International Journal of Computer Applications (0975 – 8887)
Volume 121 – No.10, July 2015

Exploratory Implementation of Stream Clustering Algorithm using MongoDB

Jyotsna Talreja Wassan
Asstt. Professor
Maitreyi College
University of Delhi, India

ABSTRACT
In the recent years, Big Data has become ubiquitous and various big data tools are greatly in use to accelerate the computing and analytics in various fields. Various algorithms in *Computer Science* use large and heterogeneous data sets; and hence could be explored with Big Data platforms. One such class of algorithms is stream clustering algorithms; dealing with large scale processing of incremental data. This motivation of using Big Data tools may lead to improved efficacy of running the algorithms. Hadoop, the most popular open source implementation of MapReduce, has been utilized and modified for catering the needs of numerous clustering problems. But various scientific and computing fields are also using MongoDB, a document oriented NoSQL store supporting Map Reduce. The main purpose of this paper is to try and judge the usage of MongoDB as a Big Data platform for implementing a stream clustering algorithm using MapReduce programming model to study the factors relating Map Reduce and MongoDB together.

General Terms
Data Mining, MongoDB, NoSQL, ExCC

Keywords
Big Data, MapReduce, Sharding, Clustering, Grid

1. INTRODUCTION
The capability to process large volumes of data streams is one of the key areas of interest in today's computing world. In the present scenarios, dynamic web data analysis, e-commerce, internet news and data monitoring systems, real-time analysis of scientific or geo -spatial data etc. require large volumes of multi-dimensional data at higher data rates [17, 18, 19]. The processing of this kind of data needs to be highly productive and efficient to achieve real time response [7, 8]. Thus, we need a paradigm that can do scalable data mining technique on multidimensional data as per the user real time requirements. Clustering as a data mining technique on streaming data, focuses on partitioning a list of data points into n groups of "similar" data objects after scanning the given data [15, 16]. The input data to stream clustering algorithm is usually complex, with hundreds of mixed attributes, and is continually evolving [6, 11]. It is being preferred and is useful to discover patterns in data and validate its correctness. In this paper, the implementation of ExCC, a stream clustering algorithm with, scalable document-based NoSQL data store (MongoDB) and its MapReduce framework is being presented.

2. ExCC: A STREAM CLUSTERING ALGORITHM
ExCC stands for Exclusive and Complete Clustering. The algorithm particularly deals with data streams which are continuous and unbounded in nature [4, 5, 6]. The nature of data streams makes storage of data and multiple accesses of data a difficult task for pattern discovery. The algorithm uses grid based synopsis structure, to consolidate incoming data points from streams. This synopsis once build, is used in clustering scheme. The Exclusive and Complete Clustering (ExCC) algorithm tries to generate non-overlapping clusters in data streams with mixed attributes. It also ensures that each data point has its membership in some cluster or is an outlier/noise otherwise. Hence, it is termed as ExCC [1, 4, 5, 6].

2.1 The Clustering Approach
Input: List of data points to be mapped to grid structure. [4, 5, 16]
List of cells is in the cell pool "cell_list" made from grid structure.

```
Initialize cluster pool CLUSTERS, empty in the beginning
 While (cell_list not empty) do
{Select next cell c from cell_list
    For each cluster C in the Cluster pool CLUSTERS
    {
        For each cell cs in C do
         {   if adjacent(c, cs ) then
               {
                   Add cell c to C and update cluster statistics
            }
            Break;
         }
        If same cell is added in any of the Previous Cluster
           {Merge two Clusters}
    }
   If cell c has not been added to cluster pool CLUSTERS,
   {   Create new cluster C1 with c as a member
       Add C1 to the cluster pool CLUSTERS
    }
}
```

densitythreshold = nue/(log(avggran+dimensionality));
var **sigfthreshold** =(densitythreshold*((log(avggran)) +(log(dimensionality))))/(2*(log(avggran))*(log(dimensionality)));

//No. of points in a cluster Is calculated
If **points_clstr/cells_clstr** is less than **sigfthreshold**
{Cluster is removed}
Output: Clusters of data points based on similarity

3. MONGODB AS A BIG DATA PLATFORM

MongoDB (from "hu**mongo**us") is a NoSQL, open source document store developed by 10Gen Company. MongoDB stores structured data as JSON heterogeneous documents with dynamic schemas rather than traditionally storing data in relational database store. The concept of dynamic schema in MongoDB platform makes the integration of data easier and faster in various applications. Databases in MongoDB are the groups of collections stored on disk where collections are containers for documents/records that share one or more indexes. Each document collection is stored in one namespace file.*mongod* is the primary server process for the MongoDB system. Its purpose is to handle data requests, manages data format, and performs background management operations.

mongo is an interactive JavaScript shell which delivers and interface to fire queries and operations directly on/with the database[20, 21 , 22, 23]. MongoDB supports replication concept that ensures backup, load balancing and automatic failover over mongod instances. MongoDB scales horizontally through sharding. Sharding uses range-based partitioning to distribute *documents* based on a specific index known as *shard key*. Sharding automatically balances data and load across machines. A *sharded cluster* has one or more *config servers* which store the metadata for the cluster, two or more shards consisting of one or more **mongod** instances that store the actual data for the shard and one or more **mongos** instances that direct queries from the application layer to the shards that hold the data. The **mongos** instances are the route directors and have no persistent state or data files and only cache metadata in RAM from the config servers [9].

4. MAP REDUCE PARADIGM

Map Reduce is a powerful programming paradigm that is useful for scalable applications [10].The MapReduce programming model consists of Map and Reduce functions as listed below:

Map Function: Takes an input pair and produces a set of intermediate key/value pair's e.g.

Map: (key_1, $value_1$) → list (key_2, $value_2$)

Reduce Function: This function accepts an intermediate key and a set of values for that key.

Reduce: (key_2, list (key_2, $value_2$)) → $value_3$

The MapReduce paradigm try to group all intermediate values associated with the same key. The basic idea behind MapReduce as illustrated in Figure 1; is to divide the problem into a set of smaller sub-problems that tends to perform the same operations on a subset of the data in parallel in Map phase and subsequently results from multiple Map functions generate intermediate results which are given to Reduce Phase [10].

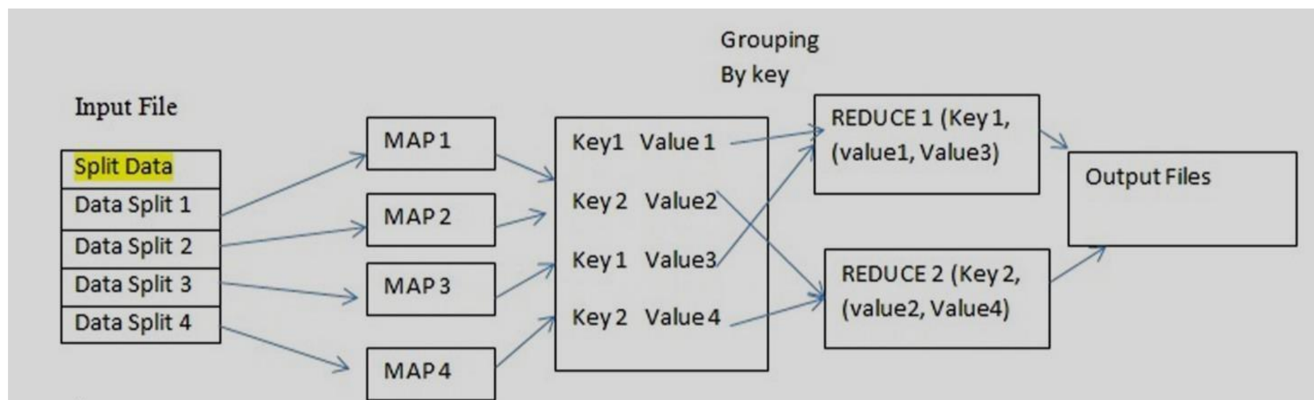


Fig1. Map Reduce Basic

5. MONGODB AND MAPREDUCE

MongoDB added MapReduce in 1.2 Version and onwards. The MapReduce basically consists of two steps – get the data (map), then perform some aggregating function -reduce, with an optional one time end operation (finalize). MapReduce is applied on an input collection. The mapping step transforms the inputted documents and emits a (key,value) pair. Then, key/value pairs are grouped by key, such that values for the same key end up in a list. The reduce gets a key and this list of values emitted for that key, and produces the final result. The final result of the operation is stored in entirely a new collection which can be used for further querying or processing [9, 20, 21, 22, 23].

The **map** function in MongoDB has the following prototype:

```
map= function (){
…
emit (key, value); }
```

The **map** function exhibits the following behaviors in MongoDB as mentioned in :

- In the **map** function, the current document is referred with **this** within the function.
- The **emit (key, value)** function associates each **key** with a **value**.
  - A single emit can only hold half of MongoDB's *maximum document size.*
  - The **map** function can call **emit (key, value)** any number of times, including 0, per each input document.

The **reduce** function in MongoDB has the following prototype:

```
function(key,values){
…
return result;};
```

The **reduce** function demonstrates the following behaviors:

- MongoDB will **not** call the **reduce** function for a key that has only a single value.
- MongoDB can invoke the **reduce** function more than once for the same key. In this case, the previous output from the **reduce** function for that key will become one of the input values to the next **reduce** function invocation for that key.
- The **reduce** function can access the variables defined in the **scope** parameter.

The **finalize** function has the following prototype:

```
function (key, reducedValue){
 …
return modified object;};
```

The **finalize** function receives as its arguments a **key** value and the **reducedValue** from the **reduce** function. It has the following properties [20]:

- The **finalize** function is used to implement the desired function for which reduce was not called in case of single key value.
- The **finalize** function can access the variables defined in the **scope** parameter.

The **db.collection.mapReduce** () method provides a wrapper around the **MapReduce** command in MongoDB and is used to run mappers and reducers over a collection.

The MapReduce framework can be applied on sharded database. The main aim is to create a sharded database and then run map-reduce on it to process the data .When using sharded collection as the input for a map-reduce operation, mongos will automatically dispatch the map-reduce job to each shard in parallel and must wait for jobs on all shards to finish. By default the output collection is not sharded. The process to perform sharding involves mongos as a router which dispatches a map-reduce finish job to the shard that will store the target collection. The target shard pulls results from all other shards, and runs a final reduce/finalize operation, and write to the output. MongoDB shards the output using _id field as the shard key by default. The functional flow is illustrated in Figure 2.

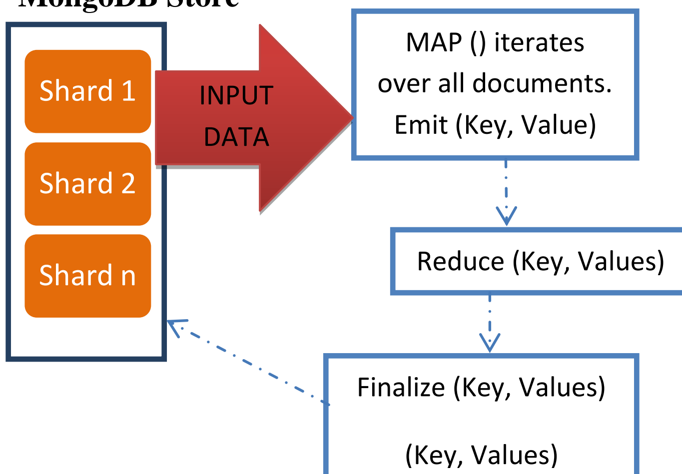


Fig2. Functional flow in MongoDB

6. MONGODB BASED IMPLEMENTATION OF EXCC

The inputs and outputs of the ExCC algorithm are maintained in MongoDB data collections.The algorithmic flow consists of four MapReduce Operations/Jobs each having a unique functionality [4,5].The stepwise job processing for EXCC is discussed in section 6.1 (Figure 3, Figure 4) [2,3,4,5]. The mappers and reducers are listed in section 6.2 .

6.1 Stepwise Implementation of ExCC

The proposed implementation is divided into four jobs as listed in following sub sections.

6.1.1 JOB 1: Cell Signature and Statistics Generation Job

The input to job 1 constitutes data in the form of "Input Data" collection (which stores input data) and "Config" collection (which stores configuration parameters). Each line in the input data collection begins with timestamp of the data followed by the value of the data in each dimension. The configuration collection describes each dimension of the data points with the indicators of **"**Type of each dimension (whether numeric or categorical)", "the granularity of the dimension", "the minimum value and the maximum value in the range of values" for the dimension. In Map Phase, the cell signatures for each data point of the current batch are generated**.** In Reduce phase, the synopsis for the current batch is obtained i.e. for each cell signature the cell statistics is generated constituting first timestamp at which data point came, Last timestamp at which data point came and the total number of points in cell. The output of running MapReduce on input collections is stored in a new output collection, named "MROutput01" which will be used in further processing (Figure 3a).

6.1.2. JOB 2: Synopsis Maintenance and Recent Cell Detection Job

The input to job 2 is data generated in previous job stored in "MROutput01" collection merged with previous synopsis "i.e. Synopsis Collection" of the last iteration for one run of the EXCC algorithm. The Map phase again considers the cell signatures and cell statistics and emits them to the reducer. The reducer updates the statistics for cell signatures which were present in the previous synopsis file, and forwards it to a finalize function as MongoDB will **not** call the **reduce** function for a key that has only single values. The pruning of non- recent cells takes place in finalize () method . The pruning phenomenon considers average interval time for the cells and the recency factor. The average interval time (aat) for a cell is then calculated as follows:

aat = (Last Timestamp - First Timestamp + 1) / (Number of points in cell)

The recency factor is calculated as follows:

Recency = (currentTime – Last Timestamp) / Number of points that have arrived in the stream since the last clusteringA cell is considered recent and generated as output only if the aat > recency. The MapReduce process creates a new output collection "MROutput02" which will be used as input to job3.The copy of MROutput02 is used to generate a new synopsis file for next iteration (Figure 3b).

6.1.3. JOB 3: Dense Recent Cell Detection and Clustering Phase 1 Job

Input to Job3 is the collection having pruned cells that have been generated as output in previous job. The purpose of

MapReduce is to generate dense cells from the pool of recent cells. The density function is calculated as follows:

nue = total_points/total_prunedcells;
denFn= Math.ceil(nue / Math.log(g_avg+dimensions)));

A cell is considered dense if the number of data points it contains is more than the density function. All dense cells are generated as output. The map function checks the condition for density and emits dense cells. All cells are emitted with same key from Map, to ensure there is only one reducer. The Reduce function partition the dense cells, on the basis of adjacency of the first dimension. All cells with first dimension, adjacent to one another are assigned the same cluster label. The value list is scanned sequentially, if the absolute difference in the first dimension is more than 1, the cluster label is incremented (Figure 3c).

6.1.4. .JOB 4: Clustering Phase 2 (Actual Clustering) and Finding Significant Clusters Job

The input to job4 is the list of clusters created in job 3.The Map function considers the list and emits it to reducer which forwards it to a Finalize () function. Finalize () function performs the actual EXCC clustering as described in Table1. From the clusters obtained; the statistics for the significant clusters is generate (Figure 3d).

6.2. Algorithmic Flow of EXCC in MongoDB

The implemented algorithm is described as follows:

1. The Input Database collections are sharded and thus are distributed over the nodes of a cluster of machines**.**
2. Then a sequence of MapReduce Operations are run over the sharded collection:

i) **Map Function:**
Input: (Input Data Collection,
Config Collection)
Output: (Key - Cell Signature for each point
Value — Cell Signatures, Time Stamps & Density of each point as 1])
Reduce Function:
Input: (Key - Cell Signature
Value – Cell Signatures with respective Time Stamps & Density of each individual point as 1)
Output: (Key - Cell Signature
Value – Cell Signatures with Statistics
[First timestamp | Last timestamp | Number of points in cell])
Output Collection: MROutput01

Merge Previous Synopsis Collection with MROutput01

ii) **Map Function:**
Input: (MROutput01 Collection)
Output: (Key – Cell Signature
Value – Cell Signature with Cell Statistics and recency flag)
Reduce Function:
Input: (Key – Cell Signature
Value – List of Cell Statistics with recency flag)
Output: (Key – Cell Signature, Value – Cell Statistics with recency flag) It updates the cell statistics.
Finalize Function:
Input (Key – Cell Signature, Value – Cell Statistics with recency flag)
It performs the function of checking the recency condition on a cell.
Output Collection: MROutput02.
New Synopsis File is generated for Next Iteration

iii) **Map Function:**
Input (MROutput02)
Output: (Key – 1, Value – Cell Signatures)
Reduce Function:
Input: (Key – 1, Value – Cells Signatures)
Output: (Key – Cluster id, Value –Clusters on the basis of adjacency on I^{ST} Dimension of Cell Signatures).
Output Collection: MROutput03.

iv) **Map Function:**
Input: (MROutput03).
Output: (Key – Cluster id,
Value – Clustered List of Cells)
Reduce Function:
Input: (Key – Cluster id
Value – Clustered List of Cells)
Output: (Key – Cluster id,
Value – Clustered List of Cells)
Finalize Function:
Input: (Key – Cluster id,
Value – Clustered List of Cells)
It performs actual Clustering and checks the condition for significant clusters.
It generates final Clusters.
Output Collection: MROutput04, containing final Clusters

3. Repeat Step 2 for all batches of a stream to be clustered.

7. A SAMPLE RUN OF ExCC IMPLEMENTED AT MONGODB PLATFORM

The algorithm was run jobwise (as indicated in section 6.1), on the simulated data at MongoDB platform installed at Acer machine with Ubuntu Linux as operating system and an AMD Phenom(tm)11 X 4 810 processor, 2GB of RAM, and a 320GB hard disk. The small experimented sample input file, configuration file and sample parameters are listed in Table1.

Table1. Input Sample for running MongoDB based Implementation of ExCC

Sample Input	Configuration	Extra features for Algorithm Simulation
0|46,56,R,1 1|164,45,Y,1 2|46,58,R,1 3|23,67,R,1 4|125,45,Y,1 5|125,78,Y,1 6|176,78,B,1 7|400,37,Y,2 8|247,56,B,2 9|156,56,Y,1 10|200,30,R,2 11|200,30,Y,2	0|N|10|0|500| 1|N|10|0|100| 2|C|4|B|G|R|Y| 3|C|2|1|2|	@Dimensions 4 @Numeric Dimensions 2 @Epsilon 0.5 @Average Granularity 3 @Current Time | Batch Size 12 | 12

Job1 deals with forming CELL SIGNATURES AND STATS and took 157ms. The results are stored in MROutput01 collection.

> db. MROutput01.find ();

{"_id" :{ "sig": [0, 5, 2, 0]},
"value" : {
"signature" : [0,5, 2, 0],
"stats": [0, 2, 2]}}
{"_id" :{"sig": [0, 6, 2, 0]}, "value": {
"signature" : [0, 6, 2, 0],
"stats": [3, 3, 1]}}
{"_id":{"sig": [2, 4, 3, 0]}, "value": {
"signature" : [2, 4, 3, 0],
"stats": [4, 4, 1]}}
{"_id":{"sig": [2, 7, 3, 0]}, "value" : {
"signature" : [2, 7, 3, 0],
"stats": [5, 5, 1]}}
{"_id":{ "sig":[3, 4, 3, 0]}, "value" : {
"signature" : [3, 4, 3, 0],
"stats": [1, 1, 1]}}
{"_id":{ "sig":[3, 5, 3, 0] }, "value" : {
"signature" : [3, 5, 3, 0],
"stats": [9, 9, 1]}}
{"_id" :{"sig": [3,7, 0,0]}, "value": {
"signature" : [3, 7, 0, 0],
"stats": [6, 6, 1]}}
{"_id":{ "sig": [4, 3, 2, 1]}, "value": {
"signature" : [4, 3, 2, 1],
"stats": [10, 10, 1]}}
{"_id" :{ "sig": [4, 3, 3, 1]}, "value" : {
"signature" : [4, 3, 3, 1],
"stats": [11, 11, 1]}}
{"_id" :{"sig":[4, 5, 0, 1] }, "value" : {
"signature" : [4, 5, 0, 1],
"stats": [8, 8, 1] } }
{"_id":{"sig": [8, 3, 3, 1]}, "value": {
"signature" : [8, 3, 3, 1],
"stats": [7, 7, 1]}}

Job2 deals with pruning for getting recent cells. Cell signature and cell statistics are retained. Since this is first iteration, all cells are retained. The job on RECENT CELL PRUNING took 16ms and stored results in MROutput02.

> db.MROutput02.find ();

{"_id": {"signature":
[0, 5, 2, 0]},
"Value":
{"signature": [0, 5, 2, 0],
"stats": [0, 2, 2],
"frecency": 0}}
{"_id": {"signature":
[0, 6, 2, 0]},
"Value":
{"signature": [0, 6, 2, 0], "stats": [3, 3, 1],
"frecency": 0}}
{"_id": {
"signature": [2, 4, 3, 0]},
"Value": {
"signature": [2, 4, 3, 0], "stats": [4, 4, 1],
"frecency": 0}}
{"_id": {"signature":
[2, 7, 3, 0]},
"Value": {
"signature": [2, 7, 3, 0], "stats": [5, 5, 1],
"frecency": 0}}
{"_id": {"signature":
[3, 4, 3, 0]},
"Value": {
"signature": [3, 4, 3, 0], "stats": [1, 1, 1],
"frecency": 0}}
{"_id": {"signature":
[3, 5, 3, 0]},
"Value": {
"signature": [3, 5, 3, 0], "stats": [9, 9, 1],
"frecency": 0} }
{"_id": {"signature":
[3, 7, 0, 0]},
"Value": {
"signature": [3, 7, 0, 0], "stats": [6, 6, 1],
"frecency": 0}}
{"_id": {"signature":
[4, 3, 2, 1]},
"Value": {
"signature": [4, 3, 2, 1], "stats": [10, 10, 1],
"frecency": 0}}
{"_id": {"signature":
[4, 3, 3, 1]},
"Value": {
"signature": [4, 3, 3, 1], "stats": [11, 11, 1],
"frecency": 0}}
{"_id": {"signature":
[4, 5, 0, 1]}, "value":
{"signature": [4, 5, 0,], "stats": [8, 8, 1],
"frecency": 0}}
{"_id": {"signature":
[8, 3, 3, 1]},
"Value": {
"signature": [8, 3, 3, 1], "stats": [7, 7, 1],
"frecency": 0}}

Here frecency is a flag which is set to 0 if cell is recent MROutput02 is collection having same data as MROutput01.In multiple iterations non recent cells will be pruned and hence MRoutput02 will not necessarily be same as MROutput01.
Job3 deals with finding of dense cells and partitioning cluster lists.
The job on DENSE CELLS and sorting on dimension and partitioning cluster lists took 21ms.

db.MROutput03.find();

{ "_id" : 1, "value" : { "cl" : [
[
1, [0, 5, 2, 0, 2],
[0, 6, 2, 0, 1]
],
[
2, [2, 4, 3, 0, 1],
[2, 7, 3, 0, 1],
[3, 4, 3, 0, 1],
[3, 5, 3, 0, 1],
[3, 7, 0, 0, 1],
[4, 3, 2, 1, 1],
[4, 3, 3, 1, 1],
[4, 5, 0, 1, 1]
],
[
3,

```
[ 8, 3, 3, 1, 1 ]
]
]} }
```

Job 4 on final clustering took 15 ms. The final results are stored in collection MROutput04.
> db.MROutput04.find();

```
{"_id": 1,
"value": {"CLUSTERS" :
[[0, [0, 5, 2, 0],
    [0, 6, 2, 0]]]
} }
```

* This Cluster 1 contains 2 cells

```
{"_id": 2,
"value" : { "CLUSTERS" :
[[0, [2, 4, 3, 0],
    [3, 4, 3, 0],
    [3, 5, 3, 0],
    [4, 3, 3, 1],
    [4, 3, 2, 1]
],

[1, [2, 7, 3, 0],
   [3, 7, 0, 0]
],
[2, [4, 5, 0, 1]
] ]
}
}
```

* This cluster contains 8 cells (5+2+1) with three sub clusters

```
{"_id": 3,
"value" : { "CLUSTERS" :
[[0, [8, 3, 3, 1]
]
]
}}
```

* This cluster Contains 1 cell.

The total time taken to run the algorithm on this small simulated data is 209ms and it takes even more time when run on big Kdd- 99 intrusion data sets in forming clusters [14]. Various limitations were faced while trial of modelling stream algorithm at MongoDB platform. These limitations are listed in section 8.Thus it is being proposed that the algorithm might be tried and improved by exploring other parallel analytical platforms like Hadoop etc. for running stream algorithm as future perspective.

8. LIMITATIONS OF USING MONGODB

Our main focus in this section is to outline the essential limiting features of MapReduce framework, faced by using MongoDB, and how they must be addressed to ensure a successful run of the framework. The MongoDB has the following features [9]:

a. MongoDB **does not** call the **reduce** function for a **key** that has only a **single value**. This optimizes the performance of MapReduce. But this feature applies to only limited sets of use cases. For instance, it would be a problem, if a mapper and reducer is designed to sum values per key. If there is a single key, one wouldn't get an aggregated count. Thus if one needs to perform an operation on a key once, Finalize () function must be used. The tradeoff is that it requires a programming effort in writing Finalize () functionality.

b. The key value emitted from a map function and return values from a reduce function, **cannot be an array** (it's typically an object or a number). Thus if an array is to be emitted, it is needed to be embedded in an object in MongoDB.

c. The MapReduce engine may invoke many **reduce functions iteratively**; thus reducer does aggregate on the results of mapper function by small groups producing **intermediate results** .Then reduce functions are run again on these intermediate results as they were direct results of the map function. And the process goes on until there is only one intermediate result left for each key. It can be seen as a hierarchal pyramid of intermediate results. Because it is possible to invoke the reduce function more than once for the same key, the following must hold for the reduce function:

    i. The *type* of the return object in reduce must be **identical** to the type of the value emitted by the map function
    ii. The reduce function must be *idempotent* (that can be applied multiple times without changing the result beyond the initial application.) i.e. for all key, values: reduce (key, [reduce (key, values)]) = = reduce (key, values).

d. MongoDB can scale horizontally via sharding. But MapReduce aggregations are single-threaded. This is due to the design limitation of JavaScript Engines which has a limitation of 1 thread per Shard and has a slower performance.

e. A document in MongoDB is limited to a size of 16 MB i.e. this can be the maximum length of a string in the records in MongoDB.

f. MongoDB, 32 bit processes are limited to about 2 GB of data.

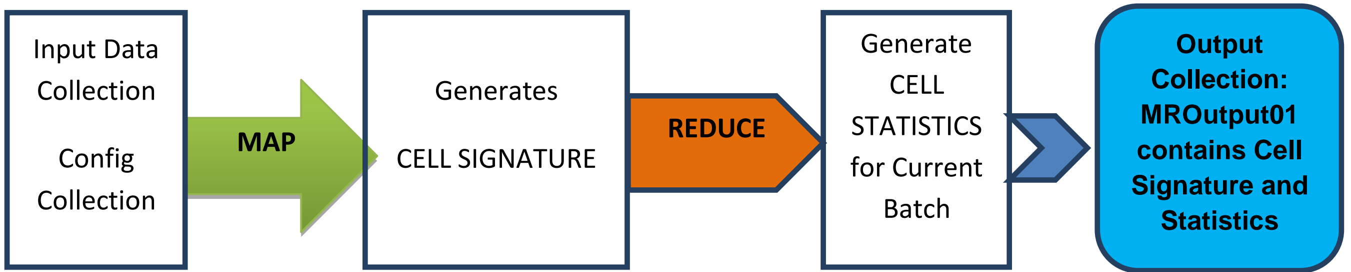
9. CONCLUSION

MongoDB is a NoSQL store empowered with its own built-in MapReduce implementation. The MapReduce scales with the number of shard servers (sharding). MongoDB's MapReduce can be executed in parallel at each shard, but it is has its own limitations and is not very much suitable for data analytics for continuous data. The two major concerns are:(1) MapReduce functional scripts are written in JavaScript, which is slow and does not have good support for analytics (2) Java script implementation used by MongoDB, is single threaded; hence only one MapReduce program can run at the same time. The platforms like Hadoop, PACT etc. may prove more useful in continuous data analytics and processing; as these platforms are not single threaded [21]. In future MongoDB – Hadoop Connector could be explored to do storage in NoSQL MongoDB store and analytics in Hadoop. This may enhance performance compared to native MapReduce of MongoDB.

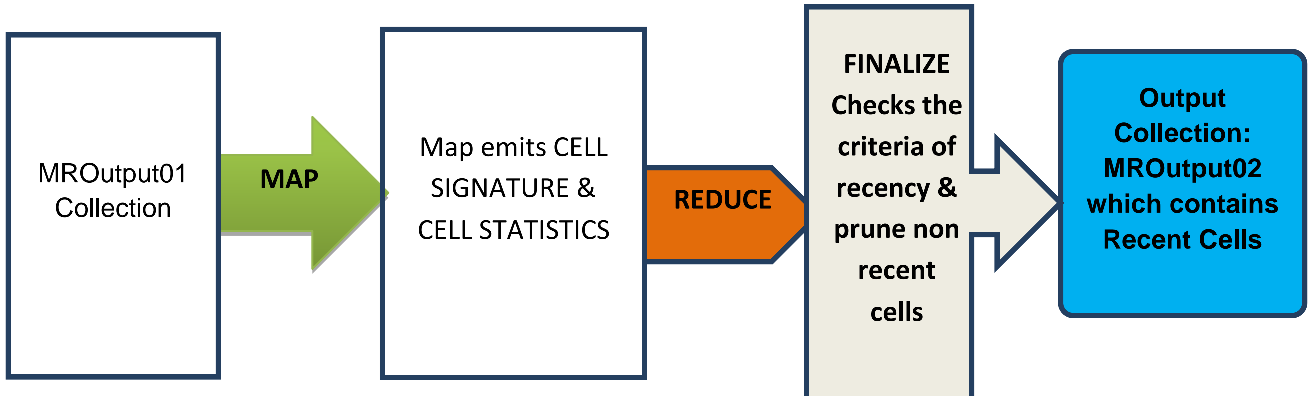
10. ACKNOWLEDGEMENT

This work was supported by Dr. Vasudha Bhatnagar, Reader, Department of Computer Science, University of Delhi.

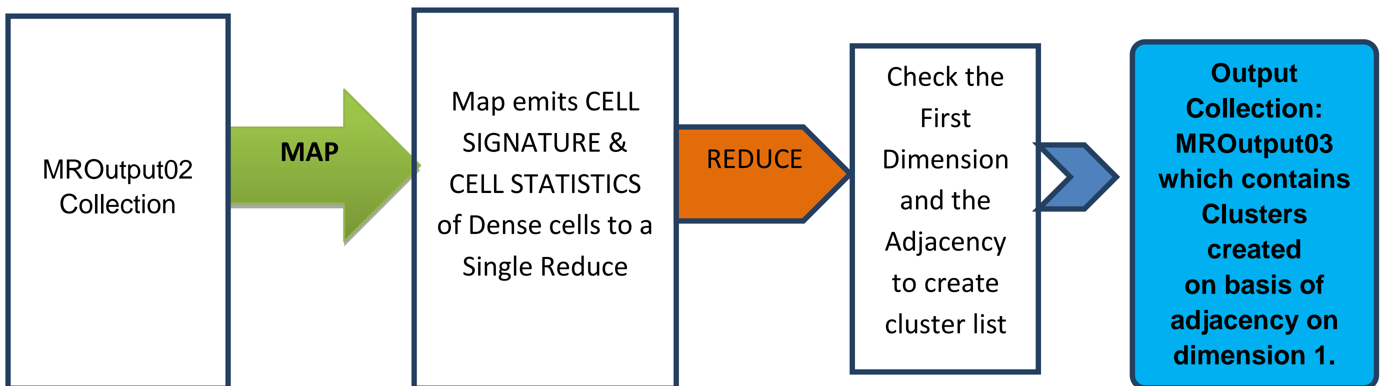
a. **JOB 1: Cell Signature and Statistics Generation Job**



b. **JOB 2: Synopsis Maintenance and Recent Cell Detection Job**



g. **JOB 3: Dense Recent Cell Detection and Clustering Phase 1 (Partitioning) Job**



d. **JOB 4: Clustering Phase 2 (Actual Clustering) and Clustering Phase 3 (Cluster Statistics) Job**

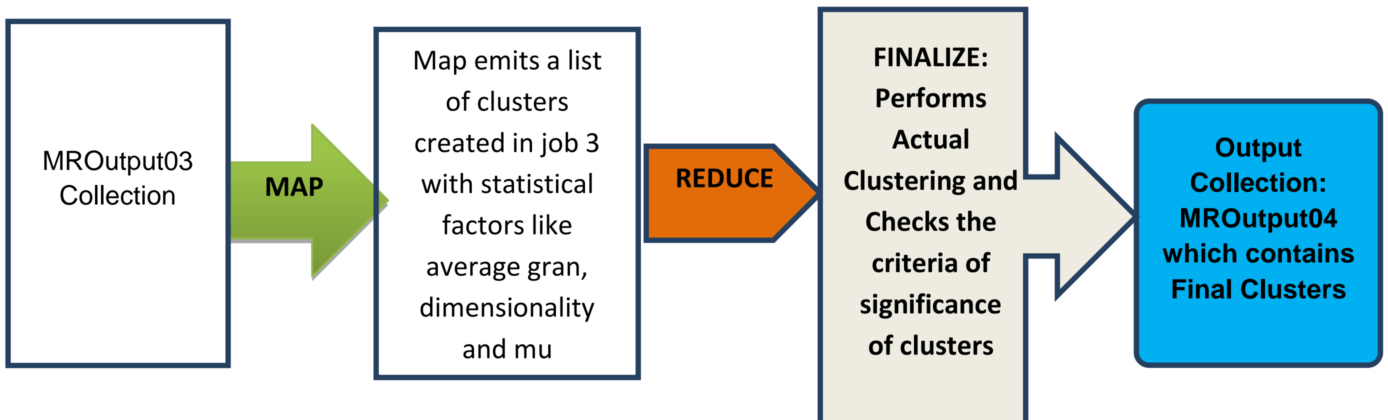


Fig3 Job-Wise illustration of ExCC implemented using MongoDB

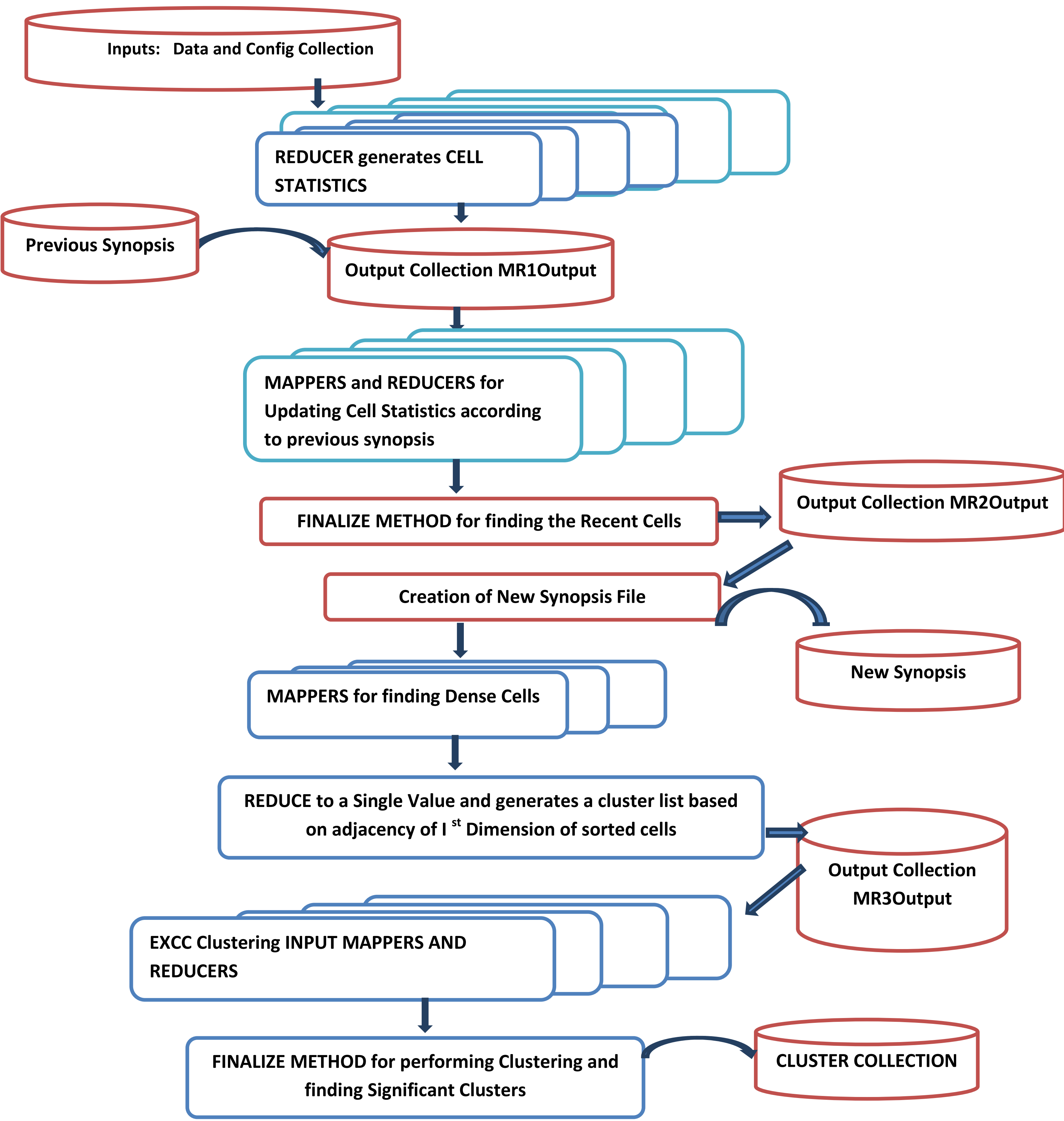


Fig4. Implementation flow of ExCC Algorithm with MongoDB

11. REFERENCES

[1] Aggarwal, C. C., Han, J., Wang, J., & Yu, P. S. (2003, September). A framework for clustering evolving data streams. In Proceedings of the 29th international conference on Very large data bases-Volume 29 (pp. 81-92). VLDB Endowment.

[2] Aggarwal, C. C., Han, J., Wang, J., & Yu, P. S. (2004, August). A framework for projected clustering of high dimensional data streams. In Proceedings of the Thirtieth international conference on Very large data bases-Volume 30 (pp. 852-863). VLDB Endowment.

[3] Antonellis, P., Makris, C., & Tsirakis, N. (2009). Algorithms for clustering clickstream data. Information Processing Letters, 109(8), 381-385.

[4] Bhatnagar, V., & Kaur, S. (2007, January). Exclusive and complete clustering of streams. In Database and Expert Systems Applications (pp. 629-638). Springer Berlin Heidelberg.

[5] Bhatnagar, V., Kaur, S., & Chakravarthy, S. (2014). Clustering data streams using grid-based synopsis. Knowledge and information systems, 41(1), 127-152.

[6] Bifet, A., Holmes, G., Pfahringer, B., Kranen, P., Kremer, H., Jansen, T., & Seidl, T. (2010). MOA: Massive Online Analysis, a framework for stream classification and clustering.

[7] Bryant, R. E. (2011). Data-intensive scalable computing for scientific applications. Computing in Science & Engineering, 13(6), 25-33.

[8] Chen, C. P., & Zhang, C. Y. (2014). Data-intensive applications, challenges, techniques and technologies: A survey on Big Data. Information Sciences, 275, 314-347.

[9] Chodorow, K. (2013). MongoDB: the definitive guide. " O'Reilly Media, Inc.".

[10] Dean, J., & Ghemawat, S. (2008). MapReduce: simplified data processing on large clusters. Communications of the ACM, 51(1), 107-113.

[11] Gaber, M. M. (2012). Advances in data stream mining. Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery, 2(1), 79-85.

[12] Gao, J., Li, J., Zhang, Z., & Tan, P. N. (2005). An incremental data stream clustering algorithm based on dense units detection. In Advances in Knowledge Discovery and Data Mining (pp. 420-425). Springer Berlin Heidelberg.

[13] Kanoje, S., Powar, V., & Mukhopadhyay, D. (2015). Using MongoDB for Social Networking Website. arXiv preprint arXiv:1503.06548.

[14] KDD CUP 99 Intrusion Data: http://kdd.ics.uci.edu//databases/kddcup99/kddcup99.html.

[15] Lin, J., & Lin, H. (2009, August). A density-based clustering over evolving heterogeneous data stream. In Computing, Communication, Control, and Management, 2009. CCCM 2009. ISECS International Colloquium on (Vol. 4, pp. 275-277). IEEE.

[16] Lu, Y., Sun, Y., Xu, G., & Liu, G. (2005). A grid-based clustering algorithm for high-dimensional data streams. In Advanced Data Mining and Applications (pp. 824-831). Springer Berlin Heidelberg.

[17] Marr, B. (Feb 2014). A Talk on Big Data- the 5 Vs Everyone must know.

[18] Marz, N., & Warren, J. (2015). Big Data: Principles and best practices of scalable realtime data systems. Manning Publications Co.

[19] McAfee, A., Brynjolfsson, E., Davenport, T. H., Patil, D. J., & Barton, D. (2012). Big data. The management revolution. Harvard Bus Rev, 90(10), 61-67.

[20] MongoDB Documentation Retrieved May 2015. From http://www.mongodb.org/

[21] NoSQL Databases Retrieved May 2015. From http://nosql-database.org/

[22] Seguin, K. (2011). The Little MongoDB Book.

[23] Strauch, C., Sites, U. L. S., & Kriha, W. (2011). NoSQL databases. Lecture Notes, Stuttgart Media University.

[24] Sun, Z. (2013, November). A parallel clustering method study based on mapReduce. In 1st International Workshop on Cloud Computing and Information Security. Atlantis Press.