finally, the good server processes the query and sends a reply to the client. In this case the response contains two IAM: that of S and that of the server which has treaty the query.

Consider a query to insert T size (header and data) at the level of a client. If $T < 64$ bytes, the query is sent in a single message following the UDP protocol. Otherwise, the client first sends the query without the data but by specifying their size in the message. The correct server, after receipt of the request, initializes a buffer large enough to receive the data.

He then opens a TCP port and then sends a unicast message to the client to ask him to connect to the data transfer. If the data have a very substantial size, the server initializes a buffer and then clarifies its size to the client in the request message of connection. It then transfers the data in blocks of a size equal to those of the server's buffer. The client stops transmission when the buffer is full. He then waits for the order from the server before starting again. The search works in the same way if the size of the data to be sent by the server is 64Kbytes.

– **Query at Interval**

It is to the search or the updating of fields non-key a set of records of key *c* belonging to an interval (a, *b*), with $a < b$. This interval is called *interval of the query*. A query by interval is sent using a multicast message. The affected servers then establish a TCP connection with the client for the transfer of records requests.

Either [*a, b*]*R* the interval of a search query *R*, sent by a client to a group of n servers, *S1,*

S2,... Sn. Either $]\lambda_i, \Lambda_i]$ the interval of server S *i* ,with i < n If *R* is received by a single server *Sk* (*1 < k < n*) such that (*a, b*)$R \subseteq]\lambda_i$,i , then ΛSk responds by sending the selected records and its own interval. This is the simplest case. The records research can however be distributed on several different servers.

In this case, *R* is treated on all servers *Si* such that $]\lambda_i, \Lambda_i] \cap [a, b]R \neq \{\}$. The query is executed in parallel on these servers and autonomously. Each door execution on a fragment of (*a, b*)*R*. The client has two strategies to complete a search query by interval:

A) Deterministic Termination: a server that receives a search query by interval responds by sending the requested data records and its interval, if the intersection of its interval and that of the query is non-empty. If no record is found, the server simply sends its interval to the client. The client completes the query if the union of intervals received covers that of the query. It also has a time out to complete the query if a missing response has not been received within a fixed deadline.

B) Probabilistic Termination: in this case a server responds only if it finds at least a record with a key belonging to the interval of the query. The client has a time out *t* to collect the answers. This time out is reset after each reply received. The query is complete if *t* expired. The practical choice of *t is to* have a time interval to make negligible the probability of loss of a response. This choice depends not only on the performance of the network but also on the processing speed of the machines

4. FRAMEWORK MR²P*

The MapReduce strategy is proving effective with regard to the initial data which are processed by the map function, however many data transfers are inevitable during the shuffle phase. This phase at which all the compute nodes will exchange data is therefore a major consumer of bandwidth and will therefore be limited by the capacity of the interconnection network. In addition, Reduce phase can only start after all data transmitted by the various maps have been received, which creates a barrier of strong synchronization. Optimizing the shuffle phase through an effective data structure can therefore have a significant impact on the overall performance of the application, especially when the congestion phenomena appear within the interconnection network.

Our contribution is to integrate into the shuffle phase of the MapReduce a more developed partitioning function based on SDDS-RP * presented in section 3.

The entry point of a MapReduce is an amount of data composed of several files. These files are going to be cuts in splits(subset). A split corresponds to a block(less than a split per file).

The idea is to substitute the output of each spot Map to a client RP * and each entry of a Reduce task to a server RP *. The shuffle operation in our proposal would therefore be a distributed file RP * (Fig 4).

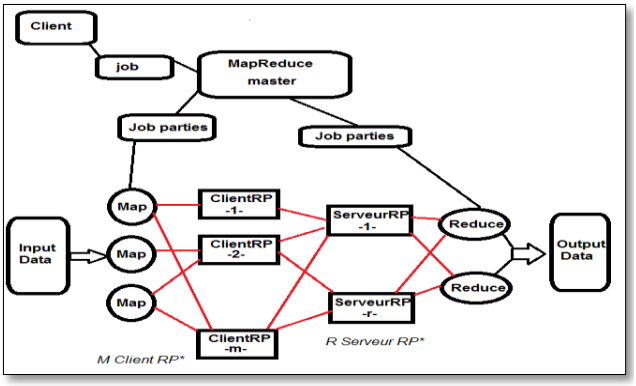

Figure 4. General architecture of the MR^{2}P*

The Server Architecture RP *

The Server RP * is the virtual entity that represents only one bucket of data. It combines the necessary modules for communicating the other entities of the system and the data.

In the cluster each Server RP * is identified by its IP address and the Port. A thread called Receiver Thread listens to all the time on this port, it is responsible for receiving the messages coming from other entities. It executes the following algorithm:

E1: wait for the arrival of message.
E2: Analyze and assign the message Processing order (Worker Threads).
E3: Return to step E1.

The following figure (Fig 5) presents the architecture adopted for the implementation of a RP* server.

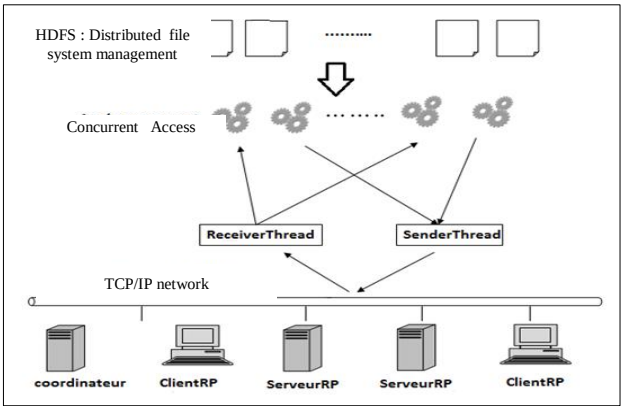

Figure 5. Architecture of a server RP *

The Architecture of Client RP *

After that the Map subtask ends, the Client RP takes the intermediate pairs key/value in output, placed into a command queue (Fig 6).

E1: wait for the query arrival.
E2: consult the shaft (index) and assign the query tothe matching WorkerThread
E3: pass on the request to the SenderThread for sending.
E4: Return to the step E1

For the queries processing of command queue, a group of threads WorkerThread is launched in correspondence with the servers containing the data. The WorkerThread run in competition according to the Client/Server model. Each of these Threads applies the following algorithm:

The proposed architecture for the implementation of a client RP * is presented in the following figure: (Fig.6)

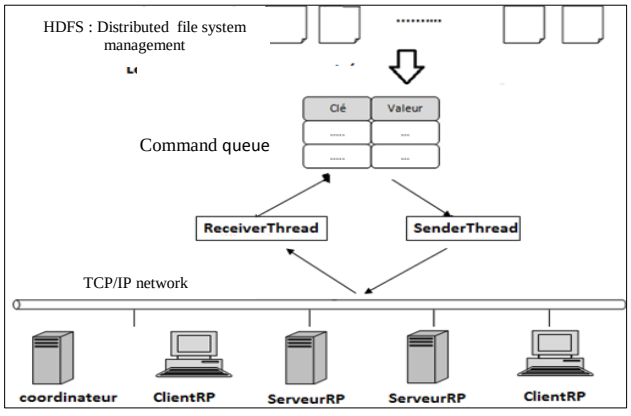

Figure 6. The architecture of the client RP *

Queries Execution in MR^{2}P*

In the MapReduce system, four types of entities (process) are involved with two layers, as shown in the following table: [3],[6]

Layer / Entity	HDFS	MapReduce
Master	NameNode	JobTracker
Slave	DataNode	TaskTracker

(HDFS :Hadoop Distributed File System).

- A NameNodelaunched on a machine playing the role of the master.
- Several DataNodes launched on multiple machines (1 unit = 1 DataNode) which will act as slaves. These processes, once functional, a logical topology of the site is defined as well as a distributed file system.

Under these hypotheses, the execution plan in the framework MR^{2}P* can be summarized as follows (Fig 7).

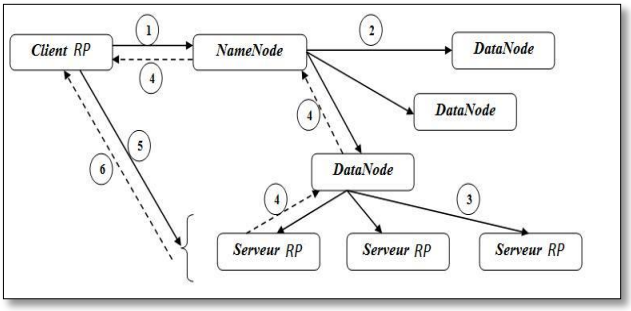

Figure 7. Execution plan for a query at intervals
In the framework MR^{2}P*

1. Load Data (In): The client sends a message of data loading of folder IN to the NameNode.
2. Load Data (In): in turn sends to the NameNode the order to the slaves (DataNodes) for data preparation and loading into RAM.
3. Start the Server Threads RP that each represents a data bucket .
4. Reply All Done: each entity notified by the statement in the request that it has been assigned to him until the client PII.
5. The client can query the whole of the server containing the data.
6. The set of servers responds to the client's query.

5. Simulation and Performance Test

We implemented the proposed scheme in Java under Linux Ubuntu 12.04 on a suitable of 4 posts (Intel Core Duo 2.13 GHz, 2 GB of RAM.). Each machine can contain multiple servers and multiple clients (concept of virtual machine).

A) Performance of distribution

Tests were done on a varied size of intermediate data (10000, 30000, 50000, 100000) by changing the capacity of servers. The following figure (Fig 8) represents the overall time required for the distribution of intermediate key by varying the capacity of servers. Always time in milliseconds (ms).

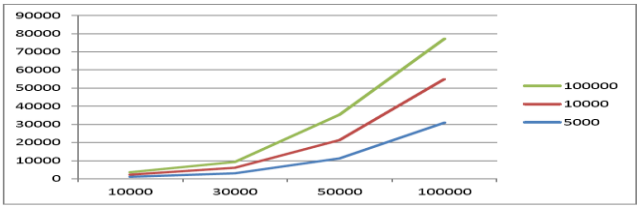

Figure 8. Distribution Time (ms) depending upon the server capacity (key).

By observing the curves in Figure (fig 9), it can be seen that the time required for the Peer Distribution (key, values) over the Reduce tasks increases linearly with the number of data, which will give a constant average time by processed data.

B) Shuffle operation Time

The curve in the figure (Fig 9) presents the average time required for the data partitioning in the interim phase of the MapReduce (shuffle) operations

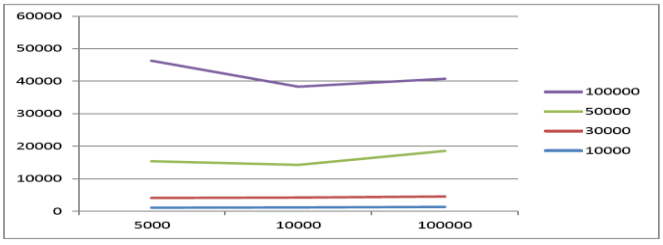

Figure 9. Average time of shuffle operation

The curves are almost linear.This implies that we have an average time independent of the number of keys, or the scalability of the method MR^{2}P*.

C) Time for search operation reconsideration:

In order to validate the adopted approach, we implemented the classic example of WordCount;

The figure (Fig 10) shows the variation of the average time required to calculate the occurrence of a word given (key) in a text. We observe that time is practically stable regardless of the number of keys present in the system, which guarantees on the other hand a better transition to scale(Scalability).

The fig (fig 11) gives the variation of the required average for the calculation of the occurrences number of an interval of words in a given text. (Ex. a*: the words that started with the character a).

Noting, that this operation cannot be carried out effectively in the basic MapReduce, because the data were not ordered. Therefore, to find the occurrences of

the words starting with the character a, we have to interrogate all reduce tasks.

However in MR^{2}P* only reduces likely to contain the data will be interrogated.

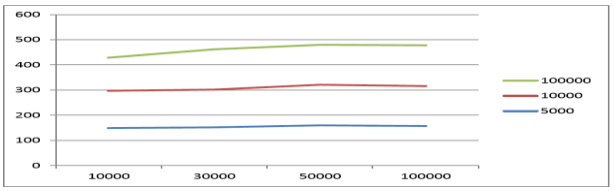

Figure 10. Time of simple search of a key (ms) depending upon server capacity

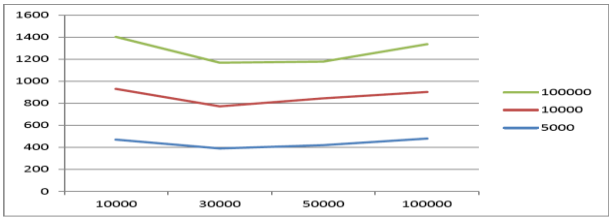

Figure 11. Search time at interval (ms) depending upon server capacity

6. CONCLUSION

The amounts of data that are captured or generated by modern computer devices have increased exponentially in recent years. This phenomenon is called "Big Data"

Big Data is the expression used to refer to data that due to their size and complexity, are difficult to manipulate with traditional data processing and management tools. To make up for this lack, the framework MapReduce was introduced.

In this article, we are committed to optimize the shuffle phase of MapReduce application. For this purpose, we replace the default partitioning function (hash(k) =k mod R) used in the basic MapReduce by a more sophisticated function (SDDS-RP *). The resulting Schema is baptized MR^{2}P*.

Compared to the basic approach of MapReduce, we showed in MR^{2}P* a possibility of execution at intervals, of better execution time, more stable and without failure, as well as a better transition to scale.

In the future, we are considering implementing those algorithms in production and compare them to existing solutions, such as Hadoop [17].

ACKNOWLEDGMENT

We would like to express our heartfelt thanks to Professor Wintold Litwin's of the Paris Dauphine University for the time devoted to the discussions of the ideas presented in this article.

REFERENCES

[1] Devine, R. "Design and Implementation of DDH: Distributed Dynamic Hashing ". *Intl. Conf. On Foundations of Data Organizations, FODO-93. Lecture Notes in Comp. Sc., Springer-Verlag (publ.), Oct. 1993*.

[2] Diene Aly W. *"Internal organization of an SDDS bucket RP * "*, memory of DEA, Dept. Mathematics and Informatics, Cheikh Anta Diop University of Dakar,february 1998.

[3] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung: "The Google File System" *Paper SOSP'2003.*

[4] Hammoud M. , M. S. Rehman, and M. F. Sakr, "Center-of-gravity reduce task scheduling to lower MapReduce network traffic *", in Proceedings of the 2012 IEEE Fifth International Conference on Cloud Computing , 2012, pp. 49-58.*

[5] Ibrahim S. , H. Jin, L. Lu, S. Wu, B. Hey, and L. Qi, "LEEN: locality/fairness-aware key partitioning for MapReduce in the cloud ", in *Proceedings of the 2010 IEEE Second International Conference on Cloud Computing Technology and Science, 2010, pp. 17-24*

[6]Jeffrey Dean and Sanjay Ghemawat, "MapReduce: Simplifed Data Processing on Large Clusters". *OSDI 2004 Google.*

[7] Karlsson, J. Litwin, W. ,Risch, T. "LH * lh: A Scalable High Performance Data Structure for Switched Multicomputers ". *Intl. Conf. on Extending Database Technology (EDBT) study program-96, Vignon, March 1996.*

[8] Larson, P. " Dynamic Hash Tables", *CACM, 31/4/1988.*

[9]Litwin, W. "Linear Hashing : a new tool for file and tables addressing ". *: Reprinted from VLDB-80 in READINGS IN DATABASES. 2-Nd ed. Morgan Kaufmann Publishers, Inc. , 1994. Stonebraker , M. (Ed.).*

[10][Litwin and al,1994] Litwin, W. ,Neimat, M-A. , Schneider, D. " RP * : A Family of Order-Preserving Scalable Distributed Data Structures". *20TH Intl. Conf on Very Large Data Bases (VLDB), 1994.*

[11] Litwin, W. ,Neimat. "K-rp* : a Family of High Performance Multi-attribute Scalable Distributed Data Structure". *In IEEE Intl. Conf. It s. & Distr. Systems, PDIS-96, (Dec. 1996).*

[12] Litwin.W. , M-A Neimat. "LH * s: a highavailability and high - security Scalable Distributed Data Structure". *IEEE Workshop on Research Issues in Data Engineering. IEEE Press, 1997*

[13] Neimat, M-A. , Schneider, D. LH *: Linear Hashing for Distributed Files". *ACM-SIGMOD Intl. Conf. On Management of Data, 1993*.

[14] Seo S. , I. Jang, K. Woo, I. Kim, J. -S. Kim, and S. Maeng, "HPMR:prefetching and pre-handle "shuffling in shared MapReduce computation environment", in *Proceedings of the 2009 EEE International Conference on ClusterComputing, 2009, pp1-8.*

[15] Severance, C. ,Pramanik was said, S. Wolberg, P. " Distributed linear hashing and parallel projection in main memory databases". *Conf on Very Large Data Bases (VLDB),1990*

[16] Wang.G. ,Butt.A. R, Pandey.P, and Gupta.K, "A simulation approach to evaluating design decisions in MapReduce setups", in *17th Annual Meeting of the IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems , 2009,pp. 1-11.*

[17] White, Tom: " Hadoop: the definitive guide ". *O'Reilly Media , Inc. 2009. ISBN: 978-0 -596-52197-4.*

Aridj Mohamed is assistant professor at Hassiba Benboali university chlef Algeria since 2001.
His area of interest includes Software Engineering, distributed System, Databases, Big data and cloud computing