Efficient Processing of Range Queries over Distributed Relational Databases

Richard Price
Information Technology
Georgia Gwinnett College
Lawrenceville, GA, USA
rprice@ggc.edu

Lakshmish Ramaswamy
Computer Science
University of Georgia
Athens, GA, USA
laks@cs.uga.edu

Seyedamin Pouriyeh
Computer Science
University of Georgia
Athens, GA, USA
pouriyeh@uga.edu

Abstract

There have been several solutions proposed for solving the distributed range query problem. These involve specialized data structures that map the location of the various data elements. In this paper, we propose a solution that solves the range query problem without the use of any auxiliary data structure.

Keywords: Distributed Databases; Distributed Systems; Range Queries

1 Introduction

With today's large datasets, distributed databases have become a necessity for query processing. As with all distributed systems, failure resilience is a major concern. Systems have been developed to overcome distributed failures such as Consistent Hashing [1] and Chord [2]. While these systems address node failures, they do not provide robust query generation for range queries.

Our research addresses the problem of querying with a range of keys rather than a single key point query. Range queries are an important aspect of data queries where a list of items needs to be generated. This list could be something as simple as a list of words beginning with the same letter or something as complex as list of products within a range of product ids. Regardless of the domain, we routinely need to retrieve all items for this search.

Using ranges of keys introduces several complexities. First, all possible keys in the range must be generated. Since we do not know all of the key values stored, we must determine a method that will allow us to build this set of keys. If data is located on a single server or a system that utilizes proprietary technology that allows us to track items that have been stored, this simply becomes a matter of leveraging the built-in mechanisms such as a between clause in SQL. For a distributed system using a hashing algorithm to place data on remote servers, there is no mechanism in place to track what data has been stored. Therefore, we must generate every possible key to allow us to successfully retrieve data from these different servers. For short keys, this is not an expensive process and can be done quickly. However, for large key sizes, key generation takes a prohibitively large time to complete.

A further complication arises when the query is attempted. Sending the query to each server to retrieve possible data requires a query for each key generated. This results in unacceptable performance for queries with a large number of possible keys

The contribution of this paper is to provide a solution for overcoming these two major problem areas and provide a robust system that supports range queries using distributed relational databases without the need for special mapping tables or any other method of tracking where data is located. While this has been accomplished for point queries, retrieving a single item from a distributed system, to our knowledge, this has not been accomplished for range queries. Our implementation using distributed relational databases does not require any of the remote databases to have any knowledge of each other nor are they constrained by the necessity to communicate between servers. This system relies on a master server and any number of independent database servers. The master server maintains a list

978-1-5386-2659-7/18/$31.00 ©2018 IEEE
DOI 10.1109/IRI.2018.00057

of remote databases and controls the routing of queries to these servers on an as needed basis.

2 Background and Related Work

Several systems have been proposed for implementing range queries using data structures to handle the range query. A range query allows a user to retrieve information between two given items. For example, a crossword enthusiast might be stumped and need a list of all of the words from a dictionary that begins with "ca". This could be done by executing a search on the dictionary requesting all words that begin with "ca" up to but not including words beginning with "cb". This retrieval would then allow the user to select a word to complete the puzzle they are solving. To create the query, the system would need to retrieve all words that begin with "ca" such as "caa", "cab", "cac", etc. Since we don't know each of the words that would met this criteria, we would need to generate each possible combination of letters and see if it exists.

When performing a range query on a single system, this is easily done in a relational database. The challenge we encountered is that we have no single source of all of the words. These words can be on any of the servers in our network. In our experiments, we used a hashing algorithm to distributed words across our servers and once sent to the server, the master node immediately forgot all knowledge of where the word was located. Other solutions that have been proposed to solve this problem have relied on data structures that maintain the location of this data. Some even proposed distributing like objects, those that begin or end with similar patterns, to specific nodes to eliminate the need to search across multiple servers to find the complete list of items. Our solution avoids bottlenecks and overloaded nodes that may result from this type of solution.

Ratnasamy et al Range Queries over DHT's [3] proposed the use of Prefix Hash Trees coupled with Distributed Hash Tables to form a trie-based scheme. In this paper, Ratnasamy et al, proposed storing the trie buckets in the DHT to allow for more efficient processing to find the corresponding prefixes. They developed the idea of placing the prefix hash tree in different buckets in the DHT. Comparisons were then done with generated keys to find the longest common prefix to use in the query. They proposed this could be parallelized by dividing a range into sub-ranges.

Yalagandula in Solving Range Queries in a Distributed System [4] referenced Skip Graphs, Aspnes et al [5] and Squid as proposed by Schmidt et al [6]. Skip Graphs rely on membership vectors as shown in Figure 1 which are tracked in a DHT. This DHT is built on the physical nodes. Each node has to maintain pointers to other nodes. Squid uses an n-dimensional data mapping to a 1-dimensional space. Squid is based on a data-lookup system that essentially implements an internet-scale distributed hash table. This system utilizes keywords to hold mapping, for example, numbers could be mapped in ranges of values and English words based on their first letter. This scheme is able to perform efficient range queries but the distributed loading becomes skewed.

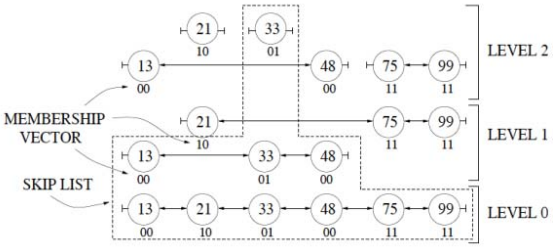

Figure 1: A skip graph with [logN] = 3 levels

In Yalagandula's approach, the alphabet is limited to one-dimensional keywords to allow for efficient distribution and lookup. The alphabet is limited and hashing is based on an ID which will correspond to the node storing the data element. In this approach, Yalagandula created tries with hash tables to map locations for queries. This is shown in Figure 2 for a tree of lower case letters.

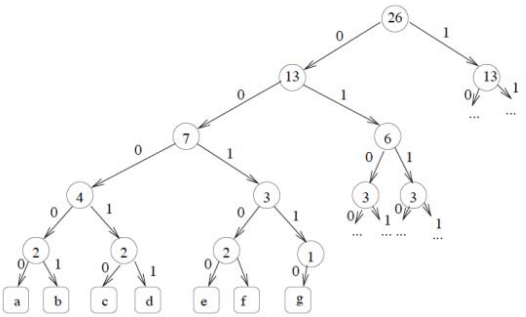

Figure 2: Encoding tree for representing 26 lower case alphabets

Once the Trie structure is created, buckets to map the IDs to a node. When attempting to create fine resolution, this results in a very large number of buckets that need to be defined and maintained.

Our motivation for seeking a different range query approach was to remove the necessity of maintaining data on the master node and allow flexibility in storage and retrieval. Our system is intended to be able to store any type of data that can be identified with a hashable ID. In our approach we differ from previous solutions in that no restrictions are applied to the keys or the alphabets used to hash and distribute the data. There is no need for any of the database nodes to know about other nodes. The master node must know about each of the slave nodes but we believe this can be handled with most distributed protocols.

3 Experiment Design

Our approach was to implement a distributed database using readily available technology, SQLite.

Our test was to process a standard English dictionary and store each word and its definition based on the hash of the word. We used varying key lengths and only used these keys to map the data. Once a hash was generated, this was used to map to the available servers.

To generate the list of keys, we employed a recursive, looping algorithm. This algorithm took the initial word used for the query and then added to it the starting word, all of the characters necessary to build all of the keys. For example, if we are using a key length of 2 characters, and we want to query for all "words" between M and O, we would need to build keys for MA, MB, MC, …, MZ, N, NA, NB, …, O. Figure 3 shows the steps required for the key generation.

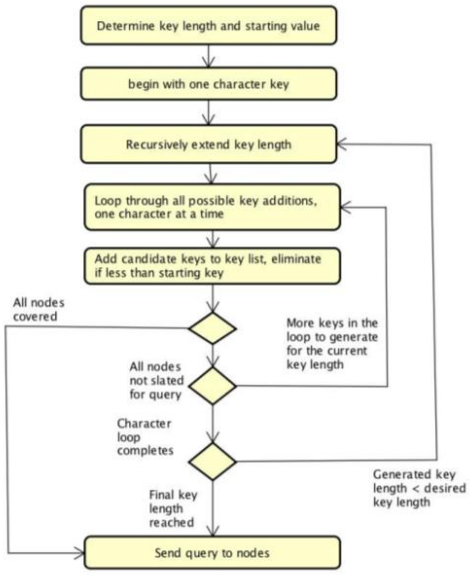

Figure 3: Key generation activity flow

Our initial attempt to solve this problem is illustrated in Figure 4.

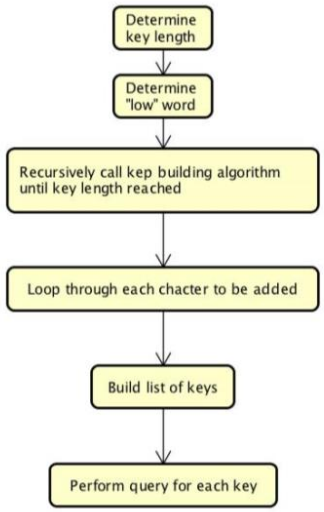

Figure 4: Initial program flow

In our initial experiments, we generated an exhaustive list of keys and sent individual queries to the servers containing the data. This proved to cause unacceptable delays for the retrieval of information due to the large number of queries being transmitted. After our initial results, we modified our retrieval process to send a single query to each server that contained the data. This allowed us to retrieve all of

the data from each server with only one query. It is important to note, servers that did not contain data were never queried. This optimization resulted in a dramatic reduction for retrieval, cutting retrieval times for "M" to "O" using 4 letter keys from 2,747,945 milliseconds to 235 milliseconds. Based on these results, we began examining other possible optimizations. Figure 5 shows the revised program flow for this communication improvement.

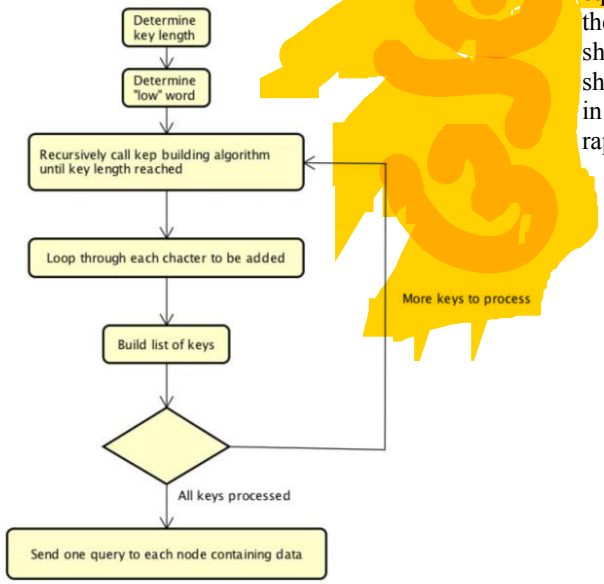

Figure 5: Query optimized progam flow

When keys grew beyond 5 characters, the generation of all possible keys became another bottleneck. This was the final major bottleneck we encountered. Assuming the key set is limited to the letters from the exclamation point(!) to the tilde symbol (~), for every position in the key you must traverse for each of the subsequent possible key additions, 95 keys are generated at the first level, this quickly becomes an extremely large number of keys. For example, with a key length of 6, to generate for all of the keys for "A" would be 1 for the first letter, 95 at the second level. At the third key, it would be 1 + 95 + 95 squared. At the fourth level, it would be 1 + 95 + 95 squared + 95 cubed. Roughly, we have $95^{key\ length-1}$ keys. For a single letter with a key length of 10, this generates $6.302\ x\ 10^{17}$ keys. The generation of these keys resulted in unacceptable performance.

4 Experimental Breakthrough

When considering these issues, we discovered that tracking the servers needed was the optimal solution. Without this optimization long keys require extremely intense calculations with unacceptable performance. By tracking the number of nodes the generated keys cover, the key generation can be done very rapidly. Once the number of servers to query equals the number of servers containing the data, there is no reason to continue generating keys. As shown in Figure 6, we allowed the key generation to short circuit and exit when all possible servers were in the query queue. The key generation algorithm rapidly exits and return the proper results.

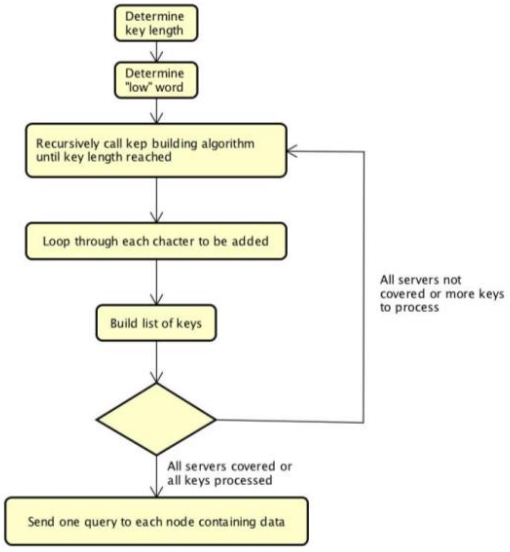

Figure 6: Final range query program flow

5 Experimental Results

In our experiment, we used 30 Ubuntu 14 virtual machines with 1 GB of Ram and a single CPU. These were hosted in on servers in our networking laboratory and resided on multiple racks. The master node was a Macintosh 15-inch laptop with 16 GB of RAM, a 500GB SSD and quad core I-7 2.5 Ghz processors. We emulated 100,000 nodes and then mapped the queries to these 30 remote devices. Communication was done over the internet with an average ping time of 20 ms. Timing was performed using the milliseconds of the master clock and was reported from the point where the query began, including key generation, communication to each

node and the processing/compilation of the results from each query.

Not surprisingly, all of our implementations worked efficiently on short keys, less than 4 characters. A 1-character key resulted in even loading across 26 servers, data used is an English dictionary, of the servers but completely ignored the other servers. A 2 or 3-character key resulted in balanced loading across servers and reasonable performance.

To solve the problem of which servers to query using a range query, an exhaustive list of keys needs to be generated. This required a complex recursive algorithm to generate each possible key. This algorithm combines a recursive component to build the keys to the length required and an internal loop to add every possible set of keys. While our implementation efficiently generated all keys for queries using keys less than 6 characters in length, when generating a key 8 characters long or greater required extensive computation and unacceptable performance delays in the query. If a starting or ending filter was shorter than the key length, the filter was extended so that it was at least as long as the key length specified. For the starting filter, a ! character was added until the filter was the correct length. For the end filter, a ~ was added until the key length was achieved.

An additional problem was the number of queries sent to the servers. Originally, we were sending a query for each key generated. This flooded the network with queries and resulted in extremely poor performance.

Figure 7 shows the time required to send each of these queries and retrieve and collate the retrieved data. Due to the extreme times required to execute queries using the original plan, only key lengths or 1 to 4 characters were used.

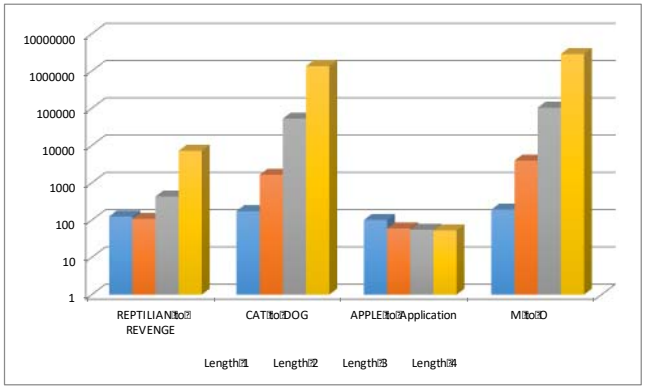

Figure 7: Time to retrieve and collate data based on key length in ms (logarithmic scale)

This led us to two changes in our process. First, we cached the queries and only sent a single query to each server. This greatly reduced network traffic. For example, this change reduced the number of queries for a key length of 4 from 52,728 queries for the filters "M" to "O" to no more than the number of servers. The second change was to short-circuit the key generation and only build keys until either the number of servers was reached or all the keys for a given filter set were generated. Even when the number of servers was increased dramatically, the key building remained efficient. In our experiments, generating enough keys to hash to 100,000 servers required less than 500 ms.

Figure 8 shows the key generation times based on our revised algorithm.

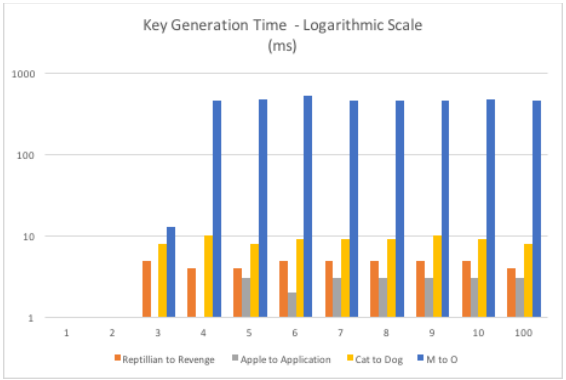

Figure 8: Key generation times in ms (logarithmic scale)

After implementing these changes, execution times were greatly improved.

Our experiments were performed using 30 database servers with distributions of keys over 100,000

virtual servers. Each of these servers were remotely located and accessed via the internet. Figure 9 shows the times for retrieval for varying key lengths.

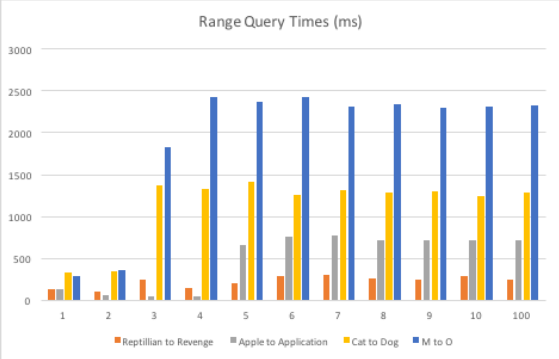

Figure 9: Range query times in ms

By tracking the number of servers that the algorithm returned, we can see that this number stabilizes when the word is shorter than the key length. Figure 10 shows the number of servers the algorithm returned. Even when returning 100,000 servers, the algorithm performed in less than 500 milliseconds. By using 100,000 servers, we believe we have modeled an extreme case to test our short-circuiting theory. Based on these results, we believe tracking the servers and stopping key generation when further generation cannot add additional servers is an effective way to generate keys for range queries.

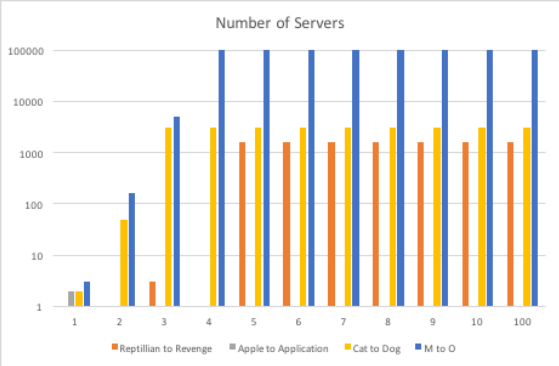

Figure 10: Number of servers generated using different key lengths (logarithmic scale)

As we increased the size of the key, we saw a dramatic decrease in words returned. This was the result we expected. By increasing the "filter" our return data set became more precise. Figure 11 shows the result of the queries with different key lengths.

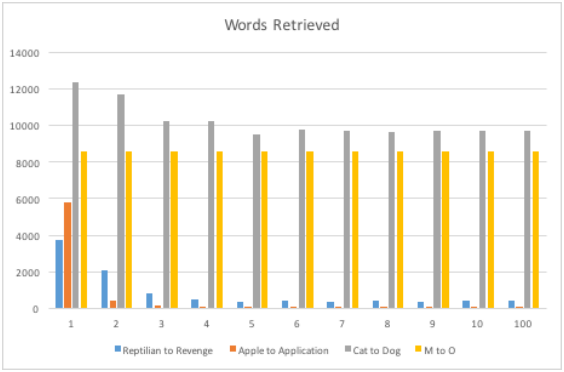

Figure 11: Number of words retrieve using different key lengths

6 Conclusions

Range querying can be efficiently performed. An exhaustive set of keys must be generated and hashed to determine the correct servers to query. For longer keys which result in more accurate results, this can be computationally prohibitive in terms of time and performance. Likewise, sending a single query to each server can be done but again this quickly becomes non-performant. By short-circuiting the key generation once all of the servers to be queried are included and sending a single query to each effective server, performance can be optimized. This method of distributed querying should be adaptable to any data that has hashable keys.

Bibliography

[1] D. Karger, E. Lehman, T. Leighton, R. Panigraphy, M. Levine and D. Lewind, "Consistet hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web.," *In Proceedings of the twnty-ninth annual ACM symposium on Theory of computing.,* pp. 654-663, 1997.

[2] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek and H. Balakrishnam, "Chord: A Scalable Peer-to-Peer Lookup Protocol for Internet Applications," *IEEE/ACM Transactions on Networking,* vol. 11, no. 1, pp. 17-32, 2003.

[3] S. Ratnasamy, J. M. Hellerstein and S. Shenker, "Range Queries over DHTs," *IRB-TR-03-009,* vol. 03, no. 009, pp. 1-2, June 2003.

[4] P. Yalagandula, "Solving Range Queries in a Distributed System," *Tech. Rep. ,* no. TR-04-18, pp. 1-7, 2004.

[5] J. Aspnes and G. Shah, "Skip Graphs," *Proceedings of the 15th ACM-SIAM Symposium on Discrete*

Algorithms, vol. 14, no. SODA '03, pp. 384-393, January 2003.

[6] C. Schmidt and M. Parashar, "Flexible Information DIscovery in Decentralized Distributed Systems," *Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing,* vol. HPDC.2003, no. 1210032, pp. 226-233, June 2003.