Optimizing vBucket Mapping Rebalance in Distributed Database Systems

(Draft 2)

Weikang Zhou

September 11, 2012

Abstract

Data mapping is a critical problem in distributed systems. A good data mapping algorithm should be scalable and minimize the amount of data migration while maintaining load balance under a dynamic environment.

This report proposes a topology-replication-mapping scheme based rebalance algorithm to optimize replication mapping in distributed database systems. The algorithm has consistent and scalable performance, and the amount of data migration during rebalance is very close to the theoretical lower bound. The algorithm further extends to accommodate various new features.

1 Introduction

In the age of Big Data, "NoSQL" (non-relational) database systems, offering scalability and flexibility, rise in popularity as an alternative to the traditional relational database management systems (RDBMS). NoSQL technology excels at managing large volumes of data that do not necessarily have a fixed schema.

NoSQL database systems support very fast create, store, update and retrieval operations when dealing with huge quantities of data that the traditional RDBMS approach could not cope with. The significant gains in scalability and performance make NoSQL database systems useful in a wide range of applications where performance is more important than consistency, such as indexing a large number of documents, serving pages on high-traffic websites, and real-time statistical analysis for a growing list of elements.

NoSQL database systems usually employ a distributed and elastic architecture in a server cluster, with the data held in a redundant manner on several servers. In this way, the system can easily scale out by adding more servers, and failure of servers can be tolerated. This architecture is controlled by maintaining a mapping from data partitions (called vBuckets, short for virtual buckets) to servers. vBuckets can be obtained by partitioning the data key space into logical storage units, and they are distributed across servers in the cluster via the vBucket mapping. Fault tolerance and redundancy is provided by data replication at the vBucket level over multiple servers. For a given vBucket, there is a node hosting this vBucket that serves client requests (the active node); the same vBucket is also being replicated on some other nodes (the slave nodes).

When failure of a server occurs, the standby replica hosted in some other server will take over and become active. As we would like the servers to achieve load balance to maximize resource utilization, it is desirable that the vBucket mapping should be evenly distributed among member nodes in the cluster for both active and replica vBuckets (balanced in the first order). The workload of data migration during server failure should also be evenly distributed among existing servers, so that there is no single bottleneck in the cluster. This requires the vBucket mapping to be balanced in the second order: to spread the replicas as evenly as possible among other member nodes.

We summarize these balance constraints of the vBucket mapping as the following:

C1 *First order balance* The active vBuckets are evenly distributed among all the member nodes in our cluster, and the replica vBuckets are also evenly distributed among all the member nodes.

C2 *Slave number balance* Every node has the same number of slave nodes. That is, every node replicates to the same number of other member nodes in the cluster.

C3 *Second order balance* For any particular node, the vBuckets that are active on this node should have their replicas evenly distributed among its slave nodes.

The core of managing the configuration of the cluster is a rebalance (also called auto-sharding) algorithm that initiates and supervises the movement of data between individual servers during a cluster resizing operation. During rebalance, the vBucket mapping is recreated such that the balance constraints are maintained and data migration across servers is minimal. The rebalance algorithm should decide optimally which vBuckets be migrated and which remain in its place. The objective is to optimize resource utilization while maintaining load balance of the new vBucket mapping, so that we have a robust database and prevent any disruption to its live applications.

In other areas of distributed computing, where one might also seek to separate load between multiple servers (either for performance or reliability reasons) and would like to maintain the load balance in a dynamic environment, such a rebalance algorithm is also useful, and our proposed algorithm could also be applied accordingly.

The rest of this report is organized in the following. In Section 2 we formally introduce the rebalance problem. In Section 3 a detailed description of the algorithm is presented, followed by simulated performance results in Section 4. Several extensions of the algorithm is discussed in Section 5. Finally, we present our conclusion and discuss future works in Section 6.

2 Problem Formulation

Suppose a database cluster with M servers $\mathbb{Z}_M := \{1, \dots, M\}$ is hosting our data, which is partitioned into N vBuckets. We assume that these vBuckets are of equal size (as a consequence of the hashing function used, for example). On the M nodes in our cluster, we maintain L copies of each vBucket. One of these L copies is active, and the rest are replicas.

Let S be the number of slave nodes for every node in our cluster. If $S = M - 1$, then every node replicates to every other node. We assume that $1 \leq L \leq M$ and $(L-1) \leq S$. In a typical setting, $N = 1024, M = 50, L = 4, S = 10$.

A vBucket mapping A is a $N \times L$ matrix of elements in $\mathbb{Z}_M$. Row i of matrix A has L elements, representing the nodes that host the L copies of vBucket i with the leading entry being the active node. Each row of A should have distinct values as the active copy and each replica are hosted in separate nodes. We consider only star replication, which is to say that all replicas are equivalent with each other.

A replication pair is a 2-tuple in A: $\big(A(i,1), A(i,j)\big), j = 2, \dots, L$, representing that vBucket i is active on node $A(i,1)$ and being replicated on node $A(i,j)$.

The vBucket mapping A defines the replication matrix R of size $M \times M$: $R(k,l)$ is the total number of occurrences of the replication pair (k,l) in the vBucket mapping A. For example, if $R(k,l) = 5$, there are 5 vBuckets such that they are active on node k and has replicas on node

l.

The Boolean matrix $RI := (R > 0)$ characterizes the replication relationships of nodes. $RI(k, l)$ is 1 if and only if node k replicates to node l for some vBuckets. Matrix RI is essentially the adjacency matrix for the replication topology in our cluster. We call RI the topology matrix.

Recall that we would like the vBucket mapping to have load balance to maximize resource utilization, and be balanced in the replication relationships so that there is no single bottleneck in the cluster. These balance constraints given in Section 1 can now be formulated equivalently using the above notations as the following:

C1 *First order balance* The vBucket mapping A should be having equal number of elements of $\mathbb{Z}_M$ in its first column (for the active vBuckets) and the rest columns (for the replicas).

C2 *Slave number balance* Each row of R should have exactly S non-zero entries.

C3 *Second order balance* The S non-zero entries of each row of R should be equal.

From now on a “balanced” mapping is a mapping that satisfies all three constraints above. The four parameters of a mapping (N, L, M, S) is provided by the user.

The rebalance algorithm attempts to rebalance the vBucket mapping with minimal data movements during cluster resize. (A movement is adding a new element to a row of A, which requires data migration of that vBucket). During rebalance, we will allow all the parameters of the mapping to be changed with the exception of N. The objective of the algorithm is to generate a new balanced mapping such that vBucket movements are minimized.

3 The rebalance algorithm

3.1 The Structure of R

We begin by analyzing the structure of R of a balanced A, because, as we will show, R characterizes A completely as far as all the balance constraints are concerned.

If the element $R(i, j) = n$, there are n rows in A where node i is in the active position and node j has a replica. Thus the column sum of R is the number of times the corresponding node has a replica in A.

The sum of all entries in R is $N \cdot (L - 1)$. As we want its column sums to be equal, if $N \cdot (L-1) = q * M + r$, there should be r of $(q+1)$'s and $(M - r)$ of q's in the column sums of

R. In other words, the column sums are R is either $\lfloor \frac{N \cdot (L-1)}{M} \rfloor$ or $\lfloor \frac{N \cdot (L-1)}{M} \rfloor + 1$ (only if $r > 0$ when N cannot be divided by M. The same qualification is true but omitted later for clarity.)

Each active vBucket has $(L-1)$ replicas, thus the row sums of R divided by $(L-1)$ is the number of times the corresponding node appears in the first column of A. We want them to be equal as well, which is given by $\lfloor N/M \rfloor$ or $\lfloor N/M \rfloor + 1$.

Therefore, for each row, R has S nonzero equal terms that sums to $\lfloor N/M \rfloor \cdot (L-1)$ or $(\lfloor N/M \rfloor + 1) \cdot (L-1)$. So each entry should be $\left\lfloor \lfloor \frac{N}{M} \rfloor \cdot \frac{L-1}{S} \right\rfloor$, or $\left\lfloor \lfloor \frac{N}{M} \rfloor \cdot \frac{L-1}{S} \right\rfloor + 1$, or $\left\lfloor (\lfloor \frac{N}{M} \rfloor + 1) \cdot \frac{L-1}{S} \right\rfloor$, or $\left\lfloor (\lfloor \frac{N}{M} \rfloor + 1) \cdot \frac{L-1}{S} \right\rfloor + 1$. This expands three consecutive integers at most, two most of the time. We can easily derive the exactly the number of occurrences for each of them for each row of R.

Conversely, if R satisfies:

R1 R has row sum $\lfloor N/M \rfloor \cdot (L-1)$ or $(\lfloor N/M \rfloor + 1) \cdot (L-1)$;

R2 R has column sum $\lfloor \frac{N \cdot (L-1)}{M} \rfloor$ or $\lfloor \frac{N \cdot (L-1)}{M} \rfloor + 1$;

R3 Each row of R has exactly S non-zeros entries, which are evenly distributed for the corresponding row sum of that row.

Then A satisfies **C1-C3**.

It can be shown that a feasible A exists always for R that satisfies **R1-R3**.

3.2 The topology-replication-mapping scheme

As we have showed, the topology matrix RI characterizes the replication relationship of our cluster, and R characterizes the balance constrains and A represents the actual vBucket mapping we want. In order to maintain balance and minimize data movements, we propose a fundamental framework of our algorithm in this subsection.

Observe that the topology RI is derived from R and can in turn determine R, in the same way R is derived from A and can in turn determine A. The three objects are connected while each has its own domain of functionality, therefore we want to separate them as each specializes in optimizing different aspects of our rebalance process, as shown in Figure 1.

The specialization of each block also extends beyond the basic functions of the rebalance algorithm. For example, if we prefer replications to occur between servers not sharing the same

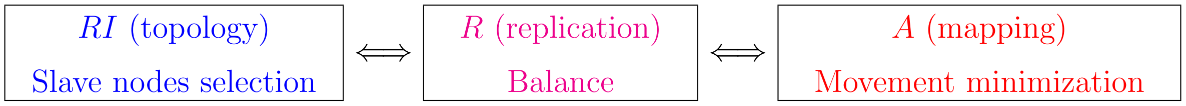

Figure 1: The topology-replication-mapping scheme

power source, we could attach a tag to each node and optimize the replication relationship with RI. For another example, if we have heterogeneous nodes that requires a modified balance constraints, we can implement this with R. Section 5 will explore these possible extensions of the rebalance algorithm based on this flexible and accommodating topology-replication-mapping scheme.

Based on the scheme, the rebalance algorithm has the framework shown in **Algorithm 1**.

Algorithm 1 Rebalance algorithm framework

```
Input: Current vBucket mapping $A_0$
Output: Rebalanced vBucket mapping $A$
1: Process input mapping $A_0$; compute its $R$ and $RI$
2: Topology: find optimal target $RI^T$
3: Balance: target $RI^T \longrightarrow$ target $R^T$
4: Minimize movements: target $R^T \longrightarrow$ output new mapping $A$
```

In the first step we process the original vBucket mapping A_0 with some simple operations, such as properly marking the deleted nodes, expanding the node space $\mathbb{Z}_M$ if new nodes are added to the cluster, etc. Note that there is no constraint on the input mapping A_0. It need not be balanced, for instance.

3.3 Optimal topology RI

Finding the optimal topology configuration RI is the most crucial and computationally expensive step of our rebalance algorithm. The space of feasible RI matrices is huge: any Boolean matrix of size $M \times M$ with S non-zeros in each of its row and column is a potential candidate. We denote this space of feasible topologies $\mathcal{RI}$. We propose to use simulated annealing to search for the optimal RI.

First we need to define the energy of any given feasible topology matrix RI. The energy should approximate the distance between the resulting target R^T and the current R_0. Generally, the further they are apart, the more data movements it takes to change current mapping A.

We define the energy function as the following:

$$\mathbb{E}(RI) := \sum_{i,j=1}^{M} \left| R_0(i,j) - \widetilde{R}(i,j) \right|, \quad \widetilde{R} := \frac{N \cdot L}{M \cdot S} RI, \quad RI \in \mathcal{RI} \tag{1}$$

Observe that we use $\widetilde{R}^T$ to approximate the resulting target R from our candidate RI, because it is accurate enough and very fast to evaluate without needing to actually balance R.

Now we can define a Gibbs distribution on the space of feasible topology matrices:

$$\pi_T(RI) \propto \exp\left(-\frac{\mathbb{E}(RI)}{T}\right), \quad RI \in \mathcal{RI} \tag{2}$$

where T is the "temperature" of the distribution.

Next we can define a Metropolis algorithm in the probability space $\mathcal{RI}$ with the above Gibbs distribution, as shown in **Algorithm 2**. (In practice, we can safely assume that $\dfrac{\mathbb{P}(RI_2 \to RI)}{\mathbb{P}(RI \to RI_2)} \approx 1$ for fast computation of acceptance rate.)

Algorithm 2 The Metropolis algorithm on feasible topology matrices

```
1: Current state $RI \in \mathcal{RI}$
2: Uniformly at random select two rows and columns of $RI$, such that the four entries are two
   1's on one diagonal and two 0's on the other diagonal
3: Make our proposal, which is to switch the two diagonals to get $RI_2$
4: Accept the proposal $RI \leftarrow RI_2$ with probability $\min\left\{1, \frac{\mathbb{P}(RI_2 \to RI)\, \pi_T(RI_2)}{\mathbb{P}(RI \to RI_2)\, \pi_T(RI_1)}\right\}$
```

The simulated annealing algorithm is the Metropolis algorithm with a cooling schedule. We start at a very hot temperature (burn-in stage) and gradually cool down so that we can sample the topologies with the lowest energy. We can run this procedure multiple times, or use other techniques like parallel tempering. The essential point is that the energy function accurately reflect the optimality of the topology matrix, and the mixing of this Markov chain is sufficient fast.

A potential problem with the energy function is that it does not reflect the actual number of movements to balance the resulting R^T, because attempting to do the rebalance would take too long to evaluate the optimality of a particular topology matrix. Consider the following difference matrices $R_0 - \widetilde{R}$ with the same energy $4n$: $\begin{bmatrix} & n & \\ n & -n & \\ & & -n \end{bmatrix}$ which requires $3n$ moves to balance, and $\begin{bmatrix} & n & \\ n & & -n \\ & -n & \end{bmatrix}$ which requires only $2n$ moves to balance.

We observe that the energy function is indifferent between the two (which differ by a switch of the lower-right corner), while obviously one takes fewer data movements to balance. This situation arises when we simultaneously add and delete nodes in the cluster. We can circumvent this problem by first matching the deleted and added nodes in the original mapping A. This can be done when we first process the input mapping A in **Algorithm 1**.

3.4 Balance target R

Once we have the optimal RI, the next step is to construct a balanced replication matrix R that corresponds to the optimal RI. Since RI gives us the positions of non-zero entries in the target R^T, this step involves adjusting the entries we place in these positions so that R satisfies the balance constraints **C1-2**.

First we determine the row sums for each row of R^T according to **R1**. We can do this by assigning the closest possible number to the corresponding row sum of the original R_0. Then we determine the value for each non-zero position according to **R3**. This can be similarly done by assigning the closest possible value to the corresponding position of R_0.

Now R^T is balanced except the equal column sums constraint **R2**. We then greedily exchange two non-zero entries on the same row to balance the column sums. If we cannot find any such adjustments on the same row and R^T is still not balanced, we go back and exchange two row sums of R^T greedily to help balancing the column sums. By repeating this procedure, we can maintain the two constraints **R1, R3** while adjusting R^T to satisfy the constraint **R2**. The algorithm is shown in **Algorithm 3**.

Algorithm 3 Balance target replication matrix

```
1: Assign initial values to non-zero positions in $R^T$ such that it satisfies **R1, R3**
2: **for** $i = 0$ to $n$ **do**
3:     **while** $R^T$ does not satisfy **R2 do**
4:         Greedily search to exchange two entries in the same row
5:         If $R^T$ still not balanced, greedily exchange two row sums of $R^T$, **go to** 2
6:     **end while**
7: **end for**
```

Note that we defined n to indicate the maximum number of trials that are allowed before we abort the attempt to balance R^T. This is because it is not always possible to construct a balanced R given its topology RI. This impossibility arises only when RI is disconnected (as a block matrix) and $(L-1)$ cannot be divided by the sum of column sums in a connected partition. Then there does not exist a balanced R^T because row sums are always a multiple of

$(L-1)$. In practice, this happens very rarely (less than one in a thousand), and when it does happen, the imbalance introduced is tiny (at most four column sums off by 1) and thus can be ignored.

3.5 From target R to new mapping A

As we have our target R^T balanced, our goal at this step is to adjust A so that its R will conform to the target R^T. We use two stages to achieve this: adjusting active nodes and adjusting replicas.

First stage, we make A conform to the row sums of R^T by adjusting active nodes. We move greedily those vBuckets on nodes having excessive number of active vBuckets to nodes not having enough active vBuckets. The procedure is greedy in the sense that we choose vBuckets such that they have the greatest number of replicas not "wanted" by its previous active node but "wanted" by its new active node. In the second stage, we fix the active nodes A has, and greedily adjust their replicas such that A will conform to R^T in each row. This two-stage procedure is shown in **Algorithm 4**.

Algorithm 4 From target replication matrix to mapping

```
 1: for Node $i = 1$ to $M$ do                                  ▷ Stage 1: adjusting active nodes
 2:     while Node $i$ has too many active vBuckets do
 3:         Find an active vBuckets $j$ having the greatest number of replica nodes by node $i$
 4:         Find a node $k$ having too few active vBuckets and want the most replicas of $j$
 5:         Change the active node of vBucket $j$ from $i$ to $k$
 6:     end while
 7: end for
 8: for Node $i = 1$ to $M$ do                                  ▷ Stage 2: adjusting replicas
 9:     while $A$ does not conform to row $i$ of $R^T$ do
10:         Find an active vBucket replicated on node $k$ and we have too many $(i, k)$ replication
11:         Find another node $l$ to substitute $k$ such that we have too few $(i, l)$ replication
12:         If no such change is possible, change through a second vBucket
13:     end while
14: end for
```

Deciding active nodes in A is much more critical than replica nodes, as the latter operation only changes R by 1 in two entries and can be adjusted with some degree of freedom, since usually N is much larger than M. But when we change an active node, we alter R by 1 in $2 \cdot (L-1)$ entries.

In our algorithm, we only change active nodes in a greedy way and go to second stage. That is, we first minimize the number of active vBuckets to be moved, and then minimize the number of replicas to be moved. An alternative is to make additional active vBucket movements such that at stage two fewer movements are needed. This alternative approach might be beneficial if there is meaningful trade-off between the two minimizations. In practice, the redundancy level L is small (in the single digit) and we do not find such a trade-off. When we have large L, however, the algorithm can be changed to bridge the two stages to minimize both movements globally.

3.6 Theoretical lower bound

Let us consider the theoretical lower bound on the number of movements to rebalance. In a simple case where the input mapping is balanced and we are either adding or deleting nodes, suppose we change the number of nodes in the cluster from M_0 to M, then the lower bound is given by:

$$\textbf{Lower bound} = \left\lfloor \frac{N \cdot L}{\max\{M, M_0\}} \right\rfloor \cdot |M - M_0| \tag{3}$$

This is because both the output and input mapping are balanced, the number of occurrences for each node that needs to be changes is at least $\left\lfloor \frac{N \cdot L}{\max\{M, M_0\}} \right\rfloor$.

Generally, for any input mapping A_0, the lower bound can be determined by considering the imbalance of its replication matrix R_0. We observe that the change of one active vBucket modifies two column sums of R by one for each of them. Denote $\mathbf{c}_0$ the vector of column sums of the input R_0, and let $\mathbf{c}_T$ be the vector of column sums of the target R^T. Since if we change one active vBucket in A, one element in $\mathbf{c}_0$ will increase by 1 and an other will decrease by 1, at least one has to change $\frac{1}{2}||\mathbf{c}_0 - \mathbf{c}_T||_1$ active vBuckets to balance $\mathbf{c}_0$.

Similarly, the change of one replica modifies two row sums of R_0 by $(L-1)$ for each of the two rows. Let $\mathbf{r}_0, \mathbf{r}_T$ be the row sums of input R_0 and target R^T, respectively. Then, in order to balance the row sums of R, we need to make at least $\frac{1}{2(L-1)}||\mathbf{r}_0 - \mathbf{r}_T||_1$ moves of replicas. Even though R need not be balanced, we know that R^T is balanced. That is, we know the values of its column sums and row sums.

In conclusion, we have shown that, for a general input, the lower bound of number of movements is:

$$\textbf{Lower bound} = \min \Bigg\{ \frac{1}{2(L-1)} \sum_{i=1}^{M} \Bigg| \sum_{j=1}^{M} R_0(i,j) - \sum_{j=1}^{M} R^T(i,j) \Bigg| + \frac{1}{2} \sum_{i=1}^{M} \Bigg| \sum_{j=1}^{M} R_0(j,i) - \sum_{j=1}^{M} R^T(j,i) \Bigg| \Bigg\} \qquad (4)$$

The minimum is taken over all possible permutations of balanced column sums and row sums of R^T.

Note that, however, the lower bound is by no means tight. For example, let $L = 2$ and $R_0 - R^T = \begin{bmatrix} & n & \\ n & -n & \\ & & -n \end{bmatrix}$. We can see that the original R_0 requires $3n$ moves to conform to R^T. But the general lower bound given by Eq. (4) is $2n$.

4 Simulated Performance Results

For balanced input mappings, Figure 2 shows the performance of our rebalance algorithm compared to the theoretical lower bound. The parameters are $N = 1024, L = 4, S = 10$. Solid lines show the actual cost (number of vBucket movement counts) of rebalance, where zero cost point on the x-axis corresponds to the original number of nodes in the cluster. Dashed lines are the corresponding balanced input lower bounds given by Eq. (3). We observe that, especially for small to medium size of clusters, the actual number of moves it takes to rebalance is very close to the theoretical lower bound. The gap between the two grows larger as the cluster size grows and the percentage of change of number of nodes is getting smaller. (We believe, however, that most of the gap is to be explained by the inadequacy of the theoretical lower bound rather than the inefficiency of the algorithm.)

When $N = 1024$, for all the rebalances between any clusters under the size of 100 nodes, on avarage the rebalance algorithm performs with an optimization gap of 5.2% as compared to the theoretical lower bound when each vBucket has two copies. For the combined average of having two, three, or four copies of each vBucket in the cluster, which are the usual configurations in practice, the rebalance algorithm has an optimization gap of 7.4%.

For imbalanced inputs, Figure 3 shows the simulated performance results of our rebalance algorithm. The parameters are $N = 1024, L = 4, S = 10$. And initially we have a balanced cluster with 30 nodes. At stage 1, the cluster adds 6 new nodes and deletes 4 previous nodes. The rebalance operation at stage 1, however, is incomplete, therefore the resulting mapping is imbalanced. We randomly select vBuckets that either could have migrated to the new mapping

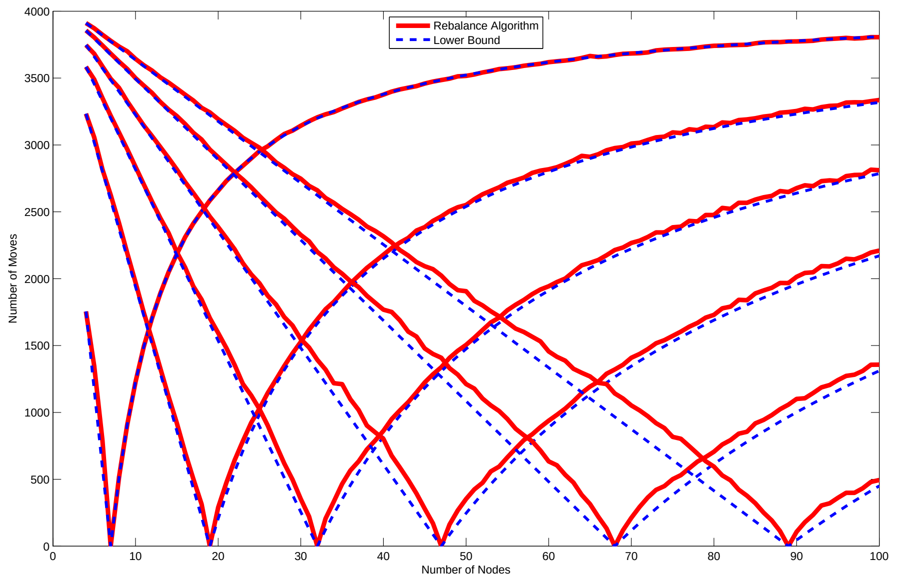

Figure 2: Performance for balanced input

after rebalance, or remain on their previous nodes. The percentage of vBuckets successfully migrated to the new mapping is the progress of completion of our stage 1 rebalance operation.

Then in stage 2, the cluster adds another 4 nodes, and deletes 4 nodes. Two of the 4 nodes to be deleted are those nodes supposed to be deleted at stage 1, but remains in the cluster because stage 1 has not completed. Therefore we retain the two other nodes deleted in stage 1. And stage 2 further delete 2 other nodes.

Figure 3 shows the cost of each operation versus the percentage of completion at stage 1. We observe that the rebalance algorithm is able to consistently maintain balance and minimize data migration. We also observe that the number of movements changes approximately linearly with the rate of completion of stage 1. And the total number of movements is minimal when the cluster resizing operation is done in one rebalance step.

5 Extensions of the rebalance algorithm

The topology-replication-mapping scheme is general and flexible that the rebalance algorithm can be modified slightly and extend to accommodate a variety of additional constraints that might arise in practice.

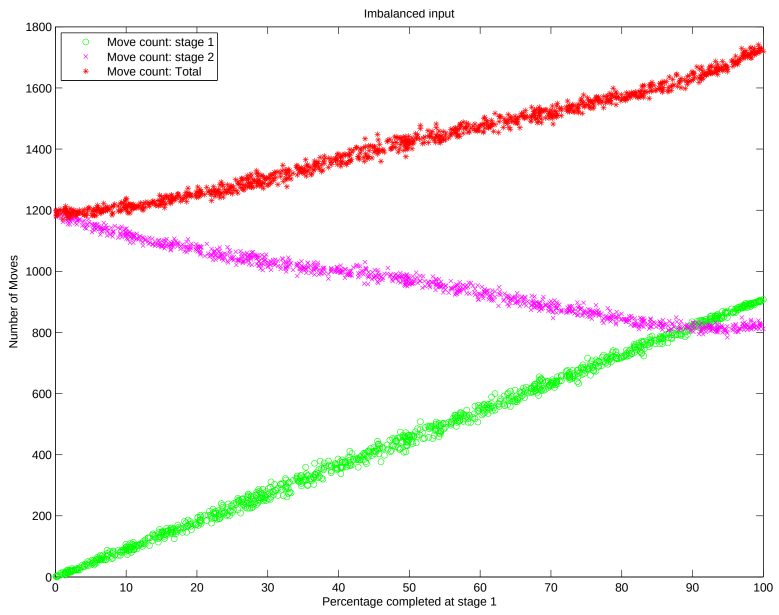

Figure 3: Performance for imbalanced input

5.1 Node tags optimization

Node tags are properties that are attached to member nodes in a cluster. It can describe the power sources of servers, the shelves a server is placed on, etc. And the additional constraint in this case is that we prefer that replications occur over nodes with different tags, so that simultaneous server failure is less likely than, say, when data is being replicated on the servers sharing a common power source. We can define multiple node tags to a server and assign different priorities of preference to each tag.

To accommodate the node tags constraint, we modify the optimal topology step. Specifically, we take the node tags optimization into consideration when defining the energy function in Eq. (1) of our Gibbs distribution. The new energy function function would be the following:

$$\mathbb{E}_{\text{tags}}(RI) := \sum_{i,j=1}^{M} \left| R_0(i,j) - \widetilde{R}(i,j) \right| + \sum_{k=1}^{t} \sum_{i,j=1}^{M} RI(i,j) \cdot \mathcal{T}_k(i,j) \tag{5}$$

where

$$\mathcal{T}_k(i,j) = \begin{cases} p_k & \text{If node } i \text{ and node } j \text{ share the same tag } k \\ 0 & \text{otherwise} \end{cases}$$

In other words, the energy function has two parts: the original energy (which minimizes an approximation of the data movements needed to rebalance) and the energy associated with a total of t tags attached to each node. When a replication over the same node tag occurs for tag k, the energy increases by the tag price p_k, which is a parameter indicating the priority of this particular tag and can be fine-tuned by the user to reflect the practical importance associated with the node tag. When p_k is relatively small (for instance, $p_k = 2$), there is trade-off between the node tags optimization and data movements minimization; when p_k is relatively large (for instance, $p_k = 15$), the rebalance algorithm will attempt to satisfy this node tags constraint regardless of the number of additional data movements.

Price		Violation count		Total number
tag 1	tag 2	tag 1	tag 2	of moves
2	3	92	88	488
5	5	52	57	643
0	0	109	103	479
3	10	87	0	752
7	10	16	0	939
10	10	0	0	1009
10	7	0	14	957
10	3	0	79	767

Table 1: Tag price and performance

Table 1 shows the performance of this extension for an instance with $N = 1024, L = 4$ and two tags attached to each node. Instances of a node with one tag replicates to another node with the same tag is being recorded in the violation count. We observe that the tag price parameter indeed controls and adjusts the trade-off between violations of each tag and the total number of movements it takes to rebalance.

5.2 Heterogeneous nodes

The rebalance algorithm can be modified to handle heterogeneous nodes as well. For now we assume that all the nodes are homogeneous, and such assumptions might not be true in certain applications. In case that we have heterogeneous nodes, we can divide each node into homogeneous virtual nodes, and servers with larger capacity (disk space, memory, etc.) will contain more virtual nodes than other servers. And we can deal with rebalance operation in the space of virtual nodes. In this extension, we will have to keep track of the physical nodes in the algorithm as well, as replication can happen on the virtual nodes level, but we need to make sure that vBuckets replicate to physically different nodes. This can be done at the optimal topology step in our topology-replication-mapping scheme.

5.3 Chain replication

The rebalance algorithm also extends to handle chain replication structured clusters. Instead of star replication, a cluster may have a chain replication, which introduces linear sequence and priority among the replicas. Chain replication requires that each level of replication should be balanced as in the sense of $L = 2$. And we can do rebalance in each level as well. At each level, we will call the original rebalance with $L = 2$ to maintain balance. A crucial difference is that now we need to separately make sure that a vBucket does not reside in the same node twice.

5.4 Choosing the slave number S

The slave number S determines for each node, how many other nodes are going to host replicas for vBuckets that are active on this node. This parameter is determined by the user.

When multiple nodes fail at the same time, the vBuckets that have all of their copies on these nodes will be lost. We would like to investigate the percentage of data loss in this case. We claim that, as a general property of vBucket mappings, for any mapping with the same M and L, the expected percentage of vBuckets lost is constant, given that the number of the nodes that failed is a constant and that node failures are independent. That is, *on average* we will always have the same level of data loss regardless of the configuration of replication relationships, as long as we have the same number of nodes and the same number of copies for each vBucket.

Proof. Suppose the number of nodes failed is f. There are $\binom{M}{f}$ such equally likely combinations of node failure scenarios. It suffices to show that the sum of vBucket loss under each of these scenarios is a constant depending only on M and L. If $L > f$, no vBucket will be lost. Otherwise for any vBucket, its L copies will be lost in exactly $\binom{M-L}{f-L}$ scenarios. Therefore the expected percentage of vBuckets lost is given by $\binom{M-L}{f-L} \Big/ \binom{M}{f}$. □

Given that on average the data loss is independent of actual replication configurations of the vBucket mapping, we will face a trade-off between the probability of occurrences of data loss and the amount of data lost. We might spread the risk and encounter either a large number of occurrences of data loss, but each time we only lose a small percentage of data; or we might concentrate the risk and only anticipate a large portion of data lost in a very small number of scenarios.

The significance of the slave number S is that this trade-off can be effectively adjusted via the value of S. For simplicity, assume that each vBucket has 2 copies, one active and one replica, and the cluster is balanced. Assume two nodes fail, and we may have data loss. If S is large, for example $S = M - 1$ (every node is replicating to every other node), then any combination of a two-node failure will result in a loss of vBuckets that are hosted on those failed nodes. But the percentage of vBucket lost in each case is small, namely $1 \Big/ \binom{M}{2}$ in this case. On the other hand, however, if S is small, for example $S = 1$ (each node is replicating only to its following node), then the probability that the failed nodes are adjacent to each other is much smaller. Now the chance of data loss is only 2 in $(M - 1)$, but when it does happen, all the vBuckets active on the first node and its slave node are lost, which is now a much larger percentage, namely $\frac{1}{M}$ of all the vBuckets. (Meanwhile the expected number of data loss is the same.)

If the servers are robust and the probability of failure is tiny, we may elect to employ a small S; if instead the servers are prone to failure but we prefer strongly a small data loss with high probability to a large data loss with small probability, we may elect to employ a large S to spread out our data replication. In practice, the preference is usually in between the two.

Note that the distribution of data loss is not controlled entirely by the slave number S. Different mappings with the same S can have different data loss statistics. If the goal is to maximally distribute risk, ideally the balancedness criterion would be that the $(L - 1)$ replica copies of each vBucket are distributed evenly among the permissible space of all $\binom{M-1}{L-1}$ such combinations. But this is difficult to achieve because the size of this space could be much larger than the number of vBuckets available, and because that this constraint imposes a heavy burden on the number of data movements required for rebalance.

6 Conclusion

Our rebalance algorithm is a highly extensible, efficient, and scalable algorithm that optimizes data movements during resizing operations in distributed database systems. The algorithm maintains the balance of replication mapping and minimizes the amount of data migration needed. The performance of the algorithm is consistent and reliable, and the algorithm can be modified to extend to include a variety of new features.

For theoretical future work, there are four interesting problems to explore: refine the lower bound on the number of movements for rebalance so that it is tight; find the exact complexity of the algorithm by consider the mixing time of the Markov chain in Algorithm 2; prove an upper bound on the optimization gap ratio; and a further statistical analysis on node failure and data loss.