Distributed Balanced Tables: Not Making a Hash of it All

Prasanna Ganesan Mayank Bawa
Stanford University
{prasanna, bawa}@db.stanford.edu

Abstract

DHTs implement a distributed dictionary, supporting key insertion, deletion and lookup. They use hashing to (a) enable efficient dictionary operations, and (b) achieve storage balance across the participant nodes. Hashing can be inappropriate for both problems, as it (a) destroys data ordering, thus making sequential key access and range queries expensive, and (b) fails to provide storage balance when keys are not unique. We propose generalizing DHTs to create Distributed Balanced Tables (DBTs), which eliminate the above two problems. To solve problem (a), we discuss how DHT routing structures can be adapted for use in DBTs, while preserving the costs of the standard dictionary operations and supporting efficient range queries. To solve problem (b), we describe an efficient algorithm that guarantees storage balance, even against an adversarial insertion and deletion of keys.

1 Introduction

Distributed Hash Tables (DHTs) [11, 12, 14, 15] provide a distributed implementation of a Dictionary ADT allowing efficient inserts, deletes and lookups of keyed data. Nodes and keys are both hashed into a circular ID space. The IDs of nodes determine both the partitioning of the hash space among the nodes, as well as the routing interconnections between nodes. We argue that DHTs suffer from the following two problems.

1 **Hashing destroys data ordering:** Hash tables are effective for enabling fast queries for individual keys. However, many applications desire to exploit the ordering of keys and pose queries over key ranges. For example, a node querying an auction system might desire a list of all computers for sale, priced between $400 and $600. As another example, in a distributed cache where keys are page URLs, an application may desire to look up all pages with a specific URL prefix. It is well-known [4] that hash partitioning is inefficient for answering queries over *ad hoc* ranges of data.

2 **Hashing does not provide storage balance:** A lot of recent work [1, 9, 11] has shown that it is possible to ensure that the hash-space partitions managed by DHT nodes remain well balanced. However, such hash-space balance is insufficient when the keys being hashed are not unique. For example, when using a DHT to build an inverted index over documents, terms are used as keys, and the number of data items with a particular term follows a Zipfian distribution. Thus, there can be a severe storage imbalance even if the hash-space partitions are balanced.

In this paper, we present initial results from designing *Distributed Balanced Tables* (DBTs), which solve the above problems by the use of range partitioning. The data domain is partitioned into multiple contiguous ranges, with each node managing one of the ranges. Nodes are then interconnected using an overlay network to allow efficient lookup of a specific key, as well as a sequential retrieval of keys. Our solution consists of two parts:

A **Designing the overlay network:** The use of DHT routing structures over range-partitioned data is undesirable for a variety of reasons. In Section 3, we discuss hurdles in using these well-studied structures. We then build upon recent work on skip-graph routing topologies to produce a modified version of Chord that can be used with range-partitioned data.

B **Balancing storage load:** We can quantify the storage imbalance in a network using the *imbalance ration* σ, defined to be the ratio of the number of data

items in the most and least loaded nodes. In Section 4, we present an online algorithm that guarantees that σ is at most 32 at all times, even against adversarial (*any* sequence of) insertion and deletion of data items. Thus, no node is ever too overloaded or too underloaded. We also show that the amortized cost of moving data to maintain balance, when data is inserted and deleted, is a small constant. Finally, in Section 5, we show that the imbalance ratio is smaller in non-adversarial scenarios.

In summary, we design an overlay network (DBT) that offers the same asymptotic bounds as DHTs for the standard dictionary operations (data insertion, deletion, lookup), as well as node joins and leaves. In addition, it supports a range query operation with the cost of retrieving a fraction k of the data being $O(kn)$ where n is the number of nodes in the network.

2 Related Work

Range Queries: There have been multiple solutions offered for performing range queries in P2P systems. Some offer storage balance, but at the price of data-dependent query cost and data fragmentation [13], or at the price of having to live with approximation [2]. Others [3, 5, 7] offer exact and efficient queries, but run the risk of arbitrarily unbalanced distribution of data across nodes.

Storage Balance: Recent work has focused on ensuring the uniformity of the hash-space partitions [1, 9, 11, 14] as nodes join and leave the system. Byers et. al. [6] address the problem of non-uniform allocation of keys to the different partitions. All the above works assume that keys are unique in order to offer storage balance. Rao et. al. [10] perform dynamic load balancing by the use of multiple "virtual" nodes per actual node. When a node is overloaded, it attempts to exchange a heavily loaded virtual node for a lightly loaded one. However, their work offers no guarantees on the load imbalance ratio, while the use of virtual servers increases routing state as well as query costs.

In a concurrent work, Karger and Ruhl [8] provide an alternative solution to the storage balance problem. They mention that their scheme can be adapted for balancing range-partitioned data. The scheme is a randomized algorithm that offers a high-probability bound on the imbalance ratio, and is analyzed under a non-adversarial setting. However, the best achievable bound on imbalance ratio using this algorithm appears to be more than 200. The amortized cost of key insertions and deletions in their scheme, though constant, also appear an order of magnitude more expensive than ours.

3 Designing the Overlay Network

In this section, we consider how to interconnect nodes for answering range queries efficiently. Each node stores data in a particular range, and range may be arbitrarily sized in order to achieve storage balance. A straightforward solution is to use skip graphs [5, 7], which arranges nodes sequentially by their data ranges, and giving each node $O(\log n)$ additional links. This enables efficient routing to the left endpoint of a range query using $O(\log n)$ messages. Data in the required range can then be efficiently retrieved.

Let us dig a little deeper to understand why skip graphs work and explore alternative solutions. One may initially imagine that standard DHTs, built with the ID of a node being the left/right endpoint of the range it manages, also offer the efficient routing. However, the tying together of node IDs with the data ranges they manage has serious repercussions. When data ranges are arbitrarily sized, node IDs are not uniformly distributed in the ID space. In fact, node IDs will reflect the data distribution. Consequently, routing between a pair of nodes is not guaranteed to be in $O(\log n)$ hops, and there may also be great skew in node in-degrees. Additionally, any repartitioning of data across nodes necessarily involves modifying node IDs, which necessitates changes in the link structure.

Skip graphs escapes the above hurdles by making the link structure rely solely on the *ordering* of the nodes in the data space, rather than on the values of their range endpoints. In fact, skip graphs can be viewed as an adaptation of the Pastry DHT structure [5, 12]. This leads us to the question: Can other DHTs be modified to eliminate the above problems and retain the standard DHT routing properties? We show that the answer is "Yes", by modifying the Chord DHT structure. We show later that our modification of Chord is an improvement on skip graphs.

Imagine a circular sequence of n nodes, ordered clockwise by the data ranges they store. We define the *data distance* from node N to node N' to be the number of nodes on the clockwise path from N to N' in this sequence. Each node chooses a random *ID* in a circular

numeric space $[0, 2^D)$. We define the *ID distance* from node N to node N' as the clockwise distance from N's ID to N''s, *a la* Chord. Each node N forms a logarithmic number of links as follows. For each i from 0 to D, node N considers the set of nodes S_i that are in the ID distance range $[2^i, 2^{i+1})$ from N. It then forms a link to the node in S_i that is nearest to N in terms of *data distance*.

This structure forms the basis of *Modula*, our modified version of Chord that enables any node to route to the node responsible for any key using only $O(\log n)$ messages, while each node maintains only $O(\log n)$ links to other nodes. Note that the above bounds are completely independent of the exact ranges of keys stored by each node. Moreover, Modula can be maintained efficiently, with insertion or deletion of nodes requiring only $O(\log n)$ messages. Finally, Modula improves on skip graphs by allowing each node to maintain an arbitrary number of links, say k, to other nodes and provides a smooth trade-off between the number of links used and the number of routing hops necessary.

4 Storage Balance

In this section, we discuss how to ensure that a dynamic set of data items can be partitioned among a dynamic set of nodes while ensuring that the imbalance ratio σ is small. We first consider a setting where there is a static set of n nodes. As mentioned earlier, data is assigned to nodes by the use of range partitioning. When a data item is inserted, it is stored at the appropriate node. In reaction to this insertion or deletion, the node may execute the balancing algorithm and move data items in order to restore storage balance. Similarly, the deletion of a data item may cause the node involved to execute the balancing algorithm. The cost of the algorithm can be measured by the total number of data item movements performed.

We will show that our balancing algorithm requires only a constant amount of data movement per insert or delete operation (in an amortized sense), while guaranteeing that the imbalance ratio σ is at most 32.

We assume that nodes are labeled $N_1, N_2, \ldots, N_n$, in the order of the data ranges they manage. We refer to nodes managing adjacent ranges as *neighbors*. Let $L(N_i)$ denote the *load* (the number of data items stored) at node N_i. Storage balance is achieved by altering the ranges managed by the different nodes as data is inserted and deleted. It uses the following two simple mechanisms:

A **Neighbor Adjustment:** A pair of neighbor nodes, N_i and N_{i+1} may perform a neighbor adjustment, by transferring some amount of data from the node with higher load to the node with lower load, and adjusting the ranges they manage appropriately. Observe that this adjustment does not necessitate any changes in the overlay-network structure interconnecting nodes.

B **Stranger Adjustment:** A stranger adjustment involves three nodes, a node N_i, and a pair of neighbor nodes N_k and N_{k+1}. Assume w.l.o.g. that $L(N_k) \leq L(N_{k+1})$. When the total load on N_k and N_{k+1} is less than the load of N_i, node N_k can transfer its data over to N_{k+1}. The node N_k then takes over half the load of node N_i instead, thus becoming its neighbor. Observe that this step causes a change in the ordering of nodes due to N_k changing its position, and may require $O(\log n)$ messages to restructure the overlay network appropriately.

The balancing algorithm makes careful use of neighbor and stranger adjustments in order to ensure storage balance, while making sure that only a small number of such adjustments are performed, thus limiting the cost of performing balancing.

Let us start with a scenario where there are no data deletions. Each node N_i attempts a balance operation whenever its load crosses a power of two. Say, N_i's load increases to $2x+1$. If N_i has a neighbor with load at most $x/2$, it performs a neighbor adjustment to "top up" the neighbor to load x. If not, and neither of N_i's neighbors have a load more than $4x$, N_i finds the lightest loaded node in the system, say N_k. (We describe later how such an N_k may be found.) If N_k's load is at most $x/4$, N_i performs a stranger adjustment with N_k and one of $N_{k\pm1}$. In consequence, N_k's load becomes x.

This simple load-balancing algorithm proves sufficient to guarantee that load imbalance is at most 16 even against an adversarial insertion sequence. Moreover, the amortized cost of inserting a data item, measured as the number of data items that are moved in order to achieve load balance, is at most 4.

The deletion of data items is handled in a symmetric "inverse" scheme. Each node attempts to rebalance itself when its size reaches a power of two, say x, and it has lost at least half its data. If the node has a neighbor with load at least $8x$, it tops itself up to load $4x$ from this neighbor. Otherwise, it, together with its neighbor, attempts a stranger adjustment with the most loaded node in the system. The amortized cost of the deletion opera-

tion is again constant, although somewhat higher, and we state the following theorems about load imbalance and the costs of insertion and deletion.

Definition 4.1 *Let $\lceil\lceil x\rceil\rceil$ be the smallest power of 2 greater than or equal to x.*

Lemma 4.1 *The following invariants hold after any sequence of data insertions and deletions:*

(a) For any pair of neighbor nodes N and N', $L(N) \leq 8\lceil\lceil L(N')\rceil\rceil$.

(b) For any pair of nodes N and N', $L(N) \leq 16\lceil\lceil L(N')\rceil\rceil$. ∎

Theorem 4.1 *After any sequence of data insertions and deletions, the imbalance ratio σ is at most 32.* ∎

Theorem 4.2 *The amortized cost of an insertion is 4 and that of a deletion is at most 28.* ∎

Note that the cost of deletion is higher than that of insertion, which is not too surprising since it is easier to make a system unbalanced by halving the load of the least-loaded node, while it requires more insertions to double the load of the most loaded node. In future work, we hope to perform tighter analysis to lower the deletion cost.

Given the above algorithm for load balancing, we can easily adapt it to a dynamic, distributed setting where nodes may join and leave the system. First, insertion and deletion of nodes from the system can be handled easily. A new node is inserted to split the largest partition, while node deletion may be handled by handing over relevant data to a neighbor of the deleted node. Both these operations incur $O(D/n)$ data movement cost, where D is the total amount of data in the system. Second, the movement of data for storage balance takes a finite amount of time in a real system, and data insertions and deletions may continue to occur while balancing takes place. We, therefore, need to assume that the rate of data insertion and deletion is never so high as to swamp the system's resources and prevent it from completing storage balancing steps. We are currently characterizing the rates of dynamism that the above algorithm can handle gracefully.

Finally, we need to explain how nodes may find the least or most loaded node for stranger adjustments. Both these operations are easily implementable by the use of Modula as described earlier, and do not alter the asymptotic bounds on the number of messages necessary for insertions and deletions of data items and nodes. Alternatively, we can use a random sampling of nodes to identify lightly loaded candidates for stranger adjustments and avoid maintaining these structures. We postpone details of such randomization to future work.

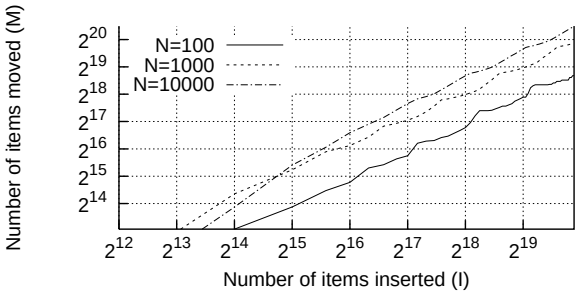

Figure 1: Data movement due to load balancing

5 Preliminary Results

We now present preliminary results that demonstrate the promise of our storage load balancing algorithm. The simulations reported here were performed on a static set of n nodes. We generated a sequence of item inserts in which keys follow a Zipfian distribution. The simulation starts with each node responsible for an equipartition of the key space, ensuring that the network does not have *a priori* knowledge of the incoming data skew. Each item i with key k that is inserted into the network is initially routed to the node N_i with the appropriate partition. Node N_i initiates load balancing steps when its $L(N_i)$ crosses a power of 2 as outlined in Section 4.

Figure 1 plots data movement caused by load balancing steps. The X-axis plots the the number I of items that were inserted into the network against a cumulative count M of item movements on the Y-axis. Both the axes are plotted on a logarithmic scale for clarity. The different curves are for different network sizes n. We see that the curves are almost linear, which indicates that each item moves, on average, a constant number of times. In fact, the constant is seen to be quite small, ranging from $\frac{1}{2}$ to 2, for the Zipfian distribution. For a given I value, there is a greater data movement for a larger number n of nodes. This result is not surprising as a fixed amount of data spread over a larger number of nodes results in a greater variance in load at nodes.

We now study the distribution of load across nodes as items are inserted into the network. Each curve labelled $(lo, hi]$ in Figure 2 plots the number of nodes (Y-axis)

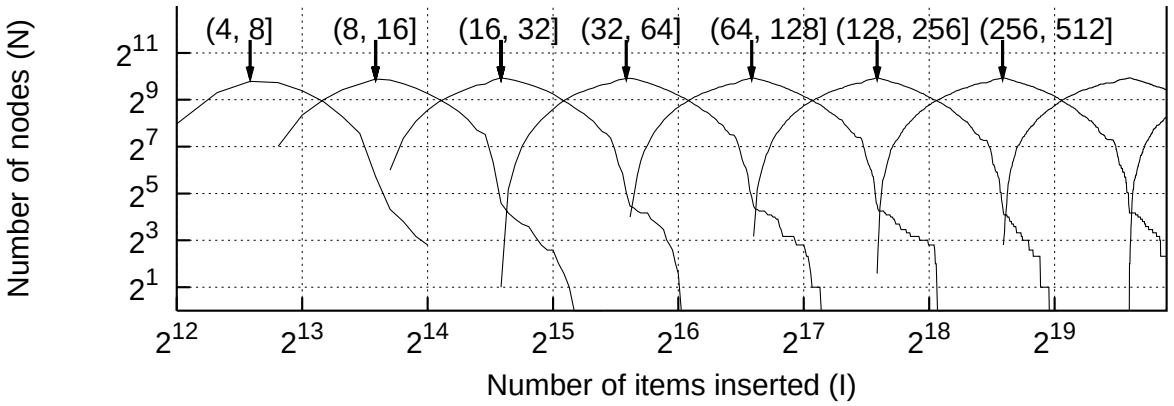

Figure 2: Threshold distribution for insert sequence

with a load in the corresponding interval observed as the number of items (X-axis) in the network increases. For example, when the number of items $I = 2^{13}$, there are about 300 nodes with loads in the interval $(8, 16]$ and about 700 nodes with loads in the interval $(4, 8]$. We note that for each I value, node loads exist only in *three consecutive* intervals. This indicates that all nodes in the network are within a (worst-case) factor of atmost 2^4. The curves also shift smoothly to the right with increasing I as the network adapts to more data.

6 Conclusions and Future Work

We have outlined a promising alternative to DHTs that allows efficient range queries, while providing strong storage balance. Our work is by no means complete. Among the various issues to be explored in future are:

- Adapting a variety of standard DHT structures to enable efficient routing over range partitions.
- Offering load balance when data items are weighted, which may be used to model data items of different sizes or with different query rates.
- Exploring the effects of replication on load balance.
- Studying randomized versions of our load balancing algorithm.
- Identifying lower bounds on the imbalance ratio achievable by any constant-cost online algorithm for load balancing.

References

[1] M. Adler, E. Halperin, R. M. Karp, and V. V. Vazirani. A stochastic process on the hypercube with applications to peer-to-peer networks. In *Proc. of STOC*, 2003.

[2] A.Gupta, D.Agrawal, and A.El Abbadi. Approximate range selection queries in peer-to-peer systems. In *Proc. CIDR*, 2003.

[3] A. Andrzejak and Z. Xu. Scalable, efficient range queries for grid information services. In *Proc. of P2P*, 2002.

[4] A.Silberschatz, H.F.Korth, and S.Sudarshan. *"Database System Concepts"*, chapter 17, pages 566–169. McGraw-Hill, 1997.

[5] J. Aspnes and G. Shah. Skip graphs. In *Proc. of SODA*, 2003.

[6] J. Byers, J. Considine, and M. Mitzenmacher. Simple load balancing for distributed hash tables. In *Proc. of IPTPS*, 2003.

[7] N. J. A. Harvey, M. Jones, M. Theimer, and A. Wolman. Skipnet: A scalable overlay network with practical locality properties. *Proc. of USITS*, 2003.

[8] D. R. Karger and M. Ruhl. New algorithms for load balancing in peer-to-peer systems. In *Prog. IRIS Student Workshop*, 2003.

[9] M. Naor and U. Wieder. Novel architectures for p2p applications: The continuous-discrete approach. In *Proc. of SPAA*, 2003.

[10] A. Rao, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica. Load balancing in structured P2P systems. In *Proc. of IPTPS*, 2003.

[11] S. Ratnasamy, P. Francis, M. Handley, and R. Karp. A Scalable Content-Addressable Network (CAN). In *Proc. of SIGCOMM*, 2001.

[12] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large scale peer-to-peer systems. In *Proc. of MIDDLEWARE*, 2001.

[13] S.Ratnasamy, J.M.Hellerstein, and S.Shenker. Range queries over dhts. Technical Report IRB-TR-03-009, Intel Tech Report, 2003.

[14] I. Stoica, R. Morris, D. Karger, M. Fran Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proc. of SIGCOMM*, 2001.

[15] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, U. C. Berkeley, Computer Science Division, 2001.