J Supercomput (2017) 73:4444–4465
DOI 10.1007/s11227-017-2027-5

A partitioning framework for Cassandra NoSQL database using Rendezvous hashing

Sally M. Elghamrawy[1,3] · Aboul Ella Hassanien[2,3]

Published online: 6 April 2017

Abstract Due to the gradual expansion in data volume used in social networks and cloud computing, the term "Big data" has appeared with its challenges to store the immense datasets. Many tools and algorithms appeared to handle the challenges of storing big data. NoSQL databases, such as Cassandra and MongoDB, are designed with a novel data management system that can handle and process huge volumes of data. Partitioning data in NoSQL databases is considered one of the critical challenges in database design. In this paper, a MapReduce Rendezvous Hashing-Based Virtual Hierarchies (MR-RHVH) framework is proposed for scalable partitioning of Cassandra NoSQL database. The MapReduce framework is used to implement MR-RHVH on Cassandra to enhance its performance in highly distributed environments. MR-RHVH distributes the nodes to rendezvous regions based on a proposed Adopted Virtual Hierarchies strategy. Each region is responsible for a set of nodes. In addition, a proposed bloom filter evaluator is used to ensure the accurate allocation of keys to nodes in each region. Moreover, a number of experiments were performed to evaluate the performance of MR-RHVH framework, using YCSB for database benchmarking. The results show high scalability rate and less time consuming for MR-RHVH framework over different recent systems.

✉ Sally M. Elghamrawy
sally_elghamrawy@ieee.org; sally@mans.edu.eg
http://www.egyptscience.net

Aboul Ella Hassanien
aboitcairo@gmail.com
http://www.egyptscience.net

1 MISR Higher Institute for Engineering and Technology, Mansoura, Egypt

2 Faculty of Computers and Information, Cairo University, Giza, Egypt

3 Scientific Research Group in Egypt (SRGE), Giza, Egypt

Keywords Cassandra · Rendezvous hashing · Consistent hashing · MapReduce · NoSQL databases · Partitioning

1 Introduction

The rapid evolution and speedy success of internet technology and social networks led to the appearance of Big data [1] term. Facebook is a global service of immense scale; 2.4 billion content items are shared on Facebook among friends every day [2]. There were serious limitations in using relational database management systems RDBMS, with ACID properties [3], for indexing these big data. One of the main reasons is that RDBMS depend on nonflexible modeling schemes. These limitations have been the driving force behind the emergence of NoSQL [4] (meaning 'not only SQL') databases. NoSQL databases are used for storing big data due to its ability to expand easily according to the data scale. The data in NoSQL are huge and growing rapidly. It has high availability, high scalability, and high fault-tolerance, especially in social network applications. To manage these data, the data in NoSQL databases must be partitioned through distributed nodes.

Data partitioning techniques are one of the critical success factors in databases design. Sharding is the horizontal partitioning of data; it means the ability to shard the database and then distribute data stored in each shard. Range, random and hashing partitioning techniques are presented for NoSQL databases to partition data on different nodes. Some NoSQL databases like Apache HBase [5], MongoDB [6] use range partitioning, while other NoSQL databases like Google BigTable [7], Amazon Dynamo [8], Cassandra [9] used by Facebook use hashing partitioning, consistent hashing [10], as their partitioning strategies. The basic consistent hashing used in Cassandra NoSQL databases presents a scalability and load balancing limitations. Because it ignores the nature of nodes being assigned the data to them, it only depends on blind hashing. When using old Cassandra's default practitioner BYTEORDER, the distribution of load is extremely not uniform. For example, when a node is down its load will be distributed to the closest node regardless of the capabilities of neighboring nodes. This leads to a non-uniform distribution. As a result, the latest version of Cassandra tries to solve this by using the concept of virtual nodes VN to balance load in a more efficient way. But using VN will cause some challenges that must be handled like: (a) implementation complexity, (b) update and upgrade difficulties, (c) cost factors. All these challenges affect the overall performance of Cassandra. There have been many researchers [11–14] attempt to enhance in partitioning NoSQL databases. Most of the existing data partitioning strategies have great limitations such as low performance, load balancing problem, low scalability, and hotspots. Great efforts had been done to adopt in consistent hashing to solve load balancing problem. Unlike consistent hashing, Rendezvous hashing (HRW) uniformly distributes records of database over nodes using spooky hash function.

In this paper, a MapReduce Rendezvous hashing-based Virtual Hierarchical (MR-RHVH) framework is proposed for partitioning Cassandra NoSQL database. Its main goal is to enhance Cassandra's partitioning performance by using Rendezvous hashing that uniformly distributes records of the database over nodes, unlike consistent hashing.

The basic idea of Rendezvous hashing based on the Highest Random Weight (HRW) algorithm, first proposed by [15], is giving each node a weight for each key and assign the key to the highest weight. This method shows some imbalance in the load due to the heterogeneity of node's load and capability. Using the skeleton hierarchical by traditional HRW leads to lack of performance due to the bottlenecks appeared when choosing and requesting the upper level's nodes. The implementation of the MR-RHVH is based on the proposed Adopted Virtual Hierarchies (AVH) structure using the proposed Improved Highest Random Weight (IHRW) algorithm. In contrast, using the AVH structure and the IHRW algorithm in MR-RHVH enhances Cassandra performance by allowing the system to take the benefits of hierarchies without causing any bottlenecks in the upper level, due to its ability to divide the system into rendezvous regions to ensure the locality advantage. The "spooky" hash function is used in MR-RHVH instead of the "Murmur" hash, used by Cassandra, as it is proved that to be twice the speed of Murmur hash.

In addition, a Load Balancing algorithm-based Rendezvous hashing LBRH is proposed to manage the balancing between nodes in the partitioning process. MR-RHVH enhances the timing of hashing by using a bloom filter evaluator in every node of the rendezvous hash range. In addition, using a spooky hash function enhances timing of hashing. The rest of the paper is organized as follows. Section 2 shows recent NoSQL systems discussing different strategies in partitioning NoSQL databases. Section 3 presents and demonstrates the proposed MapReduce Rendezvous hashing-based Virtual Hierarchical MR-RHVH Partitioning framework with its associated algorithms. The performance of Cassandra is evaluated in Sect. 4 showing the effect of implementing MR-RHVH on Cassandra, and the results are compared against recent work's results using the Yahoo Cloud Serving Benchmark [16].

2 Related work

There are enormous interests of researchers in investigating the partitioning strategies of NoSQL databases due to its impact on the performance of systems. Hash, range and hybrids between hash and range partitioning are the most common strategies used by NoSQL systems. A number of researchers have developed different methods to enhance Cassandra performance and in partitioning strategies of different NoSQL databases: Ata Turk et al. [12] proposed data partitioning method based on hypergraph that is constructed to correctly model multi-user operations. This method utilizes the temporal information in prior workloads to try to enhance the Cassandra NoSQL. Abramova et al. [17] analyzed the scalability of Cassandra by testing the replication and data partitioning strategies used. Lakshminarayanan [18] proposed an adaptive partitioning scheme for consistent hashing that effect in the heterogeneity of the systems. Huang et al. [13] proposed a dynamic programming algorithm in the consistent hashing ring based on imbalance coefficient for Cassandra cluster to calculate the position of the new coming node. Wang and Loguinov [19] proposed a greedy algorithm to position the new node in the largest range of the space by splitting the range into two pieces. Ramakrishnan et al. [20] proposed a processing pipeline, using the Random Partitioner of Cassandra, to allow any non-Java executable makes use of the NoSQL and allowing offline processing for Cassandra. Kuhlenkamp et al. [21] benchmarked the scalability and elasticity of Cassandra and HBase. Zhikun et al. [11] proposed a Hybrid Range Consistency Hash (HRCH) partition strategy for NoSQL database to improve the degree of processing and the data loading speed.

Most of these approaches either consider a scalability test of the NoSQL database or enhancing in consistent hashing of Cassandra. Cassandra had three basic partitioning strategies: Random Partitioning using the MD5 hash of each row key, ByteOrdered and Murmur3 partitioning. To the best of our knowledge, it is the first attempt to replace the whole partitioning module of Cassandra with the rendezvous hashing technique and test their scalability, load balancing, and timing of partitioning process.

Braam et al. [22] proposed a Lustre as a storage architecture based on hash partitioning. But in Lustre the files located in the same directory are distributed to different locations, causing the locality principle of namespace to be lost. Furthermore, the processing timing is increased due to the huge amount of data that changes its locations.

3 The proposed MapReduce Rendezvous hashing-based virtual hierarchical (MR-RHVH) framework

In this section, the proposed MapReduce Rendezvous hashing-based Virtual Hierarchical (MR-RHVH) framework is presented. Its main goal is to enhance Cassandra's partitioning performance by using Rendezvous hashing, instead of the consistent hashing. The traditional HRW [15] method shows some imbalance in the load could be due to heterogeneity of the nodes. In addition, using the skeleton hierarchical, described in [23], leads to a lack in performance due to the bottlenecks appeared when choosing and requesting the upper levels nodes. In contrast, the proposed MR-RHVH enhances Cassandra performance by: (a) Using the Adopted Virtual Hierarchies strategy based on the proposed Improved Highest Random Weight (IHRW) algorithm allows the system to take the benefits of hierarchies without causing any bottlenecks in the upper level, because IHRW divides each hierarchical into a rendezvous regions that allow the nodes to be more flexible in selecting the master nodes, depends on region's loads and requirements. (b) Proposing a Load Balancing algorithm-based Rendezvous hashing LBRH to manage the balancing between nodes in the partitioning process. Enhance the timing of hashing by using bloom filter method in every node of the rendezvous hash range. (c) Using spooky hash function enhances timing of hashing. (d) Monitoring the load in the system by analyzing the workload in each node and caches them. MR-RHVH framework applies the distributed structure of nodes using MapReduce, that equally partition data among Cassandra nodes using mapper and reducer functions. The MR-RHVH framework consists of four main layers, as shown in Fig. 1, Cassandra/Hadoop Cluster, MR-RHVH, Cassandra/Hadoop Data Center and Hadoop/MapReduce applier.

3.1 Cassandra/Hadoop data center

Cassandra clients' nodes are distributed in this data center. Task Tracker and Data Node services run on each Cassandra node/client in the data center. The Task Tracker accepts

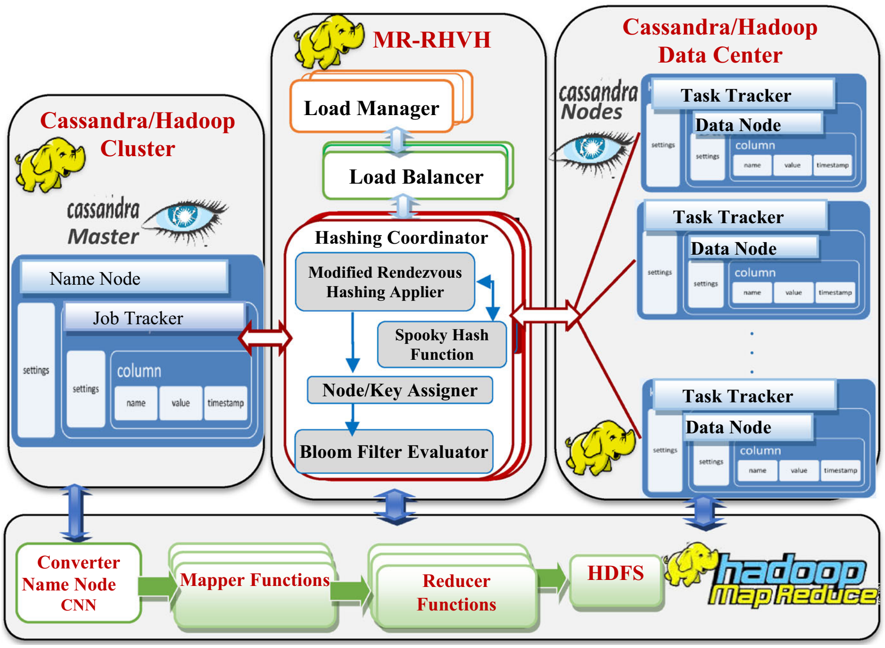

Fig. 1 The MapReduce Rendezvous hashing framework based virtual hierarchical (MR-RHVH) framework

tasks from job tracker and then recall data needed from data node. The Data Node used to provide task trackers with the required data, using HDFS in MapReduce layer.

3.2 Cassandra/Hadoop cluster

The Cassandra master node is located in this layer. In Cassandra/Hadoop Cluster, the Job Tracker service, running on the master nodes, is used to coordinate job requests sent to and from the Task Trackers in client's node using MapReduce. The name node in the Cassandra master node is used to save a list of all the files in the data center, and search for the node that keep the file or have the capability to save a file. Name node considered as a single point of failure (SPOF) in Hadoop/MapReduce, as when the name node failed, the whole system goes down. A converter name node (CNN) module is proposed to solve SPOF in the next layer.

3.3 Hadoop/MapReduce applier

Hadoop/MapReduce service runs on each Cassandra node. It consists of: A converter name node (CNN), Hadoop Distributed File System (HDFS), used for data storage, mapper and reducer functions used by MR-RHVH.

A converter name node (CNN) module is used as a secondary name node that is mapped to the storage structure of Cassandra NoSQL database. It is used to separate

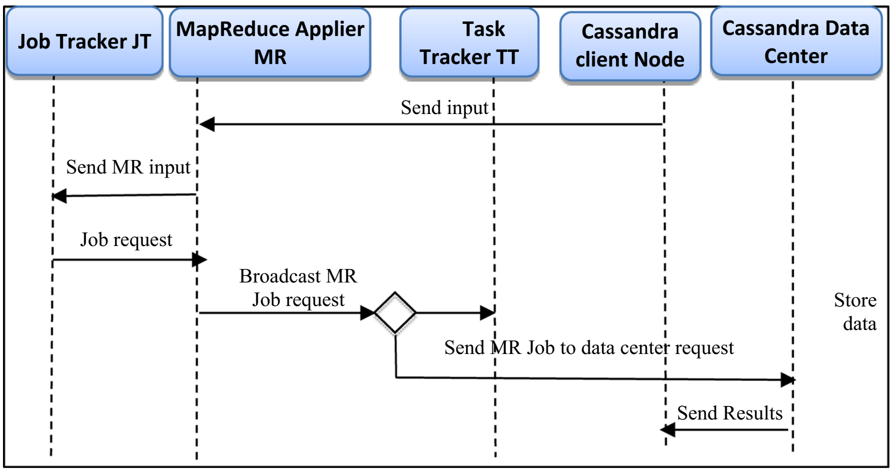

Fig. 2 Sequence diagram of MR-RHVH layers workflow

the requests, sent from the data nodes to name node, from the data itself. In MR-RHVH, firstly all requests are submitted via MapReduce functions to the CNN and then the CNN updates the data stored on the name node located in the master node.

A sequence diagram of MR-RHVM workflow is shown in Fig. 2. Any Cassandra Node (CN) sends an input request to the Job Tracker (JT) of Cassandra master node through the MapReduce applier MR. Then the JT sends a job request to MR, that broadcast the request to the Task Trackers (TT) of nodes in the data center. The data are stored to Cassandra datacenter, and results of requests are sent back to the CN, that firstly send the request.

3.4 MR-RHVH layer

This layer implementation is distributed across the nodes using MapReduce applier layer, and it consists of three main modules:

- **Load Manager** It monitors the load on the system by analyzing the workload in each node and caches them. In addition, the load requests and the most frequently appeared node in a specific Cassandra cluster are managed, based on Heuristic 1.

Heuristic 1 Suppose that there is a key with name K and a set of nodes N $= \{n_1.n_2.n_3.\dots.n_i\}$. The load here is the Cassandra data that needs to be partitioned to the nodes, and it can be defined in terms of $= <$*hashes* $\mathcal{H}$, *Keys* $\varkappa$, *Queries* $\mathbb{Q}$, *Tuples* T$>$. Each node n $\in N$ is defined in terms of $=$ <CAPACITY(n), ACTIVE DEGREE (n), DEPENDENCY (n)>, where capacity(n) indicates the ability of the node to accept a number of keys and queries by checking the current load on that node. Active degree(n) indicates the activity degree of the node (e.g., the number of finished queries). It stores the number of assigned keys and queries that the node performed. Dependency(n) is concerned with the Cooperation Degree of the node with respect to other nodes in the system, i.e., how many times the node asks the help of another node.

- **The LoadBalancer** As we have mentioned before, the basic consistent hashing algorithm used in Cassandra affects the overall performance of Cassandra DB due to load balancing problem. The load balancer sub-module main responsibility is to manage the balancing in the partitioning process without affecting the performance. Its main goal is to implement the proposed Load Balancing-based Rendezvous Hashing algorithm **LBRH**, shown in Fig. 3. It also resolves all conflicts that may occur.
  The LBRH algorithm, shown in Fig. 3, proceeds as follows: the set of the participating nodes in the cell $\boldsymbol{C_j}$ are supervised to calculate the current load for each node ($\mathbf{n_i}$). Then the calculated load is compared with two threshold values as a high-value threshold $\boldsymbol{\mathcal{T}_{\mathcal{H}LD}}$ and low-value threshold $\boldsymbol{\mathcal{T}_{\mathcal{L}LD}}$. If the current load of a node is bigger than the $\boldsymbol{\mathcal{T}_{\mathcal{H}LD}}$, then the node is categorized as "intense load" node. And its load must be splitted and partitioned to the participating nodes in the cell based on their loads. In contrast, if the current load of a node is lower than the $\boldsymbol{\mathcal{T}_{\mathcal{L}LD}}$, then this node is categorized as "low-key load" node and will bid for load more loads based on the Bid-Bonus algorithm. There are two scenarios for balancing the load when using the rendezvous (HRW) hashing:

Scenario 1 To uniformly balance the load when a new key request needed to be allocated to a specific node based on balancing algorithm.

Scenario 2 When a node is down, its load must be uniformly distributed across the other nodes participated in the system. The process of balancing the load when partitioning the database must deal with all the attributes for node and keys. Heuristic 2 illustrates some terms used in the LBRH algorithm.

Heuristic 2 Suppose that there is a set of nodes N = $\{\boldsymbol{n_1.n_2.n_3....n_i}\}$ in each cell of the HRW hierarchic. The capacity (n) in Heuristic 1 is calculated in terms of $< \boldsymbol{LD_i^{Cur}}(n), \boldsymbol{LD_i^{MaX}}(n) >$. Where $\boldsymbol{n_i}$ is the node number i in the cell and $\boldsymbol{LD_i^{Cur}}$ is the current load that the node holds right now and the $\boldsymbol{LD_i^{MaX}}$ is the maximum load that the node i can hold calculated based on each node feature. There are two main phases for the process:

First Phase Let ($n_i \in$ N) be the set of the participating nodes in the cell C_j in HRW structure. And CC_{ji} be the set of cell coordinator for each cell responsible of a number of node i. Each CC_{ji} has a load denoted by CC_{LD}^{ji}, and the cell coordinator's main goal is to manage the load between the nodes in this cell based on each node's LD_i^{Cur} and LD_i^{MaX}, to categorize the nodes to intense load, moderate load, and low-key load, as shown in the LBRH algorithm. By using two threshold values as a high-value threshold $\mathcal{T}_{\mathcal{H}LD}$ and low-value threshold $\mathcal{T}_{\mathcal{L}LD}$.

To allocate the key, it must have a popularity factor (•). While the load is assigned to a specific node, the Load Balancing algorithm will be activated to ensure that the load is balanced by giving φ the minimum value, shown in Eq. (1).

$$\varphi = \sum_{i=1,n}^{j=1,m} CC_{LD}^{ji} - \frac{\sum_{j=1}^{m} CC_{LD}^{ji}}{\sum_{i=1}^{n} LD_i^{MaX}} \tag{1}$$

```
LBRH. Load Balancing based Rendezvous Hashing
    Input: Node Tuples's ID ($\boldsymbol{Node_i}$(CAPACITY($n_i$), ACTIVE DEGREE($n_i$), DEPENDENCY($n_i$)) and node ($n_i$)'s
neighbors' in same zone according to HRW
    Output: A List of the distributed load assigned to which node ($\boldsymbol{LD_{DS_j}, n}$)

1.      Supervise ($\sum_{i=1}^{x}(n_i)$ $in$ $n_i$) $\in C_j$
2.      For Each n in $\sum_{i=1}^{x}(n_i)$
3.          Calculate $LD_i^{Cur}(n_i)$
4.          Check load ($n_i$)
5.          IF $LD_i^{Cur} > \mathcal{T}_{\mathcal{HLD}}$ IS OVERLOAD THEN INTENSE-LOAD ( )
6.          INTENSE-LOAD ( )
              {
7.              aa : Register→ Pending_Relaod_List()
8.              $LD_i^{Cur}$=fragment ($\mathcal{LD}$)
9.              If $LD_i^{MaX} < \mathcal{T}_{\mathcal{HLD}}$
10.                 Choose specific (n(i))
11.                 Split (n(i))
12.                 Partition (n(i))
13.                 Bid_Bonus ($LD_i^{MaX}, LD_{i+1}^{MaX}, LD_{i-1}^{MaX}$ )
14.             Else Go to aa
15.             Combine (n)
16.             Create ($LD_{DS_j}, n$) LIST
17.           }
18.       End if
19.       If $LD_i^{Cur} < \mathcal{T}_{\mathcal{LLD}}$ is overload then LOW_KEY-LOAD()
20.             Low_KEY-LOAD( )
21.             {
22.               $LD_i^{Cur}=LD_i^{Cur} - \mathcal{T}_{\mathcal{LLD}+i}$
23.               For each $CC_{LD}^{ji}$ in $C_j$
24.                 Bid_Bonus($LD_i^{Cur}, LD_{i+1}^{Cur}, LD_{i-1}^{Cur}$)
25.                 Minimize ($\varphi$)
26.               End For
27.               Create ($LD_{DS_j}, n$) LIST
28.             }
29.       End if
30.     EndFor
```

Fig. 3 The LBRH algorithm ()

Second Phase Each node in the cell has neighbor node, and the main goal of the LBRH algorithm is to predict the neighbor node performance in allocating the load by using a bid-bonus algorithm proposed in Fig. 4.

Hashing coordinator It is the core module of MR-RHVH. It is responsible for the partitioning process, based on rendezvous hashing, using the spooky hash function. It consists of four main submodules: Modified Rendezvous (HRW) Hashing applier: In this sub-module, the proposed Adopted Virtual Hierarchies (AVH) strategy, shown in Fig. 5, is implemented using the proposed Improved Highest Random Weight IHRW hashing algorithm, based on virtual hierarchies' skeleton on HRW is implemented.

The Bid bonus algorithm used by LBRH

```
1.  Bid_Bonus( )
2.  {
3.  For Each $n_i$(Cap) in $C_i$do
4.  CreatBID($n_i$)
5.   If $bid_i$ (Cap)=trustybids[$i$] then
6.   If Expct{$bid_i$ (Cap)} > Higest(Cap) then
7.   $bid_i$ (Cap) → $higestbidlist[i]$
8.   Else if
9.   Recieved {$bid_i$ (Cap)} > Higest(Cap) then
10.  $bid_i$ (CCap) → $higestbidlist[i]$)
11.  End If
12.           For each $bid_i$ (Cap)in $higestbidlist[i]$
13.              if $bid_i$ (Cap)=trustybids[$i$]then
14.              Sort ($bid_i$ (Cap)) → $sortedbidlist[i]$
15.              sendnew ($bid_i$ (Cap) )
16.              End If
17.           Next
18.        For each $bid_i$ (Cap) in $sortedbidlist[i]$
19.           Get HigestPairBids ($bid_i$ (Cap)) → $HighestPair[i]$
20.           If $HighestPair[i]$ > < $\mathcal{T}_{\mathcal{L}LD}$ then
21.              For first $bid_i$ (Cap, AD, Dep) in $HighestPair[i]$
22.               Get ($bid_i$ (AD))
23.               If $n_i$-ActDgree ($bid_i$( AD)> $n_i$-ActDgree$bid_{i+1}$ (AD)
24.                then $bid_i$ (AD)) → Final$HighestPair[i]$
25.                If $n_i$-ActDgree($bid_i$ (AD)< $n_i$-ActDgree($bid_{i+1}$(AD)
26.                then $bid_{i+1}$ (AD) → Final$HighestPair[i]$
27.                Else
28.                Get($n_i$-Dependency ($bid_i$ (Dep))
29.                If $n_i$-Dependency ($bid_i$ (Dep)< $n_i$-Dependency ($bid_{i+1}$ (Dep) then
30.                $bid_i$ (Dep)) → Final$HighestPair[i]$
31.                If $n_i$-Dependency ($bid_i$ (Dep)> $n_i$-Dependency ($bid_{i+1}$ (Dep) then
32.                $bid_{i+1}$ (Dep)) → Final$HighestPair[i]$
33.                Else
34.                RandomPair($bid_i$ (Cap, AD, Dep)) → Final$HighestPair[i]$
35.               End If
36.              Next
37.           End If
38.        Next
39. Next
```

Fig. 4 The Bid-bonus algorithm used by LBRH

Each zone is constructed as a virtual hierarchy that is geographic specific. Each zone has a predefined number of nodes and a coordinator $\boldsymbol{C(i)}_{11}^{1}$ to these nodes. The coordinator is assigned based on the proposed IHRW hashing algorithm, as illustrated in Heuristic 3.

Heuristic 3 The adaptive HRW protocol proposed in Fig. 6 is implemented at each level of the virtual hierarchical. Suppose the system contains n_{od} nodes that will be divided into a number of Rendezvous Geographic Zones ($\mathbb{R}\mathcal{G}_{\mathbb{Z}}$) based on their geographic area. Each zone will have a number of nodes n_{od} ranges from $\max_i$ *celi* to

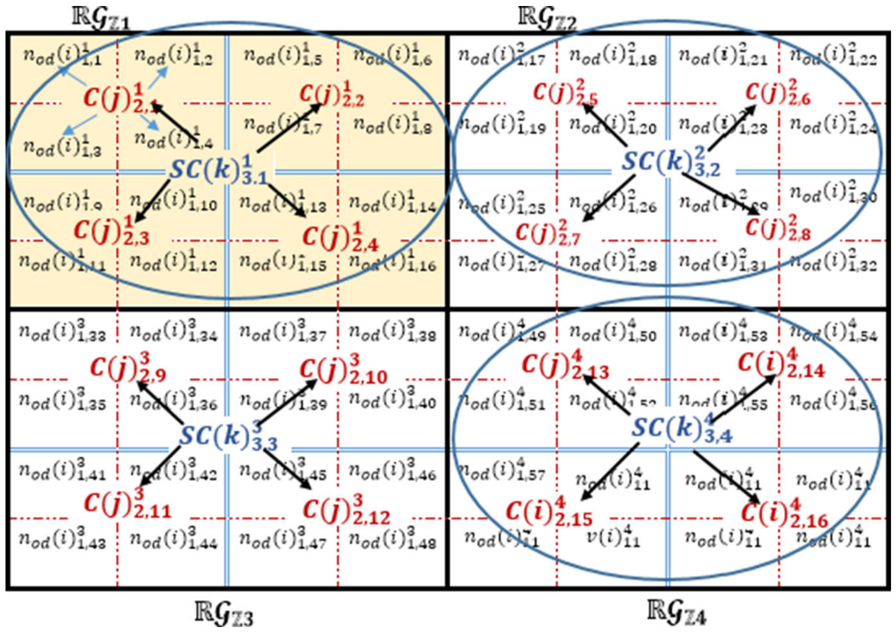

Fig. 5 The hierarchal structure of nodes organized

$\min_i$ *flo* depend on their area load. This will achieve O(log(*AVG* [$\max_i$ *celi*, $\min_i$ *flo*]) running time.

These zones are categorized as follows:

$$\mathbb{R}\mathcal{G}_{\mathbb{Z}1} = \left\{n_{od1}.n_{od2}.....n_{od_{flo\to celi}}\right\},$$

$\mathbb{R}\mathcal{G}_{\mathbb{Z}2} = \{n_{od_{flo+1\to celi+1}}.n_{od_{flo+2\to celi+2}}.....n_{od_{2flo\to 2celi}}\}$, … and so on. Then, instead of hashing all the nodes in the system, we start hashing each node in a specific zone by using the spooky hash function illustrated in the next subsection; then based on the hashes obtained, the nodes are organized in levels (level 1–3 based on number of nodes) and the node with the highest weighted hash in the upper level will be assigned as coordinator of this level $\boldsymbol{C(j)^n_{m.x}}$, where n is the zone number, m is the level the coordinator in and x is the coordinator ID. Then the coordinators' nodes are hashed and the highest weighted hash in them will be assigned as Senior Coordinator $\boldsymbol{SC(k)^n_{m.x}}$ (ex. $\boldsymbol{SC(k)^2_{3.2}}$) for this zone, as shown in Fig. 5.

The IHRW algorithm, shown in Fig. 6, partitions keys of the Cassandra database using rendezvous hashing (instead of the original consistent hashing) on the nodes distributed in a sufficient way to guarantee balanced partitioning. The nodes in a Cassandra cluster are divided into a number of Rendezvous Geographic Zones ($\mathbb{R}\mathcal{G}_{\mathbb{Z}}$) [24], and the data are distributed according to the virtual hierarchical design in each zone, as depicted in Fig. 6.

Spooky hash function The default hash function used by the original partitioning module in Cassandra is the Murmur Hash [25] as it is used in the 32-bit. The main

IHRW: The Improved HRW algorithm

Input: The Nodes $n_{od}(i)^{n}_{m.x}$ in a Cassandra cluster $\mathbb{CC}$ ***with their corresponding IP showing their geographic Areas $g\text{Å}$.***

Output: A List of the categorized nodes in hierarchical structure

```
For Each $n_{od}(i)^{n}_{m.x}$ in $\mathbb{CC}$
  {  ParseArea.Identifier IP( $n_{od}(i)^{n}_{m.x}$)
     LISTS OF ($n_{od}(i)^{n}_{m.x}$, $g\text{Å}$)
  }
Connect (ListS[ $n_{od}(i)^{n}_{m.x}$, $g\text{Å}$])
REGISTER $\sum g\text{Å} \rightarrow$ LIST ($g\text{Å}$, $\mathbb{R}\mathcal{G}_{\mathbb{Z}n}$)
Assign ( $\sum_{i=celi}^{flo} n_{od}(i)^{n}_{m.x}$ to $\mathbb{R}\mathcal{G}_{\mathbb{Z}n}$)
Xx :
     For Each $n_{od}(i)^{n}_{m.x}$ in $\mathbb{R}\mathcal{G}_{\mathbb{Z}n}$
        {     Arrange ($n_{od}(i)^{n}_{m.x}$) in $\mathbb{R}\mathcal{G}_{\mathbb{Z}n}$
              For (i=min$_i$ $flo$ to max$_i$ $celi$)
                 {  Assign$n_{od}$ ( Zone#(n), Level#(m), Node#(x))
                    Hash = SH ( $n_{od}(i)^{n}_{m.x}$)
                    TableHash[ ] += Hash
                 }
        }
   IF ( i > max$_i$ $celi$ AND $n_{od}(i)^{n}_{m.x} \in \mathbb{R}\mathcal{G}_{\mathbb{Z}n}$)
   {    DUPLICATE $\mathbb{R}\mathcal{G}_{\mathbb{Z}n}$
        go to xx
   }
SELECT Max$_{flo<i<celi}$ SH ( $n_{od}(i)^{n}_{m.x}$)
ASSIGN MAX $\rightarrow$ Coordinator ($C(j)^{n}_{m.x}$)
     For (j=min$_i$ $flo$ to max$_i$ $celi$)
   {  Assign  $C$ [ Zone#(n), Level#(m), Node#(x)]
      Hash = SH ( $C(j)^{n}_{m.x}$))
      TableHash[ ] += Hash
   }
SELECT Max$_{flo<i<celi}$ SH ( $C(j)^{n}_{m.x}$)
ASSIGN MAX $\rightarrow$ SeniorCoordinator ($SC(k)^{n}_{m.x}$)
```

Fig. 6 The adaptive HRW protocol proposed

goal is to implement the non-cryptographic Spooky Hash [26] function on the required nodes and keys in Cassandra, based on Heuristic 4.

Heuristic 4 The spooky hash function calculates the weight of each node when assigning key K to it $W_i^K = \{w_1^K.w_2^K.\ldots w_n^K\}$ where $W_i^K = \dot{\dot{\text{S}}}\text{H}(n_i\,.K)$ where n_i is the node identified in Heuristic 1. Rendezvous algorithm or HRW main characteristic is to assign the Key to node n_k yielding the highest weight, that is, such that $W_x^K = MAX(w_1^K.w_2^K.\ldots w_n^K)$. Since $\dot{\dot{\text{S}}}\text{H}$ and the nodes IP are predefined, the rendezvous algorithm will independently automatically compute identical K/node assignments.

One of the main reasons for using the spooky hash instead of Murmur hash is that later proved to be half the speed of spooky hash on x86-64. In addition, using spooky

hash speed in the hashing process due to its ability to hash any keys with bytes' array form, as it can produce many different independent hashes for the same key [26].
Node/key assigner The spooky hash function outputs a list of the hashed nodes, and the hashed key needs to be allocated; the main responsibility of the node/key assigner is to implement the process of assigning the hashed key to the node yielding the highest weight.

Bloom filter evaluator Bloom filters [27] are widely accepted in the database [28,29] applications for membership queries because it dramatically decreases the representation size of the set of elements; however, they generate false positives. The bloom filter evaluator is used in every node of the rendezvous hash range for two main goals: (1) To ensure the accurate allocation of keys to nodes and checking the load balancing while allocating. (2) To reduce the time taken for the load distribution in the load balancer module.

In this module, the bloom filter is used as follows: Suppose the nodes in Cassandra system is categorized based on IHRW algorithm into rendezvous geographic zones each of which consists of n node. $\mathbb{R}\mathcal{G}_{\mathbb{Z}1} = \left\{n_{od1}.n_{od2}.....n_{od_n}\right\}$, these nodes are organized in an array of n bits $\beta\digamma[n]$, as shown in Fig. 7.

All bits in the filter are initialized to '0', as depicted in Fig. 7, $\beta\digamma[n] = 0$.Then bloom filter uses k spooky Hash functions $\dot{\mathrm{S}}\ddot{\mathrm{H}}_k$ to calculate the hash value and then store these hashes in the m-bit array $\beta\digamma[\dot{\mathrm{S}}\ddot{\mathrm{H}}_k(m)]$. Each $\dot{\mathrm{S}}\ddot{\mathrm{H}}_k$ map n_{odn} and put the hashes results in the m array with any order. Then the information is that the $k_{ey}(19)$ is assigned to the $\mathbb{R}\mathcal{G}_{\mathbb{Z}1}$ in the node $SC(k)^2_{3.2}$. So, the $k_{ey}(19)$ and $SC(k)^2_{3.2}$ is then hashed. The query now is $k_{ey}(54) \in \mathbb{R}\mathcal{G}_{\mathbb{Z}1}$? so check if all $\dot{\mathrm{S}}\ddot{\mathrm{H}}_k(k_{ey}(19)$ set to 1. If not, $k_{ey}(19)$ is not a member of $\mathbb{R}\mathcal{G}_{\mathbb{Z}1}$. That means that the key is not correctly assigned to the appropriate zone. The membership query introduces a false positive by checking the presence of element $k_{ey}(54)$ in set $\mathbb{R}\mathcal{G}_{\mathbb{Z}1} = \left\{n_{od1}.n_{od2}.....n_{od_n}\right\}$.

In addition, the load balancer module uses the bloom filter evaluator in creating a list of the distributed load assigned to which node ($\boldsymbol{LD_{DS_j}.n}$) by determining whether the load belongs to this node in this zone or not based on the Adaptive HRW protocol. Without using bloom filter, this load membership test needs O(log(AVG n[max$_i$ $celi$, min$_i$ flo] running time depends on the number of nodes in each geographic zone while using bloom filter will shorten this time. Since the bloom filter requires a single scan of the data, its construction is less time consuming for storing elements of a set.

4 Performance evaluation

A number of experiments are conducted to evaluate the influence of the proposed MR-RHVH partitioning module in Cassandra NoSQL database.

4.1 Environments and benchmark

The Apache Cassandra version 3.4 is modified by embedding the MR-RHVH in it. The standard Yahoo! Cloud Serving Benchmark (YCSB) is installed YCSB 0.1.4 on

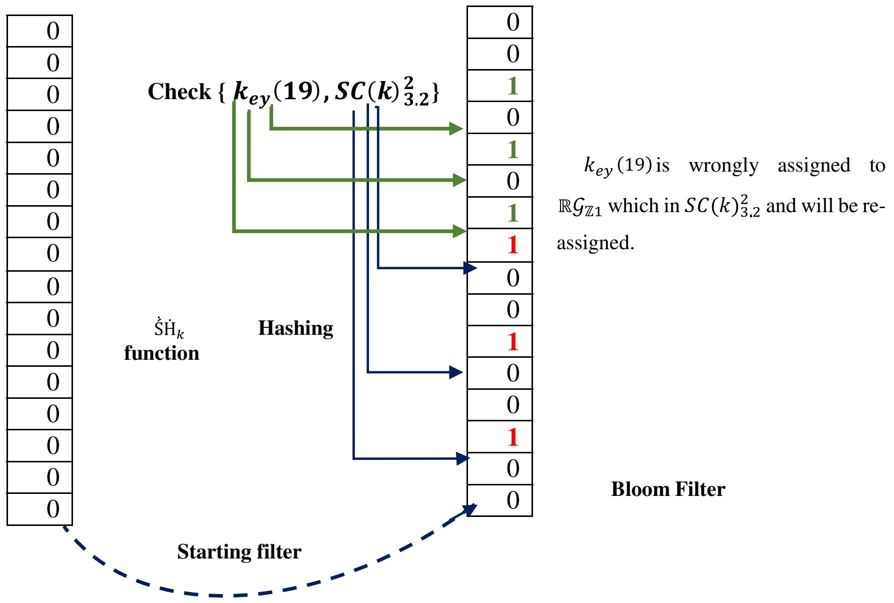
Check { $k_{ey}(19), SC(k)^{2}_{3.2}$}
$k_{ey}(19)$ is wrongly assigned to $\mathbb{R}\mathcal{G}_{\mathbb{Z}1}$ which in $SC(k)^{2}_{3.2}$ and will be re-assigned.
ŚḦ$_k$ function
Hashing
Bloom Filter
Starting filter

Fig. 7 Bloom filter evaluator example

Table 1 YCSB workloads

Workload name	Job
Workload A update heavy	50% read and 50% write
Workload B (read heavy)	95% read and 5% write
Workload C (read only)	100% read
Workload D (read latest)	Reads latest workloads
Workload E (short ranges)	Scans short ranges of records
Workload F (read–modify–write)	Clients read, modify and finally writes back

a client node to test the performance. YCSB, designed by Yahoo Labs, is a popular open-source benchmark for comparing the performance of NoSQL databases [16]. Currently, YCSB do not support Cassandra 3.X yet; YCSB is upgraded to support Cassandra 3.4 version by using the DataStax Java Driver, illustrated in [30], for Cassandra. There are six types of workloads in YCSB, which is used to compare the performance of Cassandra, as shown in Table 1.

In our experiments, two clusters of 8 nodes/cluster are used (Node0 to Node7). The specification of Cassandra cluster nodes, Hadoop and Cassandra configurations are shown in Table 2. The nodes are created by using the VMware vSphere system [31] (VMware Inc., Palo Alto, CA, USA). The specification of Cassandra cluster nodes, Hadoop and Cassandra configurations is shown in Table 2.

Table 2 Cassandra cluster nodes specifications

Number of Cassandra clusters	2	All the nodes have a 200 GB disk space	
Number of nodes per cluster	8	Run at Ubunt10 system	
Node specification	Dual-core Intel Core i5-4200U at 2.6 GHz, 8 G of RAM, 200 GB disks and gigabit Ethernet, run	Every NoSQL database consisted of 8 million records	
The machine amount of every cluster	{5 clients,1 server}		
Hadoop version 2.6.4 installed on each node "Hadoop Configuration"		Cassandra 3.4 version configurations	
hadoop.hdfs.configuration.version	1	Row cache provider	SerializingCacheProvider
dfs.replication.max	512	Replication factor	1
dfs.namenode.replication.min	1	Heap size	1 GB
dfs.blocksize	128 MB	RPC timeout in MS	10,000

RHP Replicated hypergraph partitioning model; *RP* random partitioning; *MUR* murmur Partitioning

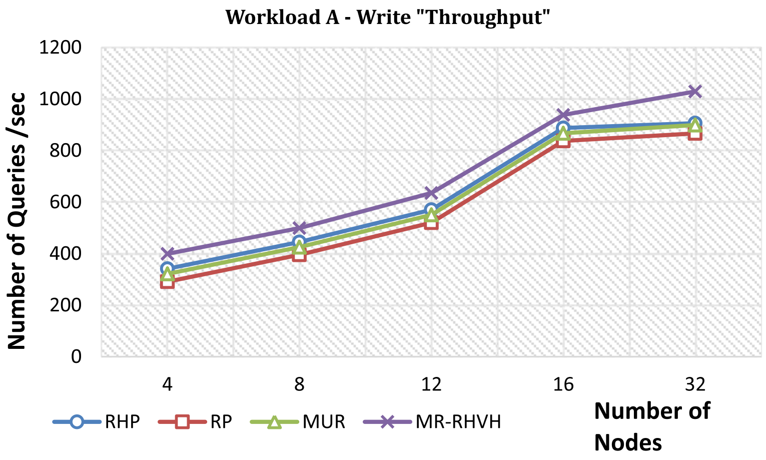

Fig. 8 The comparison of write throughput of workload A of MR-RHVH with recent systems

4.2 Experiments and evaluation

The experiments have been divided into four main parts for the evaluations, as follows:

- Evaluates the scalability in terms of throughput of the system after using the IHRW algorithm in MR-RHVH.
- Evaluates the latency of the system after using the IHRW algorithm.
- Evaluates the performance of the Load Balancing-based Rendezvous (HRW) Hashing algorithm LBRH implemented in the load balancer sub-module.
- Cassandra's response time, after implementing the MR-RHVH, is tested under different environments.

4.2.1 Throughput experiment

The scalability of the system is tested with increasing the number of nodes used and test the throughput by executing workload A and workload C over 8M records. The number of nodes is increased from 4 to 16 nodes to verify the variation of MR-RHVH throughput according to number of nodes in the cluster. Figure 8 shows the MR-RHVH throughput in workload A compared to recent systems RHP, RP, MUR [12,13,20].

Figure 9 shows the MR-RHVH throughput in workload C compared to recent systems RHP, RP, MUR [12,13,20].

4.2.2 Latency experiment

The latency of read and write operations for MR-RHVH, when varying number of nodes, is compared in the same environment of testing RHP, RH and MUR [12,13,20]. The results for workload A and C are shown in Figs. 10 and 11, respectively.

From the last two experiments, it is obvious that among comparative systems, Cassandra with MR-RHVH achieves the best latency and throughput values, compared

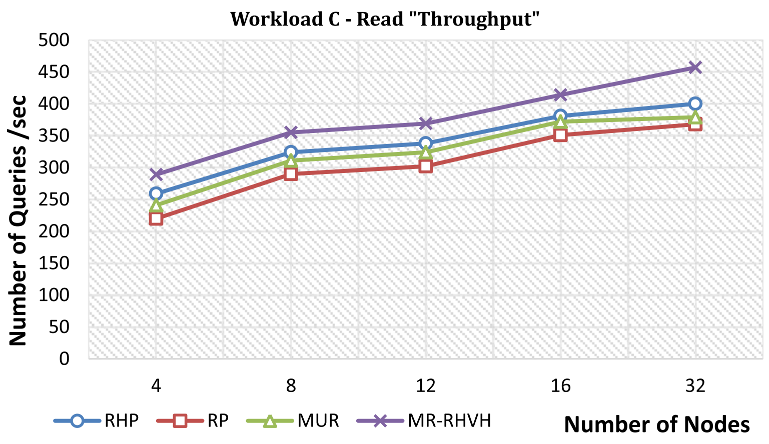

Fig. 9 The comparison of read throughput of workload C of MR-RHVH with recent systems

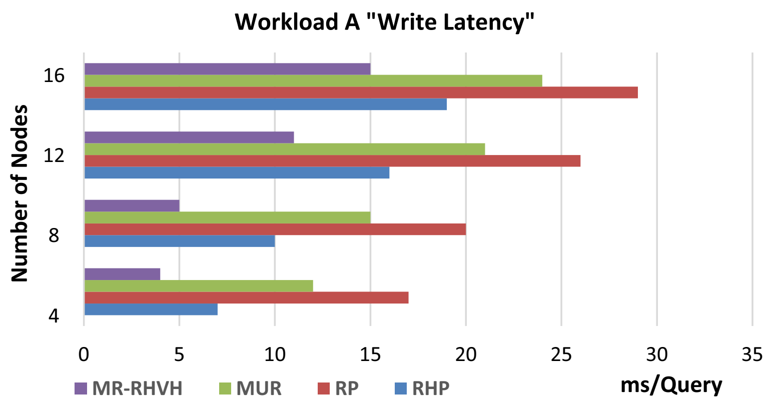

Fig. 10 The comparison of the write latency of workload C of MR-RHVH with recent systems

to recent practitioner's vales, in both experiments. In addition, it is observed that the latencies of write are 10 times less than read latencies, due to the facts that write queries need less tasks than read queries.

4.2.3 Load balancing experiment

The performance of the load balancer sub-module embedded in the framework MR-RHVH is evaluated in this experiment and compared against two partitioners used in original Cassandra [17,20]. Workload C of YCSB is used in load mode to upload the data to one cluster of 8 nodes. Figure 12 shows the performance of Cassandra with ByteOrder partitioner; the load is condensed in one or two nodes which result in uneven distribution of data. Figure 13 demonstrates that using Murmur (the current default Partitioners) in Cassandra the load is almost distributed on the 8 nodes but not uniformly.

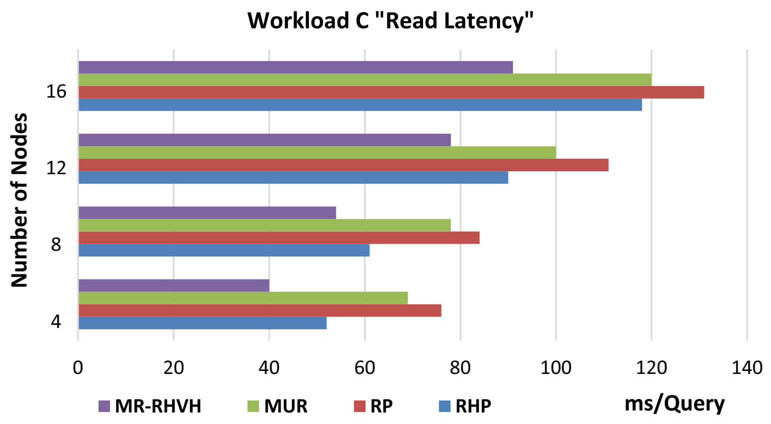

Fig. 11 The comparison of the read latency of workload C of MR-RHVH with recent systems

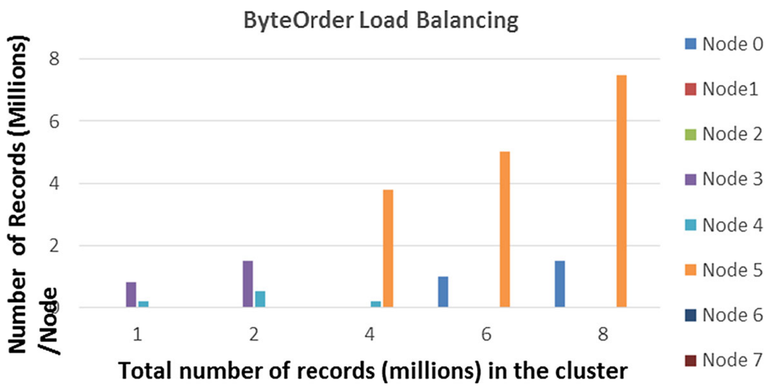

Fig. 12 The performance of Cassandra with ByteOrder partitioner

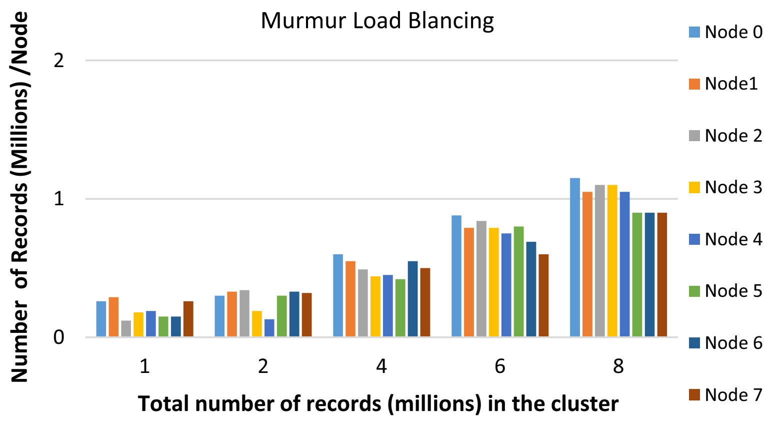

Fig. 13 The performance of Cassandra with Murmur partitioner

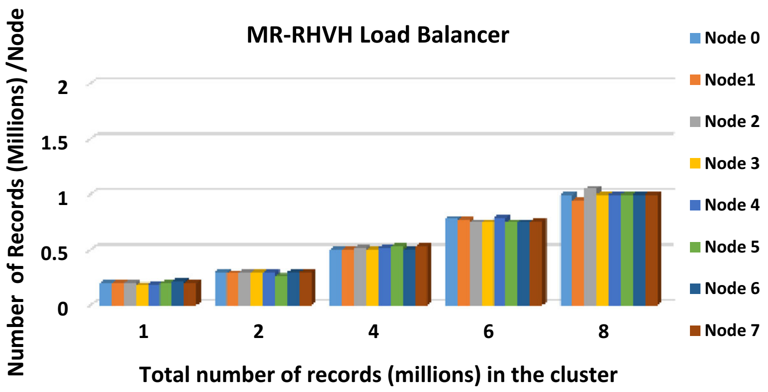

Fig. 14 The performance of Cassandra with MR-RHVH load balancer

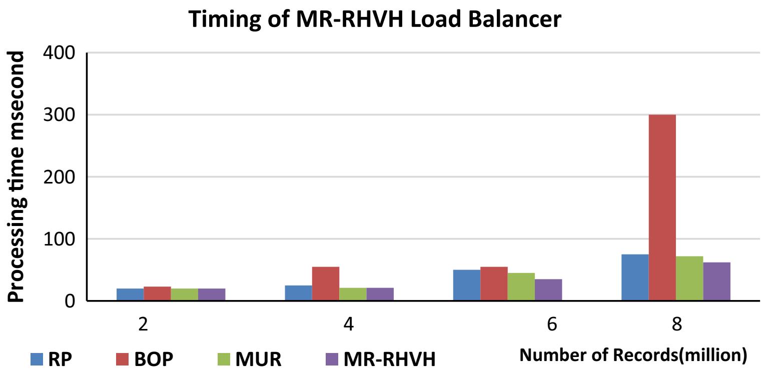

Fig. 15 The comparison of execution time of different partitioners with workload C in balancing load with recent systems

However, Fig. 14 demonstrates a uniformly balanced performance of Cassandra between the 8 nodes when using the MR-RHVH. This comparison validated the Load Balancing-based Rendezvous Hashing algorithm LBRH.

Moreover, the execution time of assigning the records to the nodes in MR-RHVH is compared with standard Cassandra Byte Ordered, Random, and Murmur3 [17,20] partitioning when using READ workload C in YCSB. Figure 15 shows that MR-RHVH is the fastest when varying number of loaded records.

4.2.4 Execution time experiment

The different attempts to develop in partitioning techniques of Cassandra affect its performance, especially when the number of parallel queries increased. MR-RHVH implemented on Cassandra uses the highly parallel MapReduce programming model to support parallel queries in the system. In this experiment, MR-RHVH is tested t [24] o calculate the executing times spent on read and write queries for each of the 4,

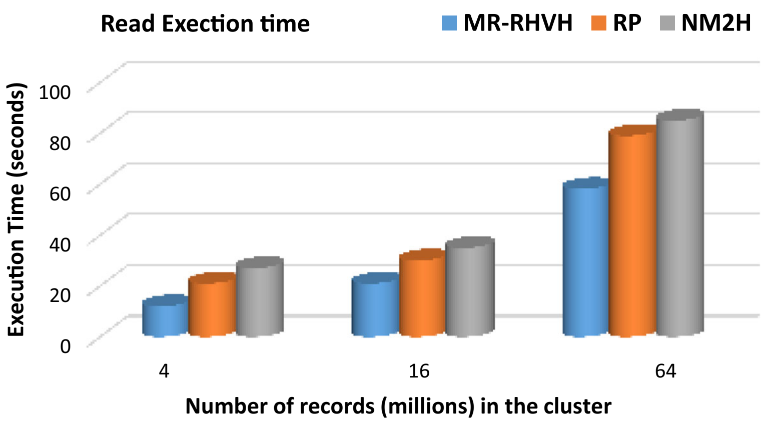

Fig. 16 The comparison of the execution time of MR-RHVH in read workload C with recent systems

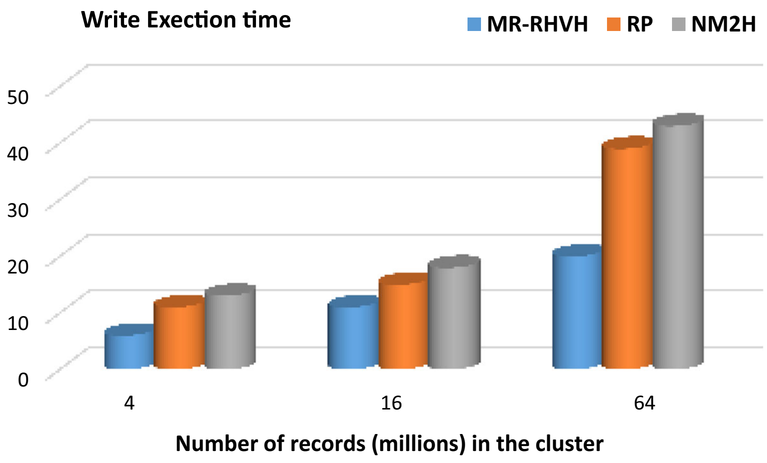

Fig. 17 The comparison of the execution time of MR-RHVH in write workload A with recent systems

16 and 64 million runs. Figures 16 and 17 show the execution time of read and write queries, respectively, of MR-RHVH compared to RP [20] and NM2H [32].

MR-RHVH proved to have less execution time compared to other systems, because each client node in the cluster uses the MapReduce layer to process each data split in its own using the region it is located in. Unlike other partitioners, it needs enormous data movement in Cassandra cluster, which leads to use more execution time. Each node has a local access to the nodes in the same region using the supervisor coordinator node assigned by the proposed AVH structure.

4.2.5 Discussion and result analysis

The comparative results showed that MR-RHVH is scalable to the huge number of nodes, and it is highly efficient in terms of throughput and latency, compared to recent systems. Also, it is more flexible in selecting which nodes can become coordinator node that store and broadcast the keys, based on loads and capabilities of nodes within the region. The MR-RHVH load is balanced more uniformly than the two default partitioners of standard Cassandra. And the MR-RHVH timing for distributing the load is faster than different partitioners of standard Cassandra. In addition, the results proved that MR-RHVH is the fastest in executing read and write queries compared to others systems, because each node in the cluster process its data split in its own, because it has a local access to all nodes in the same region using the supervisor coordinator node assigned by the proposed AVH structure. Unlike other partitioners, it needs enormous data movement in Cassandra cluster, which leads to use more execution time.

5 Conclusions and future work

A MapReduce Rendezvous Hashing-based Virtual Hierarchical (MR-RHVH) framework is proposed in this paper for partitioning Cassandra NoSQL database by using rendezvous hashing. The MR-RHVH framework divides system into Rendezvous Regions, that allows the nodes to be more flexible in selecting the master nodes, while maintaining the locality advantage in dealing with nodes in the same region. In addition, a LBRH algorithm is proposed to guarantee the load balancing with heterogeneous nodes. The timing of hashing is enhanced by using the bloom filter evaluator in each node of the rendezvous hash range.

The results showed that MR-RHVH framework is highly efficient in terms of throughput and latency when compared to the recent systems. In addition, the MR-RHVH load is balanced more uniformly than the two default partitioners of standard Cassandra. And the MR-RHVH timing for distributing the load is faster than different partitioners of standard Cassandra. Our future work intends to reduce the overhead of MR-RHVH and to test its scalability when increasing the number of nodes. In addition, the jump consistent hash will be implemented as a hashing technique in MR-RHVH and replace the Hadoop with the Spark.

References

1. Anagnostopoulos I, Zeadally S, Exposito E (2016) Handling big data: research challenges and future directions. J Supercomput 72(4):1494–1516
2. $10 - K$ Annual Report. SEC Filings. Facebook. 28 Jan 2016. Retrieved 26 Mar 2016
3. Agrawal R, Ailamaki A, Bernstein PA, Brewer EA, Carey MJ, Chaudhuri S et al (2008) The Claremont report on database research. SIGMOD Rec 37(3):9–19
4. Cruz F, Maia F, Matos M, Oliveira R, Paulo Ja, Pereira J, Vilaça R (2013) MeT: Workload aware elasticity for NoSQL. In: Proceeding EuroSys '13 Proceedings of the 8th ACM European Conference on Computer Systems, New York, NY, USA, pp 183–196
5. HBase Development Team (2013) HBase: BigTable-like structured storage for Hadoop HDFS [EB/OL]. http://wiki.apache.org/hadoop/Hbase/. Accessed 20 Mar 2013
6. Chodorow K, Dirolf M (2010) MongoDB: the definitive guide, 1st edn, O'Reilly Media, p 216, ISBN 978-1-4493-8156-1
7. Chang F, Dean J, Ghemawat S, Hsieh WC et al (2008) BigTable: a distributed storage system for structured data. ACM Trans Comput Syst (TOCS) J 26(2):205–218

8. DeCandia G, Hastorun D, Jampani M et al (2007) Dynamo: Amazon's highly available key-value storeC. In: Proceedings of the 21st ACM Symposium on Operating Systems Principles, SOSP 2007, 205–220, Stevenson, Washington, USA, October 14–17
9. Lakshman A, Malik P (2010) Cassandra: a decentralized structured storage system. Oper Syst Rev 44(2):35–40
10. Karger D, Lehman E, Leighton T, Panigrahy R, Levine M, Lewin D (1997) Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world-wide web. In: Proceedings of the 29th Annual ACM Symposium on Theory of Computing, '97, ACM, New York, NY, USA, pp 654–663
11. Chen Z, Yang S, Tan S, Zhang G, Yang H (2013) Hybrid range consistent hash partitioning strategy—a new data partition strategy for NoSQL database. In: 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom), 2013, IEEE, pp 1161–1169
12. Turk A, Selvitopi RO, Ferhatosmanoglu H, Aykanat C (2014) Temporal workload-aware replicated partitioning for social networks. IEEE Trans Knowl Data Eng 26(11):2832–2845
13. Huang X, Wang J, Zhong Y, Song S, Yu PS (2015) Optimizing data partition for scaling out NoSQL cluster. Concur Comput: Pract Exp 27(18):5793–5809
14. Schall D, Härder T (2015) Dynamic physiological partitioning on a shared-nothing database Cluster. In: IEEE 31st International Conference on Data Engineering (ICDE), 2015, IEEE, pp 1095–1106
15. Yao Z, Ravishankar CV, Tripathi S (2001) Hash-based virtual hierarchies for caching in hybrid content-delivery networks. The University of California, Riverside, Department of Computer Science and Engineering, California
16. Cooper BF, Silberstein A, Tam E, Ramakrishnan R, Sears R (2010) Benchmarking cloud serving systems with YCSB. In: Proceedings of the 1st ACM Symposium on Cloud computing, ACM, pp 143–154
17. Abramova V, Bernardino J, Furtado P (2014) Testing cloud benchmark scalability with cassandra. In: IEEE World Congress on Services (SERVICES), 2014, IEEE, pp 434–441
18. Srinivasan L, Varma V (2015) Adaptive load-balancing for consistent hashing in heterogeneous clusters. In: 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), 2015, IEEE, pp 1135–1138
19. Wang X, Loguinov D (2007) Load-balancing performance of consistent hashing: asymptotic analysis of random node join. IEEE/ACM Trans Netw 15(4):892–905
20. Dede E, Sendir B, Kuzlu P, Weachock J, Govindaraju M, Ramakrishnan L (2016) Processing Cassandra datasets with Hadoop-streaming based approaches. IEEE Trans Serv Comput 9(1):46–58
21. Kuhlenkamp J, Klems M, Röss O (2014) Benchmarking scalability and elasticity of distributed database systems. Proc VLDB Endow 7(12):1219–1230
22. Braam PJ et al (2004) The Lustre storage architecture. ftp://ftp.uniduisburg.de/pub/linux/filesys/Lustre/lustre.pdf, 2004
23. Thaler DG, Ravishankar CV (1998) Using name-based mappings to increase hit rates. IEEE/ACM Trans Netw (TON) 6(1):1–14
24. Seada K, Helmy A (2004) Rendezvous regions: a scalable architecture for service location and data-centric storage in large-scale wireless networks. In: Proceedings of the 18th International on Parallel and Distributed Processing Symposium, 2004, IEEE, p 218
25. Kurihara Yuki (2015) Digest::MurmurHash. GitHub.com. Retrieved 18 Mar 2015
26. Jenkins B (2012) SpookyHash: a 128-bit noncryptographic hash
27. Bloom BH (1970) Space/time trade-offs in hash coding with allowable errors. Commun ACM 13(7):422–426
28. Li Zhe, Ross Kenneth A (1995) Perf join: an alternative to two-way semijoin and bloomjoin. In: CIKM '95: Proceedings of the 4th International Conference on Information and Knowledge Management, pp 137–144, 1995
29. Bringer J, Morel C, Rathgeb C (2015) Security analysis of bloom filter-based iris biometric template protection. In: International Conference on Biometrics (ICB), 2015, IEEE, pp 527–534
30. DataStaX (2016) -https://datastax.github.io/python-driver/api/cassandra/policies.html-retrieved. Accessed Jan 4 2016
31. VMware VSpher (2016). Server Virtualization with VMware vSphere | VMware India". www.vmware.com. Retrieved 08 Mar 2016

32. Xue R, Guan Z, Gao S, Ao L (2014) NM2H: Design and implementation of NoSQL extension for HDFS metadata management. In: IEEE 17th International Conference on Computational Science and Engineering (CSE), 2014, IEEE, pp 1282–1289
33. Gudivada VN, Rao D, Raghavan VV (2014) NoSQL systems for big data management. In: IEEE World Congress on Services (SERVICES), 2014, IEEE, pp 190–197