Solving Range Queries in a Distributed System

Praveen Yalagandula

1 Introduction

The goal of the project is to design and build a scalable distributed discovery system for documents that (i) supports both simple queries and range queries on document names, (ii) supports efficient insertion and deletion of documents, (iii) distributes both storage and access loads uniformly among all the participants, and (iv) is efficient in terms of the communication cost incurred for responding to queries.

The ultimate goal of the project is to support a full-fledged keyword search. A full-fledged keyword search system will support the search based on regular expressions. On the other extreme, a very simple system will just support single keyword based lookups. Distributed Hash Tables (DHT) based systems typically support only simple keyword searches. Several intermediate schemes are possible – that support range queries on single dimension (e.g., [1]) and that support multi-dimensional sequence of keywords and ranges (e.g., [5]). In this project, we focus on supporting range queries on single dimension.

Our approach is to combine ideas from Extendible Hashing [2] and Distributed Hash Tables.

2 Related Work

Mainly two related papers: (1) Skip Graph [1] and (2) Squid [5].

2.1 Skip Graph

Skip Graphs, proposed by Aspnes et al [1], is a distributed data structure for supporting range queries. The structure is similar to a skip list (Figure 2.1) and uses ideas from Distributed Hash Tables [3] to achieve fault tolerance and load balance in terms of access loads. In this approach, each resource that a node has to share is assigned a random ID, called membership vector, and the resources are arranged in a non-descending sorted order according to their names or values. A DHT is constructed on the resources in the following way – for each prefix length of the membership vector of a resource, the resource has a right and left pointers to nearest resources that have same prefix. Figure 2.1 illustrates these pointers. A range query or simple query can be satisfied as in the search procedure in a skip list.

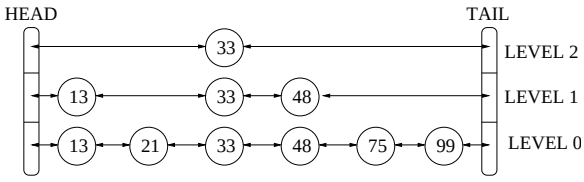


Figure 1: A skip list (source: [1])

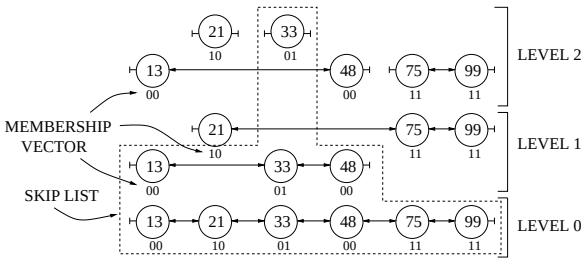


Figure 2: A skip graph with $\lceil \log N \rceil = 3$ levels (source: [1])

Skip graphs support range queries, support efficient insertion and deletion of new resources, is load-balanced in terms of both storage and access loads, is fault-tolerant and is efficient in terms of number of messages. The main disadvantage of this data structure is that it forms DHT on the resources than the physical nodes present in the system. If each node in a N node system has K documents or resources to share, then each node needs to keep track of $O(K \log(N.K))$ pointers to other nodes. Hence this system is not scalable with the

number of resources or documents.

2.2 Squid

Squid, proposed by Schmidt et al [5], supports multidimensional range queries. They map the n-dimensional data to 1-dimensional space using Hilbert Space Filling Curves. The one-dimensional data is then mapped to nodes arranged in a linear fashion along the NodeId ring. While this mapping allows range queries to be performed efficiently, the scheme loses load balancing property inherent to the DHTs.

The authors propose two load-balancing schemes – (a) Load balancing at join time and (b) Load balancing at run time. In former technique, a new node chooses multiple IDs, joins the network and then discards all but one ID that will place it in the most loaded part of the network. This technique is both expensive – $O(n \log N)$ for joining at n places in a N node network ($O(n \log^2 N)$ in case of chord) – and is not sufficient to achieve the load balancing in the face of document insertions and deletions.

Two schemes are presented for load-balancing at the run time (1) exchange the load with neighbors and (2) each node hosts multiple virtual nodes. The first scheme incurs an $O(N \log^2 N)$ communication cost and hence is very expensive to be performed periodically. The second scheme is good for load-balancing at the expense of increase number of DHT pointers to maintain at each nodes. With each virtual node containing a single document, this scheme is same the Skip Graph. A node hosting k virtual nodes needs to maintain connections to $O(k \log N)$ DHT neighbors; hence, the scheme is not scalable with the number of documents in the system.

3 Our approach

Supporting range queries is harder than supporting simple single keyword queries. Hash tables provide efficient constant lookup, insertion and deletion costs but can not support range queries. Balanced binary trees, B-trees, Tries, etc., support range queries at the cost of $O(\log N)$ insertion, lookup, and deletion complexity. Our approach is the fusion of Tries with hash tables: arrange data in a Trie structure and map the tree onto physical nodes using DHT techniques. This enables us to support range queries while exploiting the load-balancing property of the DHTs but at the cost of increased insertion, deletion, and lookup costs. We propose caching based optimizations

3.1 Simple Algorithm

In this project, we consider range queries on one-dimensional keywords only. Further, we assume that keywords are drawn from a small alphabet, say Σ. We use kleene star notation $*$ to express the ranges. We assume that a DHT is already constructed on the nodes in the system and can route messages for a key with ID to the corresponding responsible node (whose NodeId is closer to ID than other nodes in the system). We will denote a node responsible for the hash of a string S by $[S]$.

Insertion and deletion of Documents Initially, each node has zero documents. When a new document is inserted, the entry is routed to the node $[\Sigma*]$, say the root node. When the number of entries at the root node exceed a *blockSplitThreshold* factor, then the block of entries is split into $|\Sigma|$ blocks, one block for each character of the alphabet, and are spread to nodes $[a\Sigma*]$ for $a \in \Sigma$. This process is further recursively repeated at the child nodes. To be able to merge back the entries upon deletion of documents, the nodes need to keep track of their $|\Sigma|$ child nodes. when the total number of entries over all children fall below *blockMergeThreshold* for a node, then the node collects back all the entries from its $|\Sigma|$ children. Note that the *blockSplitThreshold* and *blockMergeThreshold* parameters satisfy the following invariant: *blockSplitThreshold* $\geq$ *blockMergeThreshold*.

Lookup Any lookup request, say $abc*$, is routed to the root node $[\Sigma*]$. If the entries were already split, then the request is passed down to the node $[a\Sigma*]$ and then down to $[ab\Sigma*]$ and so on.

Example A simple example depicting the insertion of keywords is shown in Figure 3.1. In the example, we assume that the *blockSplitThreshold* factor is 4.

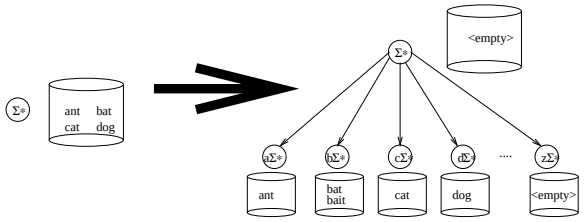


Figure 3: An example Trie construction with split threshold of 4

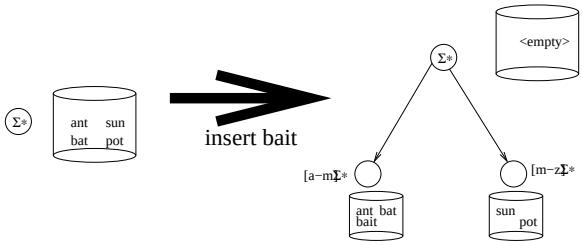


Figure 4: An example Trie construction with 2-way splitting.

Discussion The insertion, deletion and lookup costs increase from $O(\log n)$ in DHTs to $O(k \log N)$, where k is the size of the document name. The procedure achieves load balance in terms of storage, but the root node is accessed on each operation and hence the access load is not distributed fairly. Further splitting and merging are costly as $|\Sigma|$ nodes have to be accessed and thus atleast $|\Sigma|$ number of messages.

Problem 1: *Access load is not uniformly distributed.*

Fix: Use binary search. For example, start the lookup for keyword *computer* at node $[comp\Sigma*]$ and proceed to either node $[co\Sigma*]$ or $[comput\Sigma*]$ based on the information at the node $[comp\Sigma*]$. The lookup cost decreases to $O(\log k \log N)$ from $O(k \log N)$ and the access load also gets distributed uniformly among the nodes.

Problem 2: *Splitting and merging are costly.*

Fix: Instead of splitting $|\Sigma|$ ways at a node, a more coarse granular splitting can be done — 2-way, 4-way or some b-way to reduce the costs of splitting and merging. This increases the depth of the lookup tree and hence the trade off is increased lookup time: $O(\log_b |\Sigma|. \log k. \log N)$. An example showing the 2-way splitting is shown in the Figure 3.1.

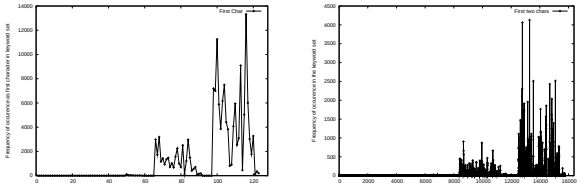
4 Issues

Access Load Balance and Fault Tolerance Data replication at neighboring nodes on the logical NodeId ring. Replication provides fault tolerance and also helps in load-balancing the access loads. The keywords and keyword ranges that are most accessed are replicated on more nodes to offset the load from one single node. Consistency will be an issue with replication and a simple eventual consistency model will be efficient in terms of communication costs and is generally acceptable for the application domain in consideration.

Optimizations: Caching to reduce the number of steps in lookup, insertion and deletion. Nodes can cache the information about how far down the tree is already split to optimize the search performance. Result caching can also be done to further optimize the performance.

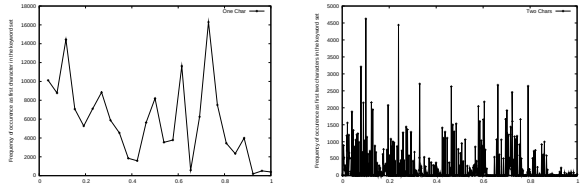
Supporting General Expressions The algorithm described above supports the range queries with wildcard character only at the end. Range queries with kleene star appearing anywhere else can also be supported albeit at the increased communication cost. For example, a range query for $\Sigma * uter$ will need to be sent to all of the leaves of the Trie structure and hence possibly all nodes in the system. A query like $com\Sigma * ter$ can be efficiently answered compared to query for $\Sigma * uter$. The length of the prefix before the appearance of a kleene star in the query greatly effects the performance of the system.

Choosing Thresholds A large value of *blockSplitThreshold* will imply that all entries are stored in one or few places causing the storage load imbalance in the system but provides an efficient range query support. A small value gives a good storage load-balancing but at the cost of increased Trie depth and hence an increased lookup, insertion and



(a) (b)

Figure 5: Frequency distribution of first one and first two letters in the chosen keyword set.



(a) (b)

Figure 6: Frequency distribution of first one and first two letters in the chosen keyword set after converting to lower case.

deletion costs.

We propose a dynamic scheme for picking appropriate threshold value that minimizes the tree depth while ensuring the storage load balance. Initially, the *blockSplitThreshold* is set to a moderate value, say 1MB. When a node is assigned more than a few number of leaf nodes of the Trie, then it increases *blockSplitThreshold* and decreases the *blockMergeThreshold* value.

5 Evaluation

Workload We use the WordNet's word list available for free download from `dict.org`. There are about 150000 words in the database.

In Figure 5(a), we plot the frequency of words against the starting ASCII character in our word set. As expected, most words in our workload start with either capital or small alphabets in the English language. Words with small alphabet starting are dominant than the ones starting with the capitals. In Figure 5(b), we plot the frequency of words taking first two characters of the word into consideration. These two graphs clearly show that the distribution of words has a very high variance factor. An approach similar to Squid, where order-preserving hashing is used to map the keywords to the set of ordered nodes, will suffer from storage load-imbalance. To further substantiate this point, we converted all words into lower case letters and plot the distribution only based on the 26 alphabets of the English language. Figures 5(a) and 5(b) depict the frequency distribution of words for the starting one and two characters.

Encoding of alphabet matters – while a straightforward ASCII encoding does not evenly distribute the keywords, using something like a hamming code based on the occurrence rate of the characters might load-balance the keywords across bit space evenly. Further investigation necessary to quantify this. For now simple straightforward encoding in the same order as the ASCII ordering. Figure 5 illustrate the encoding scheme we have used in our simulations. We encode each character with 5 bits making sure that the alphabets are evenly spaced in the 32 element size space.

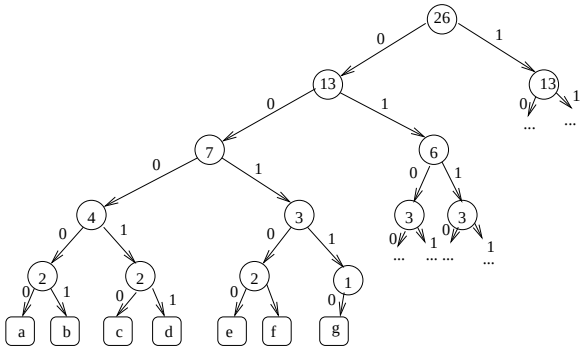


Figure 7: Encoding tree for representing 26 lower case alphabets with 5 bits

We construct a Trie structure on the keyword set. Two parameters that affect the construction are *blockSplitThreshold*, which we will refer to as *Split Threshold (ST)*, and the granularity at which the splitting is performed, which we will call *Split Granularity (SG)*. The split granularity is measured in the number of bits. A split granularity of 5 denotes that the split is done based on the 26 alphabets.

In Figures 5, we plot the number of buckets and the average depth of the words in the Trie struc-

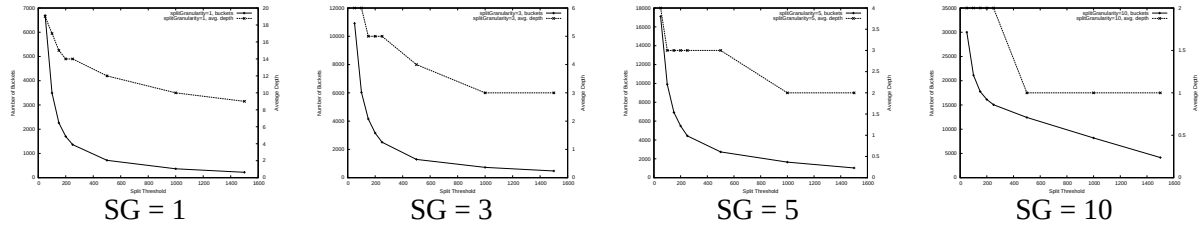


Figure 8: Number of buckets and the average depth of the Trie structure after inserting all keywords with split threshold (ST) for different values of split granularity (SG) parameter.

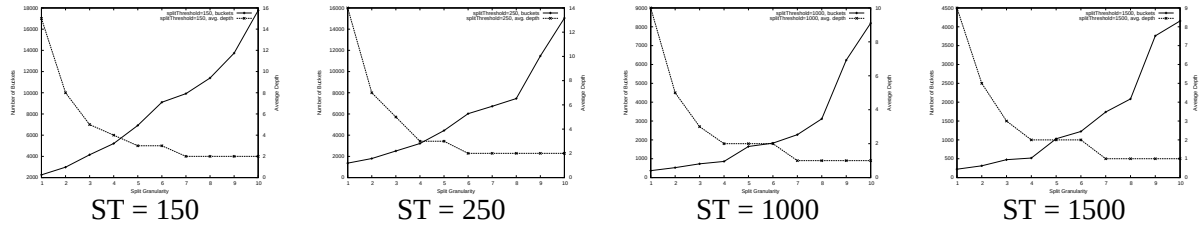


Figure 9: Number of buckets and the average depth of the Trie structure after inserting all keywords with split granularity (SG) for different values of split threshold (ST) parameter.

ture against the split threshold for various values of split granularity. As expected, the number of buckets reduces as the split threshold is increased and the average depth of the tree is also decreased. The number of buckets and the average depth of the words is plotted against split granularity in Figure 5. Increasing the granularity increases the number buckets while the average depth of the Trie decreases. At higher granularity, a bucket is split to more number of children than at a lower granularity. Hence the number of buckets increases and the necessity for further splits decreases leading to a shorter Trie. These simulations clearly show that a higher split threshold and larger split granularity factors decrease the average depth.

We observe that the number of buckets is very large even at the small split granularity values. For example, at split granularity of 1 and a threshold of 250, we expect around 150000/250 = 600 buckets while the observed valued is about 1500 buckets. The large number of buckets is due to the fact that many buckets with few entries are created during the Trie construction. For example, with 26-way splitting or split granularity of 5 as shown in Figure 3.1, we split the bucket at root into 26 buckets with 4 non-zero size buckets (We do not count the zero-sized buckets in our bucket count). While a smaller split granularity reduces the buckets with fewer entries, we still observe that there are still a lot of buckets with small number of entries.

We propose a B-tree flavored Trie based approach that tries to reduce the number of buckets. Upon the need to split a bucket as its size goes beyond split threshold, instead of splitting it into k-ways as specified by the split granularity, delegate down only for few ranges. This idea is explained with the illustration in the Figure 5. Instead of splitting 26-way as shown in Figure 3.1, this controlled delegation reduces the number of buckets. Notice that this splitting has a flavor of B-tree where the intermediate nodes in the tree maintain some entries. We propose to investigate the effectiveness of this approach as part of the future work.

We measure the storage load-balancing property of a scheme by measuring the normalized standard deviation of the storage loads across all nodes. Figure 5 compares the load-balancing properties of basic Squid approach with Trie based scheme for (ST=250, SG=5) values in case of Trie based approach. In basic Squid, the keywords are mapped to a linear space from 0 to 1 in an order-preserving manner and the nodes are also uniformly mapped to

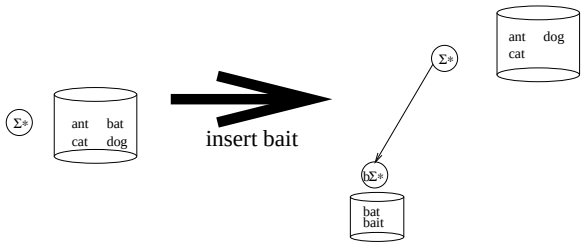


Figure 10: An example Trie construction with B-Tree flavor.

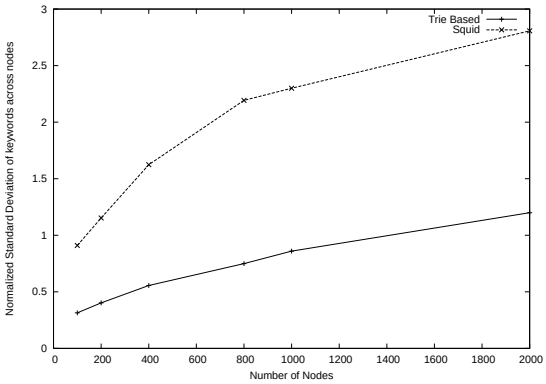


Figure 11: Normalized standard deviation of the storage load on the nodes with basic Squid (without load balancing schemes) and Trie-based approach

the same space. A keyword is assigned to a closest node with larger ID. In these set of simulations, we do not consider the run-time load-balancing techniques proposed by the authors of the Squid system. Clearly the Trie based approach is better in spreading the storage load than basic Squid approach.

We plot normalized standard deviation for Trie-based approach with split threshold for various values of split granularity in Figure 5. With increasing split threshold, the imbalance in the storage load across nodes increases. Smaller thresholds increase the load balance because the fine granularity at which the keywords can be spread across the machines. At split threshold of just 1, the approach is same as the standard DHT where each keyword is hashed separately. We also plot the normalized standard deviation with split granularity for various values of split threshold in Figure 5. Higher values of the split granularity spread the load more evenly than for the lower values of the split granularity.

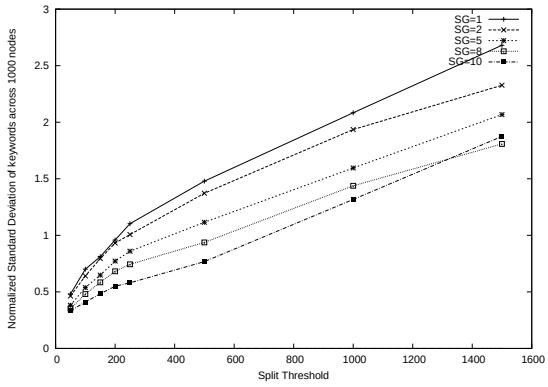


Figure 12: Normalized standard deviation of the storage load on the nodes for Trie-based approach with Split Threshold (ST) for various values of Split Granularity(SG).

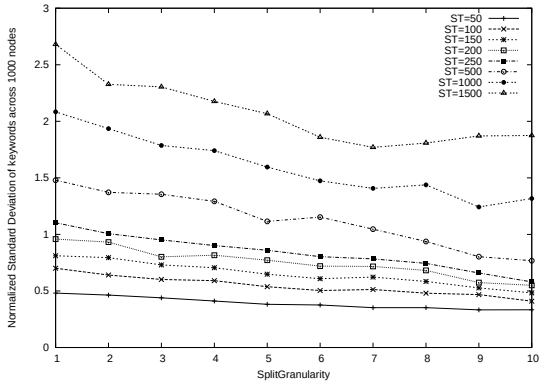


Figure 13: Normalized standard deviation of the storage load on the nodes for Trie-based approach with Split Granularity(ST) for various values of Split Threshold(ST).

6 Conclusions and Future Work

In this project, we propose a distributed Trie-like data structure for storing the keyword and documents so that (a) range queries can be supported, (b) efficient insertions, deletions and lookups are supported, (c) storage load is uniformly distributed across the participating node, and (d) access load is also uniformly distributed.

The preliminary simulation results show that the Trie-based structure is more effective at distributing the load across than the nodes. One interesting point we observe is the creation of large number of buckets with fewer entries at higher split granularities. We propose a modification to the Trie-based

data structure with the flavor of B-trees that reduces the creation of smaller sized buckets.

Some words are more common than others and hence more documents matched with those words than others. As future work, we plan to query google with each word to estimate the number of documents matching a particular keyword and use those numbers as part of our workload.

We plan to implement our algorithm on top of Pastry [4], a free DHT system and evaluate the efficacy of the algorithm with respect to the following metrics: (1) storage load balance, (2) access load balance, and (3) insertion, deletion, and lookup times and message costs (with and without caching enabled). We will also study the behavior of the algorithm under fail-stop fault model for varying number of faults.

References

[1] J. Aspnes and G. Shah. Skip Graphs. In *Proceedings of the 14th ACM-SIAM Symposium on Discrete Algorithms*, January 2003.

[2] R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong. Extendible hashing - a fast access method for dynamic files. *ACM Transactions on Database Systems*, 4(3):315–344, September 1979.

[3] C. G. Plaxton, R. Rajaraman, and A. W. Richa. Accessing Nearby Copies of Replicated Objects in a Distributed Environment. In *ACM Symposium on Parallel Algorithms and Architectures*, 1997.

[4] A. Rowstron and P. Druschel. Pastry: Scalable, Distributed Object Location and Routing for Large-scale Peer-to-peer Systems. In *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms(Middleware)*, November 2001.

[5] C. Schmidt and M. Parashar. Flexible Information Discovery in Decentralized Distributed Systems. In *Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing*, 2003.