Implementing Range Queries with a Decentralized Balanced Tree Over Distributed Hash Tables

Nuno Lopes[⋆] and Carlos Baquero

CCTC-Department of Informatics
University of Minho
Braga, Portugal

Abstract. *Range queries, retrieving all keys within a given range, is an important add-on for Distributed Hash Tables (DHTs), as they rely only on exact key matching lookup. In this paper we support range queries through a balanced tree algorithm, Decentralized Balanced Tree, that runs over any DHT system.*

Our algorithm is based on the B^+-tree design that efficiently stores clustered data while maintaining a balanced load on hosts. The internal structure of the balanced tree is suited for range queries operations over many data distributions since it easily handles clustered data without losing performance.

We analyzed, and evaluated our algorithm under a simulated environment, to show it's operation scalability for both insertions and queries. We will show that the system design imposes a fixed penalty over the DHT access cost, and thus inherits the scalability properties of the chosen underlying DHT.

1 Introduction

Distributed Hash Table (DHT) systems [1–4] are used as efficient distributed dictionary implementations, offering a scalable and robust P2P framework that efficiently locates objects given a key [5, 6]. However, such efficiency is achieved by an exact key matching lookup interface. The discrete key lookup interface uses an hash function on the key value to locate objects. This hash function removes locality properties from keys which restricts it's use for range queries. A range query consists in retrieving all keys that fall within a specific range interval. Range query is a desired feature when using data that is indexed by contiguous values (consider for example, numeric spatial coordinates).

Previous systems have offered range queries by either using specially designed structures [7–9] or building on top of generic DHTs [10–12]. Because the first class of systems is bound to some particular basic storage structure, it offers a limited solution that may not be as efficient as some DHT systems are. This

[⋆] Supported by a Ph.D. Scholarship from FCT-Foundation of Science and Technology, the Portuguese Research Agency.

makes the second class of systems, building a tree structure over a generic DHT, the most flexible choice. By building on a generic DHT, one can choose the best DHT implementation available for the system, while maintaining the range query functionality. However, recent structures available in the literature: Prefix Hash Tree (PHT) [10] and Distributed Segment Tree (DST) [12], are sensitive to clustered data.

Clustered data is common on real data sets, in particular when data depicts geographical placement of items that are tied to human activity. For instance, the concentration of WiFi access points is clustered around cities and along roads [10], so that sharing access point locations and querying for nearby access points will yield a response depicting clustered data.

This steams from population concentration patterns, where clustered data typically follows a power-law distribution, or a combination of power-laws centered on several focus points [13]. This common setting depicts a few higher density key regions while most of the data is sparsely distributed across the key domain.

In this paper we show that the Decentralize Balance tree (DEB tree) algorithm, an algorithm based on the B^+-tree design [14], offers a structure suitable for storing clustered data on block oriented storage (in this case a DHT) while supporting range queries without loss of performance.

The algorithm is capable of running on top of any generic DHT without incurring in a significant overhead. Insertions can be reduced to O(1) complexity in terms of DHT operation requests, if caching is used at clients. Each DHT request cost depends on the DHT implementation selected. In this sense, the scalability of the tree design closely follows the scalability properties of the used DHT.

Query cost depends on the data stored on the index rather than on the range size. Additionally, it is possible to parallelize the query operation, reducing latency to a logarithmic factor on the stored data size in terms of DHT operations.

2 Related Work

Related work can be divided into two groups: range query systems with specific underlying structures and range query systems using a tree structure over a generic DHT interface. Due to space restrictions we will focus on the later group and only provide a brief mention to some systems in the first group.

Mercury [7] supports multi-attribute range queries using a circular overlay, similar to Chord, but without key hashing, so that locality is preserved. Skip graphs [8] are a generalization of skip lists in which nodes are part of distributed linked lists that form a distributed binary tree. Baton [15] builds a binary balanced tree structure where each tree node is mapped into a peer host and maintains connections to the peers containing tree node neighbours.

All previous systems maintain a specific routing algorithm that offers logarithmic cost in terms of network hops to access data. Our DEB-tree algorithm assumes a generic DHT implementation is used. Such assumption enables the use of one-hop DHTs [5, 6] or the use of specific DHTs to handle churn efficiently [4] or efficient load-balancing extensions.

Chawathe et al. proposed the use of a Prefix Hash Tree (PHT) for building a trie-based structure over a generic DHT [10]. The main difference between the PHT and DEB tree is that our structure is not sensible to clustered data. The PHT, in order to adapt to clustered data places leaves at different tree levels, causing an irregular tree structure that has impact on the query performance. Data is placed on tree nodes according to a prefix value, which is also used as the block identification scheme. Since the identification scheme is independent from the data itself, it is possible to access any tree block directly without knowing the tree structure in advance. On the other hand, this independence between data and block ids can create overloaded blocks storing a large number of objects that share a common prefix value. Our algorithm adapts efficiently to clustered data distributions, creating a balanced tree with bounded block sizes, but cannot directly access any block without an initial tree traversal.

Zheng et al. presented a Distributed Segment Tree (DST) algorithm, also running over a generic DHT [12]. This binary tree structure is static, where all tree nodes have a pre-defined range limit. Static range limits are incapable of handling clustered data properly, possibly generating either empty or overfull nodes depending on the key distribution. Just like on the PHT, the block identification does not depend on the data, allowing direct access to any tree node. This algorithm allows access to any block directly, a feature that is used on queries to reduce the number of accesses by replicating data on additional tree nodes. This design is very efficient for small queries, at the cost of using more storage. However, when using clustered data, even queries for a small range can produce large results, requiring additional accesses.

Our tree algorithm assigns node ranges dynamically according to the data distribution, which tends to create a good storage distribution for data even in presence of strong clustering. Furthermore, queries do not make redundant accesses to the tree but instead access only the minimum amount of nodes necessary to retrieve the (complete) answer.

3 Decentralized Balanced Tree

In this section, we will review the DEB tree algorithm, which was described in [16], and show how it can be adapted under a generic DHT to support range queries.

3.1 Tree Structure

The tree structure, following the B^+-tree design, is made from *leaf* and *internal* blocks. Leaf blocks contain data items stored on the tree while internal blocks contain only references to children blocks. All leaf blocks are at the same tree level, that is, all leaf blocks are at the same distance from the root. This feature creates a logarithmic bound on the number of block accesses to reach any leaf block.

To maintain high availability even during tree structure maintenance, each block keeps a *next* block reference, that points to the consecutive block at the same tree level [17]. The block size is bounded by the block's maximum size parameter s. Every block must have at least $s/2$ items and at most s items, except the root block which only has the maximum bound [14]. Additionally, each block contains a limit interval (minimum and maximum values) that specifies the range of data the block is responsible for.

3.2 DHT mapping

We use a single DEB tree to store the entire index data. The support for multiple spaces or dimensions can be obtained using one of two methods: 1) using multiple trees or 2) using a space-filling curve function. The support for multiple trees requires the capability to distinguish blocks of different trees on the same DHT key domain. A single tree can only store and compare values (or range intervals) on a linear space. Storing n-dimensional data on the system is possible by mapping the n-dimensional space into a single-dimension space with a space-filling curve [10].

Each tree block is stored on the DHT host responsible for the hash value of the block id. Although block ids are generated dynamically, they must be globally unique (on the DHT key domain) and not collide or force the change of an already existing identification. This would require moving the block's data on the DHT and updating all the references on other blocks pointing to it.

The block identification is defined as the tuple: $\langle level, minlimit \rangle$, where the *level* field identifies the tree level this block is and the *minlimit* distinguishes the block inside the level.

3.3 Operation Request Model

Access to block data uses the typical *put* and *get* DHT interface. The modification of a block's content requires a three cycle procedure at the caller: *get*–execute–*put* operations. For each block access, a *get* operation must be issued to retrieve the block's data from the DHT to the caller. If the data is modified, an additional *put* operation must also be issued to store the new data on the DHT. This design was used by both PHT and DST to implement a tree structure on top of a generic DHT, like the OpenDHT system [18]. However, unlike the previous structures, the DEB algorithm does not support concurrency on some operations. To operate correctly, the algorithm requires some mechanism to detect concurrent modifications of the same object, in this case the block's content.

We propose a simple extension to the *get* and *put* semantics: to include a logical time stamp parameter, so that concurrent puts can be detected and prevented from happening. This extension would work as follows. When a *get* is made, the current time stamp, an integer, is returned together with the data.

When a put is made, the caller sends an increased time stamp value, indicating a modification. If another caller has, in the meantime, already putted a new value for that key, the DHT host receiving the put request can detect that both puts are concurrent and abort the second.

4 Index User Interface

In this section we will describe the two index operations available to the user: insertions (or removals) and range queries. The user calls these operations to store or retrieve data items from the system. Each client host must explicitly store data in the system so that it can be later retrieved by range queries.

4.1 Insertion and Removal

The insertion operation stores a data item into the index. Data items consist on two fields: 1) the location key, which places the item on the space and 2) the data item value, that will be fetched if it's location is contained inside range queries. In order to insert a data item into the tree, the insertion operation must first locate the correct leaf block by performing a top-down traversal of the tree and then adding the new data to the leaf content's. The removal of an item from the tree is identical to the insertion except for the removal of the item on the leaf block. When leaf blocks get full, the caller must also perform a split operation on the block by transferring half of the data to a new block. We omit here the details of the split operation.

Both insertion and removal operations are bound by the cost of vertically traversing the tree, the tree height, which is $O(\log I)$ for I stored items in the tree. Each tree access is equivalent to a *get* DHT call except for the leaf block that requires both *get* and *put* calls. Any DHT call complexity depends on the DHT implementation, ranging from $O(\log N)$ to $O(1)$ for N hosts. Overall, the insertion is bound either by $O(\log I \log N)$ or $O(\log I)$, respectively.

4.2 Range Queries

A range query for the $[s, t]$ interval consists in retrieving all the items on the tree whose locations are contained in the interval. Since items are stored on leaf blocks, the operation is divided in two steps: 1) locate the leaf blocks whose range limits intersect $[s, t]$ and 2) retrieve such blocks. The first step is identical to the location step in the insertion case. This step complexity grows logarithmically with the number of items inserted on the tree, which corresponds to the tree height, $O(\log I)$. The second step involves *get*ing the targeted leaves. The number of *get*s used to retrieve the answer depends directly on the number of stored items whose location keys are contained in the range interval ($I_[s, t]$) and not on the size of the query range itself. The size of $I_[s, t]$ depends on the distribution of data and will be analyzed in more detail in the Simulation section.

The retrieval of all leaves can be made by following the *next* pointer at each leaf starting at the first ordered leaf that contains s. The downside of this approach is increasing the query latency. Recall that each block access requires a DHT *get* call, whose complexity depends on the DHT implementation. To reduce the query latency, it is possible to parallelize the block accesses by having the client retrieve simultaneously all tree blocks of the same tree level that intersect the range interval. This procedure reduces the latency to the tree height ($O(\log I)$), at the cost of additional concurrent block accesses. The number of *get*s on leaves remain the same.

4.3 Internal Block Caching

User oriented operations, insertions and queries, always require a tree top-down traversal from the root block to reach the target leaf block. We eliminate the bottleneck on upper level blocks by caching internal blocks at the callers. Caching reduces the top-down traversal cost while maintaining the operation's correctness as internal blocks serve only for location purposes. Furthermore, even if stale cached versions are used, the caller either succeeds in finding the correct block or detects it is using a stale cache entry. To reduce cache update overhead, caching uses a passive update mechanism in which clients only refresh block data after detecting tree inconsistencies.

Caching can reduce the cost of a vertical tree traversal from $O(\log I)$ to $O(1)$ when all internal blocks except leaves are already present at the client. In the worst case, it incurs in constant overhead for the client to update it's entire local cache.

5 Simulation

We implemented the DEB tree algorithm over a custom-made simulator in Python. The basic DHT functionality was simulated using a local storage. However, using a real DHT platform like OpenDHT would be quite straightforward except for the concurrency issue already described.

The simulation data was synthetically generated using either uniform and power-law distributions in order to determine the impact of clustered data on the algorithm. In the clustered case we considered a small number of focus points of clustering chosen uniformly at random.

Points are randomly allocated to the left and right of the focus points under a Pareto distribution with density $f(x) = \frac{kb^k}{x^{k+1}}$. We used a shape parameter $k = 0.5$ and set the minimum x to be 1 by making $b = 1$. Each random value obtained from the distribution was decremented by 1 (transforming the range from $[1, +\infty[$ to $[0, +\infty[$) and either added, or subtracted, to the position of a focus point. The resulting data set mimics the usual distribution of population in a geographic linear space.

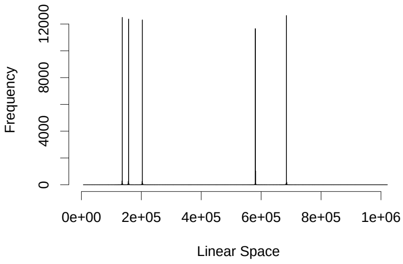

Fig. 1. The data point distribution for a five cluster configuration.

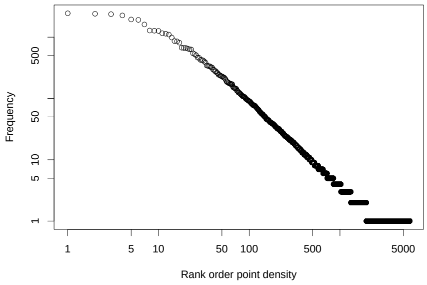

Fig. 2. Log-log rank order.

5.1 Insertion

The insertion simulation consisted in placing 2^{16} points within a 2^{20} linear space ($\approx 10^6$) into the system sequentially and determine the adaptation of the algorithm to clustered and uniform data. We used two data distributions for the insertion points: an uniform distribution and a five cluster distribution. Figures 1 and 2 show some details of an actual five cluster sample. The first Figure shows the concentration of points in the linear space and the second shows rank ordered point densities. Here we can observe the five clusters and the expected linear decrease of point densities in a log-log scale, the graphical signature of power-laws.

We ran the simulations for three maximum block sizes: 64, 256 and 1024 items, and repeated each experiment 50 times to exclude random variations. The DHT is simulated and we assume that the DHT hash function uniformly distributes blocks across the system hosts.

Figure 3 shows the ECDF on the (average) number of stored items per block. The tree algorithm balances data perfectly, as we can see that simulations using the same block size but with different data distributions, whether uniform or clustered, tend to create identical ECDF's. The block usage, the number of stored items per block, varies between 50% and 100% of the block size with the single exception of the root block. The largest block size case, 1024, shows two clusters: one around half the block size and the other around the full block size. This clustering is due to the number of inserted points being large enough to split blocks but small enough to fill them completely. A greater number of points ($\approx 2^{16}$) would make the 1024 line more similar in shape to the 256 and 64 lines.

To insert a single item on the tree, a vertical tree traversal is made, requiring the access to one block for each level. The cost of insertions is therefore equal to the tree height. The tree height h is defined by the expression $h \leq \log_t \frac{n+1}{2}$ where $2t$ is the block size and n is the number of stored items. We measured the tree heights for all simulations, resulting in 3 levels for trees with block sizes of 64 and 256 items, and 2 levels for the 1024 item block size case.

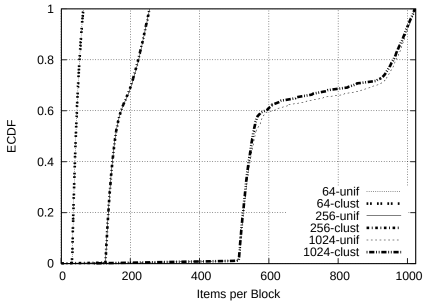

Fig. 3. ECDF (Empirical Cumulative Distribution Function) on the number of stored items per block.

Block	Data	Gets		Items/Gets	
Size	Dist.	mean	s-d	mean	s-d
64	unif	222.1	171.6	42.9	3.5
	clust5	245.7	308.7	23.6	19.7
256	unif	59.2	46.1	152.9	25.1
	clust5	63.6	77.4	82.3	81.2
1024	unif	15.2	11.4	559.8	142.1
	clust5	16.9	20.2	306.2	314.0

Fig. 4. The simulation of a query set over different storage distributions (uniform and five clusters) and different block sizes.

5.2 Range Query

We generated a set of 500 range queries using a middle point and a range size around that middle point. The middle points were generated from an uniform distribution on the linear space. The range size was generated from a normal distribution with 0 mean and a standard deviation of 5% of the space length. We measured query performance as the number of leaf blocks that had to be retrieved in order to obtain a complete answer. We did not take into account internal blocks since their cost is constant for all queries, equal to $h - 1$ DHT accesses. Clients can cache internal blocks locally, removing the height traversal cost. Furthermore, queries can be parallelized for each tree level, reducing query latency to the tree height in the number of hops at the cost of increased bandwidth.

Figure 4 shows the results of simulations with two different insertion point distributions (uniform distribution and five cluster distribution) and three maximum block sizes: 64, 256 and 1024 items.

We used the same query set for all simulations. Measured the mean and standard deviation of the number of DHT *get* calls (on leaf blocks) and the ratio between items retrieved and gets made per query. Since we used the same insertion set for simulations with the same insertion data distribution, these simulations returned the same results no matter what the block size was (an average of 9855 items for the uniform case and 10913 average items for the clustered case).

As the block size increases, we see that the mean number of DHT gets necessary to reply the query decreases, along with it's standard deviation, since larger blocks are capable of returning more data. The mean ratio also increases because of the greater block capacity, where each get request returns more items. However, the ratio's standard deviation also increases, meaning that queries obtain increasingly different amounts of items per get on each query. This can be explained by the larger block capacity storage distribution, where the size of blocks is distributed between half and full maximum size. Larger block capacities will generate blocks spawning a wider load variation, which is reflected on the different number of items each block returns and consequently on the ratio's standard deviation.

Table 4 also compares the algorithm efficiency when running a query set on point data created by an uniform distribution or a five cluster distribution. As expected, the number of gets per query were about the same for both distributions. However, the standard deviation was greater in the clustered case. An uniform distribution is likely to return the same approximate number of items for queries across all the linear space, whilst the same queries on a clustered distribution are likely to have more disparity on the number of items returned depending on the density of stored data at the query range area, hence the higher standard deviation.

The item/get ratio mean is higher for the uniform case across all block sizes. The standard deviation, on the contrary, is always significantly smaller. These results show that the algorithm adapts perfectly to the storage data distribution. When running over uniform data, the standard deviation is low because all queries tend to receive the same amount of results. When running over clustered data, the higher standard deviation shows that the algorithm returns different amounts of data for the same query set, which is in accordance to the clustered distribution of data with many sparse regions and a few highly dense regions.

6 Conclusion

In this paper we show how our DEB tree algorithm, which is based on balanced trees, can easily provide a scalable range query implementation over generic DHT systems.

The solution induces an even distribution of items per DHT block and consequently balances the storage and network load on the hosts that support the DHT. We have considered two opposite data sets, one with data uniformly distributed in space and another exhibiting highly clustered data. These two scenarios induce almost indistinguishable patterns of data allocation in the DHT, depicting similar ECDFs.

Finally, we considered the effects of a typical range query scenario. Users query for items (e.g. WiFi access points) within a given spatial distance of their current location. The choice of maximum block size is the driving factor that dictates the number of DHTs requests that are needed for a given usage pattern. An optimal size must take into account not only the expected usage pattern, but also the number of stored items and the combined effects of caching and concurrent DHT gets.

This approach presents a clear advance over previous systems by providing a design that is mostly insensitive to the presence of clustered data, while building on of-the-shelf DHT middleware.

References

1. Stoica, I., Morris, R., Karger, D., Kaashoek, F., Balakrishnan, H.: Chord: A scalable Peer-To-Peer lookup service for internet applications. In: Proceedings of the ACM SIGCOMM'01 Conference. (2001) 149–160
2. Rowstron, A., Druschel, P.: Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In: Proceedings of the 18th IFIP/ACM International Conf. on Distributed Systems Platforms, Germany (2001) 329–350
3. Ratnasamy, S., Francis, P., Handley, M., Karp, R., Shenker, S.: A scalable content addressable network. In: Proceedings of the ACM SIGCOMM'01 Conference. (2001) 161–172
4. Rhea, S., Geels, D., Roscoe, T., Kubiatowicz, J.: Handling churn in a dht. In: Proceedings of the USENIX Annual Technical Conference. (2004) 127–140
5. Gupta, A., Liskov, B., Rodrigues, R.: Efficient routing for peer-to-peer overlays. In: Proceedings of the 1st Symposium on Networked Systems Design and Implementation (NSDI'04). (March 2004) 113–126
6. Ramasubramanian, V., Sirer, E.G.: Beehive: O(1) lookup performance for power-law query distributions in peer-to-peer overlays. In: Proceedings of the 1st Symposium on Networked Systems Design and Implementation (NSDI'04). (March 2004) 99–112
7. Bharambe, A., Agrawal, M., Seshan, S.: Mercury: Supporting scalable multi-attribute range queries. In: Proceedings of SIGCOMM 2004. (2004)
8. Aspnes, J., Shah, G.: Skip graphs. In: Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms. (January 2003) 384–393
9. Zhang, C., Krishnamurthy, A., Wang, R.: Brushwood: Distributed trees in peer-to-peer systems. In: Proceedings of the 4th International Workshop on Peer-to-Peer Systems (IPTPS'05), New York, USA (February 2005) 47–57
10. Chawathe, Y., Ramabhadran, S., Ratnasamy, S., LaMarca, A., Hellerstein, J., Shenker, S.: A case study in building layered dht applications. In: Proceedings of the ACM SIGCOMM'05 Conference. (2005) 97–108
11. Gao, J., Steenkiste, P.: An adaptative protocol for efficient support of range queries in dht-based systems. In: Proceedings of the 12th IEEE Int. Conference on Network Protocols (ICNP'04). (2004) 239–250
12. Zheng, C., Shen, G., Li, S., Shenker, S.: Distributed segment tree: Support of range query and cover query over dht. In: Electronic publications of the 5th International Workshop on Peer-to-Peer Systems (IPTPS'06), California, USA (February 2006)
13. Zipf, G.: Human Behaviour and the Principle of Least Effort. Addison-Wesley (1949)
14. Cormen, T., Leiserson, C., Rivest, R.: Introduction to Algorithms. MIT Press (1989)
15. Jagadish, H., Ooi, B.C., Vu, Q.H.: Baton: A balanced tree structure for peer-to-peer networks. In: Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway (September 2005) 661–672
16. Lopes, N., Baquero, C.: Using distributed balanced trees over dhts for building large-scale indexes. Technical report, University of Minho (October 2006)
17. Johnson, T., Krishna, P.: Lazy updates for distributed data structures. In: Proceedings of the 1993 ACM SIGMOD Intl. Conf. on Management of data. (1993) 337–346
18. Sean Rhea, B.G., Karp, B., Kubiatowicz, J., Ratnasamy, S., Shenker, S., Stoica, I., Yu, H.: Opendht: A public dht service and its uses. In: Proceedings of SIGCOMM 2005. (August 2005) 73–84