CS618: Group 08
Efficient key based distribution technique for a distributed B+ Tree

Proneet Verma
Student ID 44
Roll Number 12520
proneetv@iitk.ac.in
Dept. of MSE

Saurav Kumar
Student ID 45
Roll Number 12641
ksaurav@iitk.ac.in
Dept. of CSE

Indian Institute of Technology, Kanpur

Final Report
August 13, 2015

Abstract

B+ Trees are efficient and effective in indexing data stored in databases. This project aims to efficiently implement and analyze performance of general queries in B+ Trees distributed over a multi-node computer network. We have considered several factors for *data distribution* among servers, such as geographical location of the node, bandwidth of network connecting the root and the node, hardware configuration of the node, and *likelihood of data* being queried in order to arrive at a comprehensive analysis.

1 Introduction

The problem consists of 2 parts:

- Implementation of distributed database which uses B+ Tree to efficiently index the database.
- To devise an efficient distribution of nodes and analyze its performance on general queries.

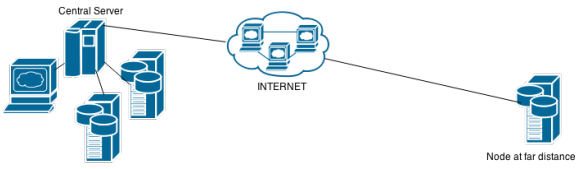

Figure 1: Distributed Database

Score based system: For this problem, we will use a score-based technique to distribute data among the nodes. Servers will have their scores computed based on their average response time which depends on network response time, processing speed and disk I/O time. This score will roughly determine how much data should be kept on which servers.

Likelihood: Since some data (keys) may be returned more often than others depending on distribution of data and query responses, we will try to find an optimum strategy to keep more frequent data in servers with better score. To test the effectiveness of the strategy we will train the model with some distribution of queries and test with same distribution of query data. For actual practise, we propose online migration of nodes between servers based on query and response analytics

2 Implementation

2.1 Building and Distribution of the B+ Tree

To build the distributed B+ tree, we took the approach of dividing the whole cluster into two kinds of servers:

- Central Server
- Data Servers

The *Central Server* specifically has four tasks:

- **Bookkeeping jobs**: Basically keeps a track of root, and on to which server it is placed on. Maintains a frequency count of internal and leaf nodes.

- **Determines key distribution**: Determines the best server to keep a node whose indicative key is passed to it.
- **Query response analysis**: This analysis tool is broken down to two modes, offline as well as online, to train the learning model. In the offline mode, the analysis tool iterates through the whole query file to determine the probability of occurrence of query terms. In the online mode, this would keep the scores updated with time. Apart from computing a term based probability distribution, this also does a routine check on the data servers to determine which servers are good and which are bad. Here, by good we mean that their ping time is less (or more respectively). This also keeps account of which query takes how much time for further analysis of the approach.
- **Interface between index structure and user**: This acts like the face to our whole indexing architecture. The user with the help of this interface is able to query through our indexing schema.

Each server has its own server ID, whose IP address and ports are known to the system. Node pointers are pair of *<serverID, fileName>*. Each Data Server in itself acts like a standard B+ Tree and has all its properties.

2.2 Algorithm for querying:

- User queries the central server
- Central server routes the query to appropriate data server which can return faster response to the query.
- Data server listens to the requests and creates a new thread to handle the query and makes appropriate networks calls, if need be, and computes the complete result and returns the response
- Each data server has its own instance of B Plus Tree which enables these servers to perform all B Plus Tree operations. The result is then returned to caller recursively, finally to the central server which is then analyzed by the central server and passed to user.

Each request made to any server returns the response after completion, thereby closing the thread pool sequentially.

2.3 Scoring

Scoring is being done for two entities in our structure:

- **Server scoring**: Assign scores to the servers based on network response time and hardware configurations of data servers
- **Likelihood**: Assign scores to the keys based on their probability of being queried

Based on the above two scores, we decide which data-server to place this key on. Both the scores are scaled to bring in [0, 1] range.

When to decide?
Nodes are created when some other node splits. We use the first key in the node as the indicative key to determine which data server this new node should reside on.

How to decide?

- Randomly
- Equal segmentation of key ranges and randomly assign each sub-range to a data server
- Using key and server scores to compute a mutual score

Our Scoring Technique:
Now we have both the server score as well as the key scores, and we need a function which can help us calculate a mutual relationship score which will indicate which key should reside on which type of servers.

We need a **Mutual Scoring Function** with the following properties:

- Takes in two arguments both between 0 and 1
- Returns a high value when key score and server score are both high (or both low)
- Returns a low value when key score and server score are of opposite nature

For x being the server score and y being the key score, one such function is:

$$f(x, y) = (x - 0.5) \times (y - 0.5) \quad (1)$$

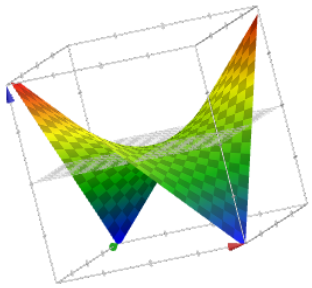

Figure 2: Graph of first function (1)

Demerit of the above proposed function is that it works good only for two data server system since it considers only extreme ends of score ranges. This will lead to overloading on the best and the worst server in case of multiple server distributed architecture. (Ref: Figure 4)

Second choice of function is the following:

$$f(x, y) = 1 - abs(xy) \tag{2}$$

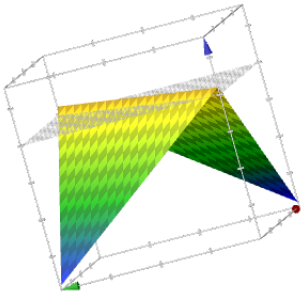

Figure 3: Graph of second function (2)

Merit of above proposed function is that it takes into consideration scores of intermediate keys and intermediate servers. The idea is to give a high mutual score to scores of similar nature.

It very simple and efficient to compute the mutual scores and this works very satisfactorily as evident from the key distribution across. (Ref: Figure 5)

Henceforth, we now have a mutual scoring function which will be used to fuel our efficient key based distribution technique for our distributed B+ Tree.

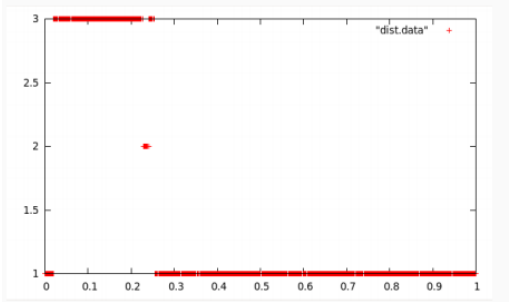

Figure 4: Key distribution across servers with the first function (1)

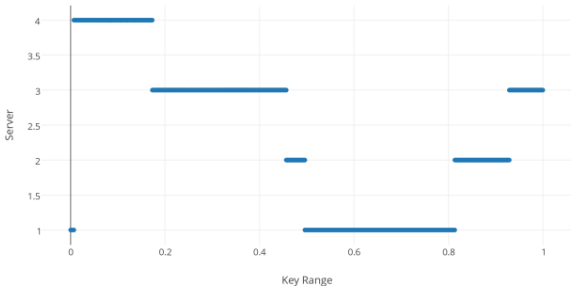

Figure 5: Key distribution across servers with the second function (2)

3 Analysis

We will now outlay how we went upon analyzing our distributed index structure. For that firstly, we built a data set of 5000 points selected uniformly randomly from [0, 1] range and 500 queries comprising of Insertion, Point Query, Range Query, Window Query and kNN Query.

We are analyzing the query times of these queries, and also keeping a count of total network based calls and total running time. We have used 4 servers for our analysis, whose scores we determined based on their respective ping times over the network, as 0.9, 0.7, 0.4 and 0.1 (higher is better).

After the query generation, in the offline mode, we analyzed the query data to determine the probabilistic key scores based on the frequency of a key being returned in some query.

We are comparing these metrics over 3 different key distribution strategies:

- **Random**: randomly assign a node to a server
- **Equal Segmentation**: Partition the whole key space in equal chunks and randomly distribute each chunk to a server. For comparison, we have placed the chunk consisting of the most search keys on the best server and so on
- **Our technique**: Based on the mutual score determined, the key is placed on a specific server

We then analyzed our index structure for the two query sets whose frequency distribution is shown below.

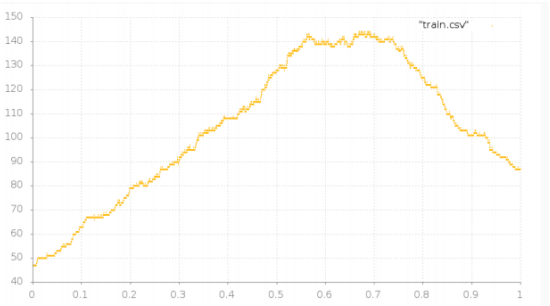

Figure 6: Query distribution(1): frequency of occurrence of keys

For the above query distribution, following values were observed:

	Random	Segmentation	Scoring
Total Running Time (s)	2607	940	680
Central Server calls	11165	11196	11196
Data Servers calls	21321	4738	5650
Insertion (s)	0.304	0.150	0.109
Point Query (s)	0.297	0.138	0.101
Range Query (s)	8.785	0.434	0.435
KNN Query (s)	0.728	0.152	0.117
Window Query (s)	4.051	0.211	0.204

Table 1: Comparison of metrics for Query Distribution(1)

For another query distribution:

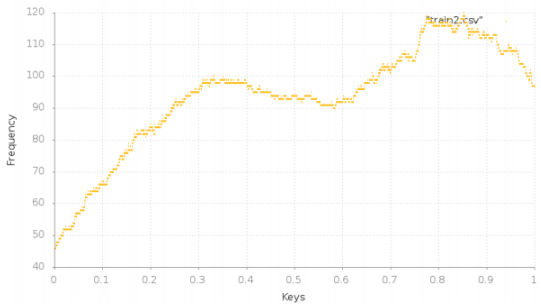

Figure 7: Query distribution(2): frequency of occurrence of keys

For this query distribution, following values were observed:

	Random	Segmentation	Scoring
Total Running Time (s)	1796	732	480
Central Server calls	11196	11196	11196
Data Servers calls	4738	4804	3469
Insertion (s)	0.178	0.141	0.098
Point Query (s)	0.163	0.142	0.095
Range Query (s)	1.627	0.389	0.332
KNN Query (s)	5.528	0.157	0.133
Window Query (s)	1.355	0.230	0.163

Table 2: Comparison of metrics for Query Distribution(2)

4 Conclusions

- Our scoring technique outperformed the other two trivial approaches by a considerable margin.
- For queries which are more popular, there has been nearly about 40% reduction in their look up time.
- With the analysis that we have put forward, we can infer that this indexing approach can be widely used for faster look up of globally important or popular terms.

References

1. Incrementally distributed B+ trees: approaches and challenges, Pallavi Tadepalli, H. Conrad Cunningham *http://dl.acm.org/citation.cfm?id=1566451*

Repository

Github: `https://github.com/proneetv/turingdb`