TEMPLATE B+ TREES : AN INDEX SCHEME FOR FAST DATA STREAMS WITH DISTRIBUTED APPEND-ONLY STORES

BY

PARIJAT MAZUMDAR

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2016

Urbana, Illinois

Advisor:

Professor Emerita Marianne Winslett

Abstract

Distributed systems are now commonly used to manage massive data flooding from the physical world, such as user-generated content from online social media and communication records from mobile phones. The new generation of distributed data management systems, such as HBase, Cassandra and Riak, are designed to perform queries and tuple insertions only. Other database operations such as deletions and updates are simulated by appending the keys associated with the target tuples to operation logs. Such an append-only store architecture maximizes the processing throughput on incoming data, but potentially incurs higher costs during query processing, because additional computation is needed to generate consistent snapshots of the database. Indexing is the key to enable efficient query processing by fast data retrieval and aggregation under such a system architecture.

This thesis presents a new in-memory indexing scheme for distributed append-only stores. Our new scheme utilizes traditional index structures based on B+ trees and their variants to create an efficient in-memory template-based tree without the overhead of expensive node splits. We also propose the use of optimized domain partitioning and multi-thread insertion techniques to exploit the advantages of the template B+ tree structure. Our empirical evaluations show that insertion throughput is five times higher with template B+ trees than with HBase, on a variety of real and synthetic workloads.

To my parents, for their love and support.

Acknowledgments

I would like to thank my advisor, Prof Marianne Winslett, for motivating me and providing me with valuable support and insights throughout the duration of this work. I would also like to thank Zhenjie Zhang (Advanced Digital Sciences Center, Singapore) for his support, guidance, advice, valuable comments, suggestions, and provisions which have helped me immensely in completing this thesis. Zhenjie has played a pivotal role in all stages of this work by brainstorming ideas, sharing knowledge and solving the problems that arose. I am very thankful for having a good mentor like him. I also want to thank Li Wang (Guangdong University of Technology) for his valuable inputs in developing the indexing scheme and designing the necessary experiments.

Table of Contents

List of Figures

Chapter 1

Introduction

Massive stream data, such as user-generated content from online social media [17] and communication records from mobile phones [11], now flood into database systems at an extremely fast rate. Huge demands for efficient processing and management of the streaming data have driven the growth of interest from both academia and industry in new solutions based on distributed systems. A handful of open source systems, such as BigTable [5], HBase [1], Spark Streaming [3] and Storm [2], are now commonly used to process, analyze and store the data, by using a cluster of commodity PCs or the computation resources on cloud platforms.

One common strategy used in existing distributed systems is the adoption of lightweight protocols for processing incoming data, in order to maximize the processing throughput. Basically, all data-related operations are transformed into tuple insertions. Such an append-only store infrastructure provides more opportunities to scale up the processing throughput at the gate of the distributed system. However, such an architecture brings additional overhead for data retrieval and querying.

Indexing is the natural option for solving the data retrieval and querying difficulty, by reducing the computation efforts needed to locate the target data of particular queries and analytics. Conventional B+ tree structures and their variants [8, 9] are designed to minimize the number of I/Os incurred by the index manipulations, and the overhead of node splits becomes dominant when the insertion rate grows. Even parallel insertions into the index cannot help, because potential contention at the intermediate nodes in the tree structure limits the scale-up effect of concurrent writing. The LSM-tree [12, 14] is the de-facto solution used in real systems (e.g., HBase), which maintains a small buffer in memory for indexing

recent updates and merges the index with historical records on disk when the buffer is full. While an LSM-tree improves the efficiency when it is used at a single node, it is tricky when multiple nodes in the distributed system are processing updates at the same time. In HBase, the workload is partitioned based on a fixed scheme and each partition is handled by an individual *region server*. This leads to difficulties when workload migration is triggered by workload imbalance or workload growth. We believe a desirable solution to the indexing problem for distributed append-only stores must meet the following requirements, including 1) the insertion rate at a single node is high enough to maximize the utilization of CPU resources; 2) the performance of the index structure is good enough to support a wide variety of retrieval and query tasks over the data.

In this thesis, we present a new index scheme to take advantage of the excellent retrieval performance of B+ trees, without the high overhead of traditional tree structure construction. Our new scheme is motivated by observations of fast data streams, whose content usually follows a relatively stable distribution [15]. It is therefore possible to maintain a template of a B+ tree structure and reuse the template for newly arriving tuples. Instead of building a new index for a group of tuples from scratch, our scheme allows the system to locate the leaf page for a new tuple and directly insert the tuple without revision of the intermediate nodes in the B+ tree. To fully exploit the advantage of the template, we propose two additional optimization techniques, a multi-thread parallel insertion approach to maximize the insertion throughput and an overflow management approach to handle minor distribution shifts.

The main contributions of this thesis are summarized below:

1. We present a new index scheme to support lightweight tuple insertion in distributed append-only stores. It uses a template B+ tree structure to facilitate efficient tuple insertion without expensive node split operations on the tree structure.

2. We discuss optimization techniques to further enhance the efficiency of template B+

trees by supporting parallel insertion and overflow management.

3. We evaluate our proposal on two real and one synthetic workloads and find that template B+ trees with concurrent insertions achieve almost eleven times lower latency and five times higher insertion throughput than Apache HBase (version 1.1.3) [1], an open-source implementation of HBase.

Chapter 2

Preliminaries

2.1 Data and Query Model

In our system model, the incoming data stream consists of a possibly infinite sequence of key-value pairs. Each key-value pair, (k, v), contains a key k from a fixed domain $\mathcal{D}$ and the corresponding payload value v. We assume that $\mathcal{D}$ is a totally ordered domain (e.g. numerical and integer domains). For other domains, e.g. strings of varying length, it is possible to apply mappings to transfer the keys from a more complex domain to a totally ordered domain, e.g. a hashing-based mapping on strings [16].

As shown at the bottom of Figure 2.1, the incoming tuples and the index structures over the tuples are eventually dumped to a distributed file system, e.g. HDFS [13], for persistent storage. The basic storage unit of data in a distributed file system is a *file*. To maximize the processing efficiency, the size of the files is expected to align with the block size specific to that distributed file system. In HDFS, for example, the block size is 64 MB. Different from traditional schemes, our system only builds a *local* index whose scope is the tuples in one single file. One copy of the index is merged into the file and written to the distributed file system.

Based on the concepts above, we formulate the problem we address in this thesis as follows. Given the stream data flooding into the system, we aim to design a new index scheme, which enables efficient index building for each set of tuples in a file. The index must support both point search and range queries. Given a value $x \in \mathcal{D}$, point search query $q(x)$ returns all tuples with $k = x$ in the append-only store. Similarly, given a range $R \subset \mathcal{D}$,

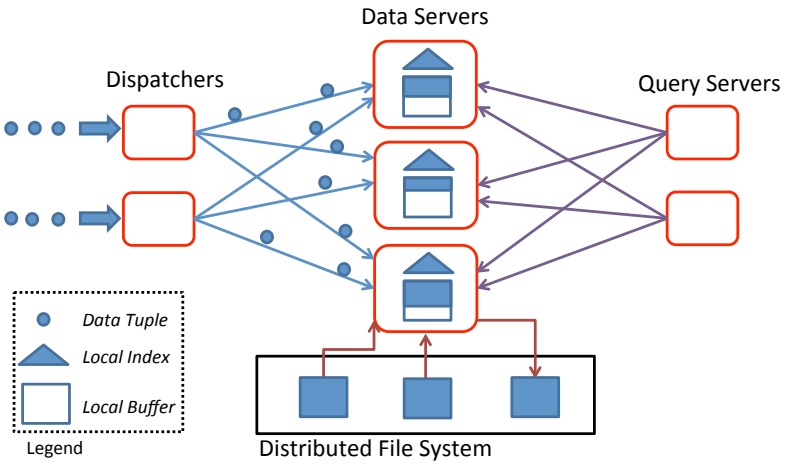

Figure 2.1: The overall architecture of our system

range query $q(R)$ returns all tuples with $k \in R$.

2.2 Overview of System Architecture

Figure 2.1 shows the architecture we assume for a tuple stream processing system that uses template B+ trees. In the figure, each red box is a running thread in the distributed system. There are three different types of running threads, *Dispatchers*, *Data Servers* and *Query Servers*. Dispatchers are responsible for pre-processing on the incoming tuples. They also assign the tuples to one of the data servers for further processing. In our implementation of the system, the data server allocation strategy is round robin, which roughly balances the workloads of the data servers. A data server maintains a local index and a local buffer for the received tuples. Generally speaking, the data server appends each new tuple in the buffer and inserts the offset address of the tuple into the local index based on the key of the tuple. Once the buffer is full, the data server dumps the buffer as well as its corresponding index

into the distributed file system, as a unified file. When a query arrives at the system, it is processed by a query server. The query server pushes data requests to all the data servers. Each data server retrieves the relevant data from all files it has dumped to the distributed file system and returns all result tuples to the query server.

The performance of the system depends on the effectiveness and efficiency of the index structure. A poorly performing index scheme could slow down the rate of tuple insertion and increase the query processing latency. Interestingly, one of the implicit advantages of this system architecture is the possibility of more efficient index structures, based on our observation of workloads from real world applications. The distribution of the key values of tuples usually does not vary dramatically over time. It is thus unnecessary to rebuild the index from scratch for every buffer of stream tuples. Instead, when a buffer at a data server fills up, the structure of the index (such as the intermediate nodes of a well-optimized B+ tree) can be directly reused for the next buffer of tuples. Thus, in each data server, initially we grow a traditional B+ tree from scratch to serve as the local index. Once this initial B+ tree is full, we flush the contents of the leaf nodes of the B+ tree and use the its internal nodes as our template B+ tree.

The overall architecture of HBase is similar to the presented architecture, but some differences exist. HBase uses a single dispatcher server called the *HMaster* instead of multiple dispatcher servers. The data servers are called the *region servers* and the local buffer of an individual data server is called the *memstore*. But, in HBase, each data server has multiple local indices (instead of a single local index), called *regions*. These regions provide parallelism in data servers because each region independently manages a partition of the entire domain of key values, i.e. all tuple insertions having key values in the appropriate domain partition are indexed in this region. When the *memstore* (or local buffer) of a data server fills up, its contents are flushed to the HDFS and stored as files (called *HFiles*). Unlike the presented architecture, the HFiles do not have the corresponding local indices (regions) appended to them. HBase, instead, maintains a global index file in HDFS and the contents of the

local indices are merged into the global index file (called the *META* table), everytime the local buffer is flushed to HDFS. HBase uses Log Structured Merge Trees (LSM trees [12]) as the data structure for each region to make the merging of these indices as efficient as possible. HBase also does not use separate query servers. The HMaster handles the query preprocessing itself.

A key difference in the presented architecture, compared to existing distributed solutions like HBase and Cassandra [10], is the adoption of local indexes instead of a global index. Specifically, with template B+ trees, a separate index is associated with each data file in the distributed file system. This design principle simplifies the workflow of tuple insertion and thus provides more opportunities for throughput optimization.

Chapter 3

Index Scheme

In our approach, the data servers employ B+ trees as the core data structure for tuple indexing. Each data server has its own in-memory B+ tree that is built on-the-fly by repeated top-down insertions of incoming key-value pairs. As a new key-value pair (k, v) comes into a data server, the value v is first serialized and appended to the local buffer. The offset of the value v in the local buffer (say o_k) and the length of the serialized value (say l_k) are noted. Finally, the key k is used to insert the pair (o_k, l_k) in the B+ tree. Once the local buffer reaches its capacity, the contents of the local buffer as well as the B+ tree index are written to a file in the distributed file system. The local buffer is emptied and a new B+ tree index is constructed for subsequent key-value pairs that come in.

A major drawback of vanilla B+ trees is the expensive node splits required to maintain the maximum permissible fanout of B+ tree nodes (called the *order* of the B+ tree) and the difficulty of balancing the tree structure. Consider the example B+ tree shown in the top half of Figure 3.1. If a new key is to be inserted in node a, the node gets stretched beyond its maximum size and triggers a sequence of bottom-up node splits starting at a, which consequently splits nodes b and r. These split operations hurt the processing throughput and increase the insertion latency.

3.1 Template-based B+ Tree Index

Motivated by the observation that the distribution of key values of tuples changes very slowly with time in a variety of domains, we conjecture that the B+ trees constructed over

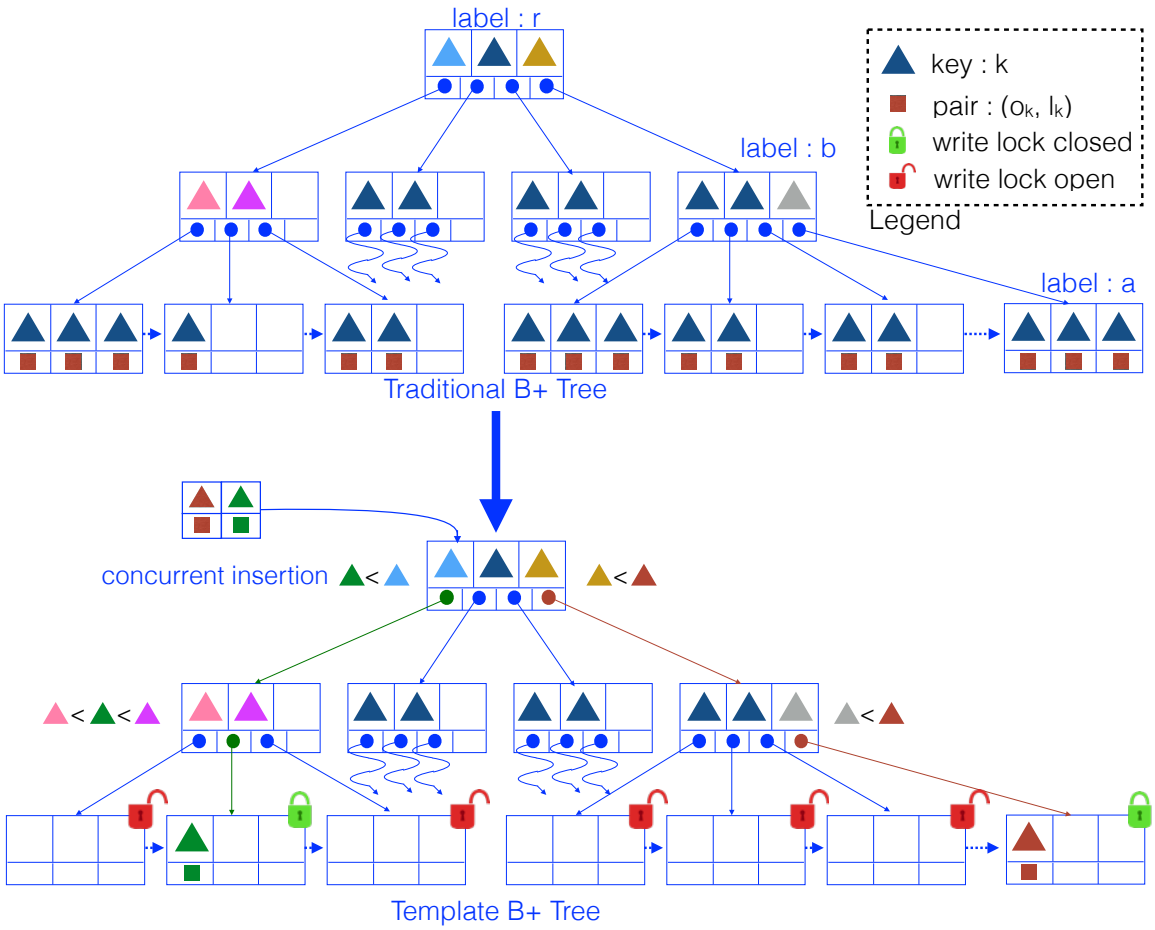

Figure 3.1: The process of formation of a template B+ tree from a traditional B+ tree; concurrent index insertion in a template B+ tree

subsequent batches of tuples are similar to each other (assuming the batch size is reasonably large). While the payload or the specific values in the leaf nodes may vary across different B+ trees, the distribution of keys in internal nodes does not change much.

We optimize the construction of a B+ tree in the following manner. When a new B+ tree is initialized for new incoming tuples, we do not build it from scratch. We just empty the leaf nodes of the previously constructed B+ tree and keep the intermediate nodes of the tree intact. We call the combination of these intermediate nodes a *template*, which contains a complete domain partitioning plan for the incoming tuples. When a tuple is inserted into the tree, the data server finds the appropriate leaf page by traversing the tree from top to bottom. If the leaf page still has space for a new tuple, the data server directly adds the

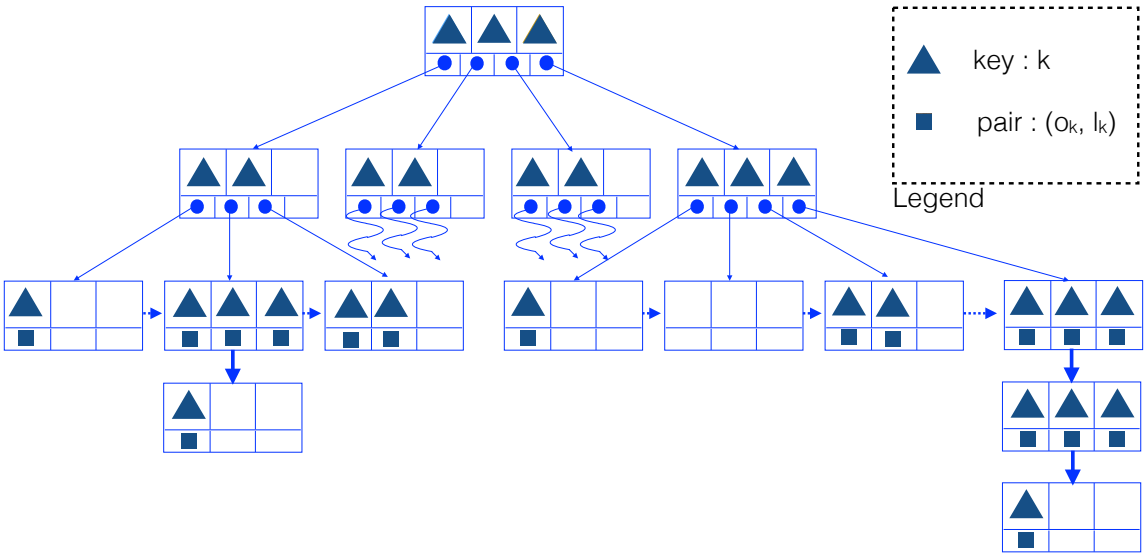

Figure 3.2: Template B+ tree with two stretched leaf nodes; one stretched by one overflow node and the other stretched by two overflow nodes

tuple to the page. Even if the overall distribution of key values does not change dramatically, there may not be room for a new tuple on a leaf page. To tackle the overflow problem, we allow the leaf nodes to stretch beyond their original size by linearly chaining additional leaf nodes to the original leaf node, as shown in Figure 3.2. But, since the overall distribution of key values is the same, the probability of a leaf node stretching to a size much larger than its original size is very small. In our empirical validation, we find the occurrence of such overflows is very rare on real workloads. The amortized cost of leaf page expansion is thus minimal.

3.2 Concurrent Insertion

Index insertion in a template B+ tree is a two stage process. The two stages are searching through the template tree to find the right leaf node for insertion and inserting the tuple into the found leaf node respectively. A data server that inserts tuples into the template B+ tree structure has to perform only read operations in the intermediate nodes, which can be done concurrently for multiple nodes. The write operation, for each arriving tuple, is restricted to a single leaf node, and such writes can be done concurrently if the tuples are inserted in different leaf nodes. These properties enable us to optimize insertion performance by using more than one concurrent thread for index insertion in data servers. We ensure thread-safety in data servers by implementing write locks for leaf nodes.

Concurrent insertions are possible only because of the fixed internal structure of template B+ trees. Such concurrent insertions in traditional B+ trees lead to intricate race conditions and incur huge overhead. For example, consider the concurrent insertion of two tuples in the traditional B+ tree depicted in Figure 3.1. Let the leaf node labelled a be the appropriate node for insertion of one of the tuples (say t_1). The insertion of t_1 leads to the splitting of internal nodes b and r. It is possible that the thread handling the insertion of t_1 initiates the split of node b at the same time another thread (handling the insertion of the other tuple t_2) starts reading node b (to find the appropriate leaf node for t_2). This creates a race condition that is difficult to handle. Despite extensive research efforts, e.g. [8, 9], the scale-up effect of concurrent insertion is poor in traditional B+ trees. As shown by our experimental results in Figure 4.5, this kind of scalability is an important advantage of template B+ trees.

3.3 Query Processing

Single point queries and range queries are performed on template B+ trees in the same manner as on traditional B+ trees. In our implementation of B+ trees, we connect each leaf node to its right sibling node. This standard optimization makes processing of range queries

more efficient. Given a range query, we first find the leaf node containing the left bound of the range query and then sequentially read all keys, traversing through sibling nodes, until the right bound of the range query is reached.

Most real-world workloads are a mixture of insertions and queries. We have found in our experiments that range queries can be very slow, especially for highly un-selective queries. Unlike traditional B+ trees, template B+ trees with write locks on leaf nodes can perform index insertions as well as range queries concurrently. This possibility of concurrency under mixed workloads makes template-based trees much more efficient than traditional B+ trees in real-world scenarios.

Chapter 4

Experimental Results

4.1 Experiment Setup

All experiments are run on Amazon Web Services (AWS) EC2. We rented four EC2 instances (or nodes) of type c3.xlarge, each of which are equipped with 4 vCPUs of Intel Xeon E5-2680v2@2.8 GHz, 7.5 GB memory and 80 GB storage space. We implemented template B+ trees on top of Apache Storm 0.9.5 [2]. Hadoop Distributed File System (HDFS) was deployed on one of the four nodes with block size 64 MB. Except the HDFS node, all other nodes host one supervisor process and two worker processes for Storm processing. The maximum heap sizes allocated to Nimbus, UI and supervisors are set at default Storm values, i.e., Nimbus with 1024 MB, UI with 768 MB and supervisor with 256 MB. The maximum heap size for every worker process is 1024 MB.

The dispatchers, the data servers as well as the query servers in our architecture are all implemented as Storm bolts. The data is read from dataset files stored in HDFS and passed to the dispatchers (i.e. dispatcher bolts) by Storm spouts. In experiments involving template B+ trees with concurrent insertions, we spawn a thread pool (i.e. a collection of daemon threads) during initialization of the bolt. Insertion of the tuple into the local buffer is done by the executor thread of the bolt (i.e. the main process thread of the data server) itself and the insertion of the key into the template B+ tree is delegated to the previously initialized thread pool (i.e. one of the idle threads from the thread pool handles the insertion into the template B+ tree).

In Storm, all bolts have inherent queues for incoming tuples. A tuple delivered to a bolt

waits in this queue while the the bolt finishes processing all previously delivered tuples. The total time spent by a tuple in a bolt (the total latency in other words) includes this queue waiting time in addition to the processing latency of the bolt itself. In latency experiments, we wanted to measure only the processing latency (i.e. how much time an indexer bolt or the data server spends on indexing the tuple), not the latency due to the tuple waiting in a bolt queue. We achieved this in the following manner. We configured the indexer bolts (i.e. data servers) to send an acknowledgement to the spout that forwarded the tuple as soon as the tuple is indexed. We further configured the spouts to generate new tuples only after the acknowledgement for the previous tuple has been received. As a result, a new tuple is forwarded to a data server only when the previous tuple has been fully processed and hence the new tuple does not spend any time in the queue.

For throughput experiments, we wanted the indexer bolts (i.e. data servers) to be at their full capacity, i.e. processing tuples one after the other without any idle time. We achieved this by gradually increasing the number of spouts (thus increasing the arrival rate of incoming tuples) until the output throughput at data servers saturated.

The following approaches, including three baseline methods, are used in our experiments:

1. **No-Index**: does not use any global or local index for the incoming data. Each data server dumps the data to the distributed file system when the buffer is full. To answer queries from a query server, each data server retrieves all of its data files from the distributed file system and scans each file to find matching tuples. This approach is used to illustrate the maximal data processing throughput the system could achieve with the available computation resources and no indexing.

2. **HBase**: adopts an LSM-tree as the index scheme. The incoming tuples are partitioned and sent to different data servers. Each data server maintains an LSM-tree for data indexing. When a buffer fills up, its tuples are dumped to the distributed file system for persistent storage, and its LSM tree is merged with the global tree. We used the

Apache HBase 1.1.3 [1] implementation with default parameter settings to generate the performance numbers presented in this thesis. The performance numbers of HBase presented below are similar to those presented in the HBase benchmarking paper [7].

3. **B-Tree**: uses the traditional B+ tree index structure on the data server. A new B+ tree is built from scratch every time the previous tree is flushed to HDFS. This scheme also does not support concurrent insertions and query processing.

4. **TB-Tree**: is our approach with the template B+ tree index structure. The initial template B+ tree is created by building a traditional B+ tree from scratch, based on the first buffer of data in the data set, and then emptying its leaf nodes. Thus, these template B+ trees might require leaf nodes to stretch as the buffer is refilled, because the quality of the template depends on the dataset and the sequence of tuples that arrive during the initial B+ tree construction.

We tested on three different workloads, including two real datasets and one synthetic dataset.

1. *Teleco* dataset: this 9 GB dataset contains 59 million caller detail records (CDRs) from a telecommunication company in Singapore. Each CDR contains the location of a mobile phone user and their action on the phone, such as a phone call, messaging and Internet access. We indexed the CDRs based on user location, after mapping the 2-dimensional coordinate to a single dimension using the Z-order curve [6] to preserve spatial locality. The distribution of key values of tuples from this dataset is shown in Figure 4.1. The histogram shows the distribution of three consecutive sequences of 500k tuples from the dataset. As evident from the histogram, the distribution drifts very slowly with time.

2. *Building* dataset: this 4 GB dataset contains 150 million sensor readings from a smart building. These sensors provide fine-grained measurements of the energy consumption

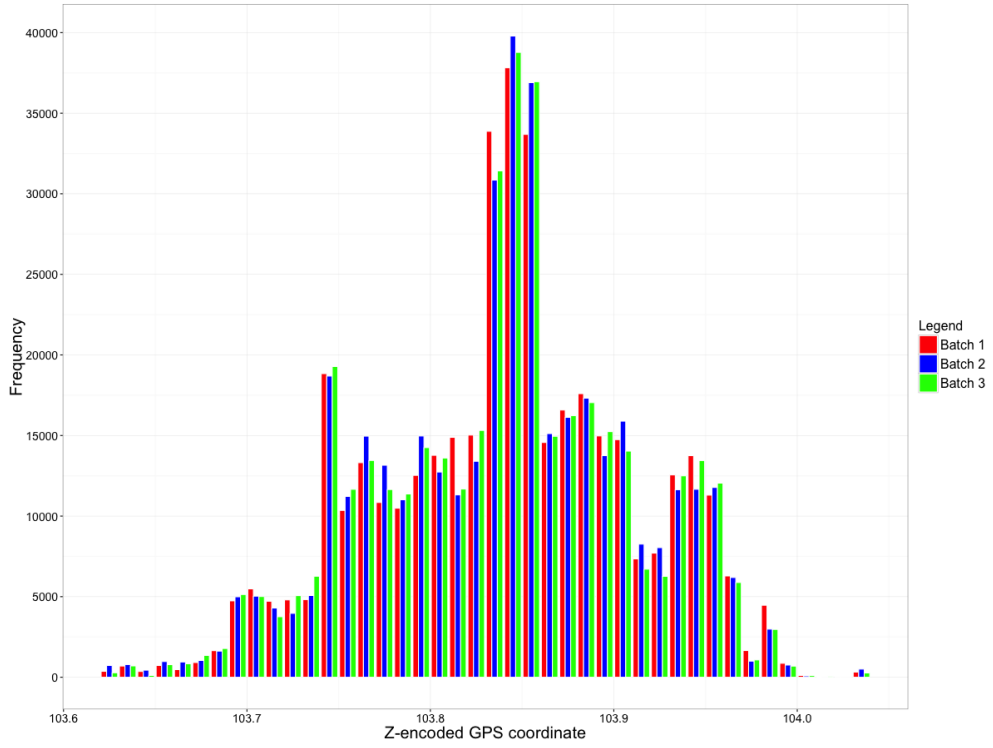

Figure 4.1: Histogram of the key values in three sequential batches of 500k tuples each from the *Teleco* dataset. In each bin of the graph, the three bars appear in batch order, i.e., red-blue-green

of individual machines in the smart building. The readings of these sensors are used as keys for indexing. The distribution of key values of tuples from this dataset is shown in Figure 4.2 which shows that this dataset has a bit more drift than Teleco does.

3. *Synthetic* dataset: this 1 GB dataset follows a mixed Gaussian distribution, with three independent Gaussian components (refer to Figure 4.3). This distribution is kept constant and does not drift with time.

We report a number of measurements on the performance of the approaches:

1. *Insertion Latency*: is the time elapsed between a tuple reaching a data server and the completion of the insertion of the tuple in the local index of the buffer and the data server. No measurements are taken at a data server until its buffer has been filled and written out once, to provide a template for subsequent use.

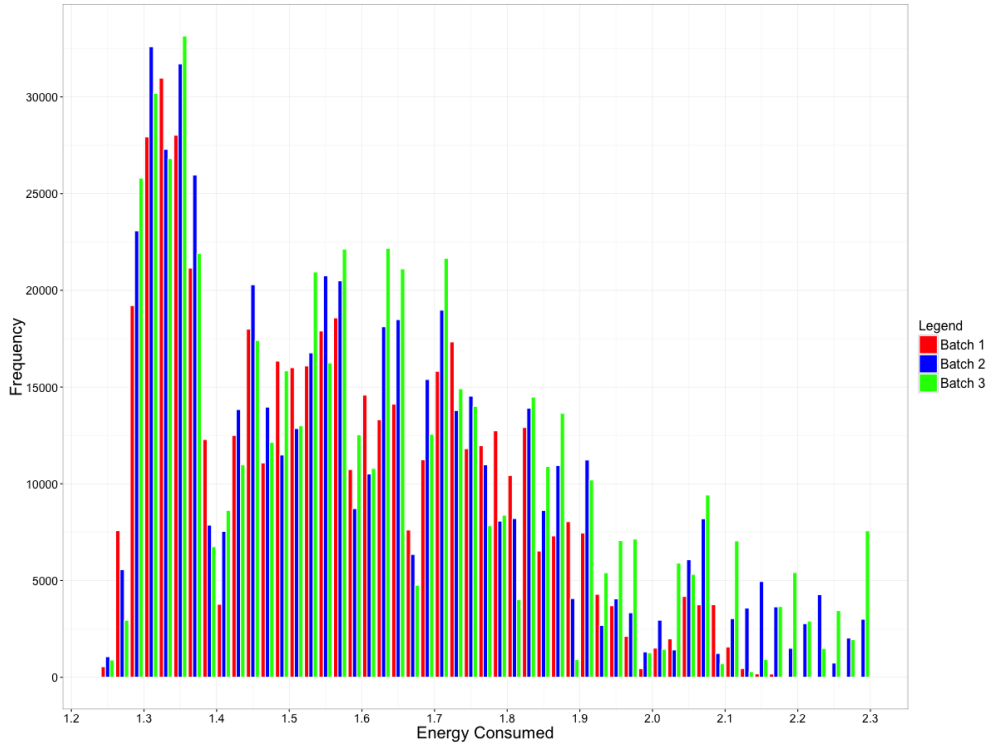

Figure 4.2: Histogram of the key values in three sequential batches of 500k tuples each from the *Building* dataset. In each bin of the graph, the three bars appear in batch order, i.e., red-blue-green

2. *Insertion Throughput*: is the number of new tuples processed per second (i.e. TPS) by an individual data server. No measurements are taken at a data server until its buffer has been filled and written out once, to provide a template for subsequent use.

3. *Query Completion Latency*: is the time elapsed between a query reaching a query server and all satisfying tuples being retrieved by the data servers.

4. *Mixed Throughput*: is the number of tuples processed per second by an individual data server when subjected to mixed workloads comprised of index insertions as well as range queries. No measurements are taken at a data server until its buffer has been filled and written out once, to provide a template for subsequent use. No queries are issued before that point, either.

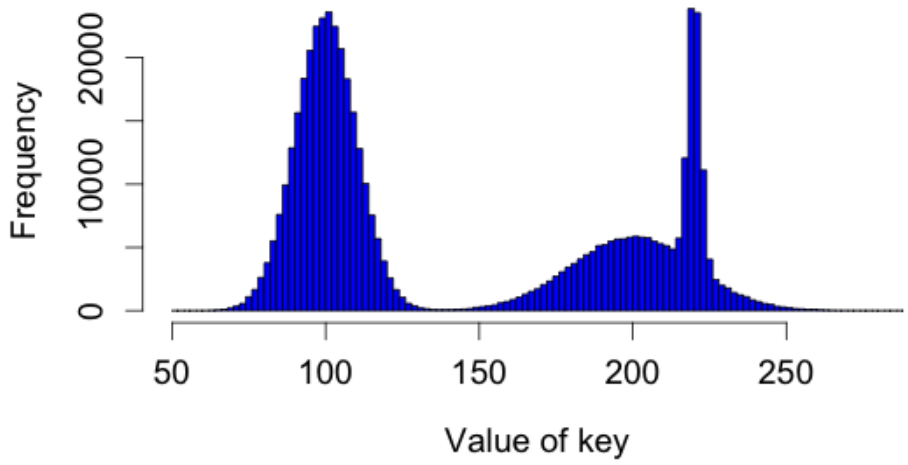

Figure 4.3: Histogram of the key values in the *Synthetic* dataset

4.2 Experiment Results

Quality of Template B+ Tree. The template B+ trees used in all our experiments are derived from the initial B+ trees created from scratch using the initial buffer of arriving tuples, in the manner shown in Figure 3.1. We measure the quality of a template B+ tree with respect to a given dataset as the percentage of leaf nodes that have additional overflow nodes attached to them. When a template is of higher quality for a given dataset, fewer leaf nodes are stretched to much larger sizes than their initial size. The quality of B+ trees for tree fanouts of 4, 10 and 100 is tabulated in Tables 4.1, 4.2 and 4.3. In the three tables above, the number of tuples inserted per tree (or average buffer capacity) is 500k tuples and the results are averaged over 100 fillings of the tree.

As evident from a comparison of Tables 4.1, 4.2 and 4.3, a higher fanout value increases the quality of a template B+ tree. A smaller fan-out, which in turn implies a smaller number of keys in each node of the tree, increases the probability of leaf nodes in the template tree being stretched to much larger sizes than their original size. We prove the above claim as follows: Consider a random leaf node r_n in a template B+ tree. Let l and u be the lower and

upper bounds of the possible key values in the leaf node decided based on the keys of the intermediate nodes in the template B+ tree. Let f_1 and f_2 be two possible fan-out values, such that $f_1 < f_2$. Under the assumption that all arriving tuples have key values which are independent and identically distributed samples, we can define a finite value (p_{lu}) as the probability of a tuple having key value within the range $[l, u)$. The probability of the tuple being inserted in r_n is also p_{lu} because the ranges of possible values in leaf nodes are disjoint in a template B+ tree.

$$\mathcal{P}(r_n \; is \; stretched \mid f_1) = \mathcal{P}(||r_n|| \geq f_1 \mid f_1) \; = \; p_{lu}^{f_1} \; + \; p_{lu}^{f_1+1} \; + \; \infty \; = \; \frac{p_{lu}^{f_1}}{1 - p_{lu}} \tag{4.1}$$

$$\mathcal{P}(r_n \; is \; stretched \mid f_2) = \mathcal{P}(||r_n|| \geq f_2 \mid f_2) \; = \; p_{lu}^{f_2} \; + \; p_{lu}^{f_2+1} \; + \; \infty \; = \; \frac{p_{lu}^{f_2}}{1 - p_{lu}} \tag{4.2}$$

Since $f_1 < f_2$ and $p_{lu} \in (0, 1)$, comparing the above two equations, we see that

$$\mathcal{P}(r_n \; is \; stretched \mid f_1) > \mathcal{P}(r_n \; is \; stretched \mid f_2) \quad if \; f_1 < f_2 \tag{4.3}$$

So, the probability of a leaf node being stretched is higher for smaller fan-out values.

Dataset	1 overflow node	2 overflow nodes	3 overflow nodes
Teleco	11.68%	2.70%	0.00%
Building	10.58%	8.44%	0.10%
Synthetic	7.55%	0.45%	0.00%

Table 4.1: Percentage of leaf nodes that have overflowed, for a template B+ tree with fanout 4

Dataset	1 overflow node	2 overflow nodes	3 overflow nodes
Teleco	1.18%	0.25%	0.00%
Building	3.00%	0.94%	0.01%
Synthetic	0.76%	0.08%	0.00%

Table 4.2: Percentage of leaf nodes that have overflowed, for a template B+ tree with fanout 10

Dataset	1 overflow node	2 overflow nodes	3 overflow nodes
Teleco	1.00%	0.03%	0.00%
Building	2.98%	0.88%	0.00%
Synthetic	0.37%	0.01%	0.00%

Table 4.3: Percentage of leaf nodes that have overflowed, for a template B+ tree with fanout 100

Index insertion latency. The latency of index insertions in TB-Tree and other baseline approaches is shown in Table 4.4. The buffer capacity is set at 500k tuples for all the approaches and the reported latency values have been averaged over 10 fillings of the buffer. TB-Tree is 11 times faster than HBase, but the no-index store is faster than TB-Tree by 56%.

While HBase is slower overall, we cannot attribute that difference directly to LSM-trees. HBase and Storm have very different architectures, optimized for different kinds of workloads. A stream tuple insertion workload is ideal for Storm, while HBase's architects were thinking primarily about the needs of batch analytics over massive data already on disk.

Approach	Average Latency (ms)	Standard Deviation
No-Index	0.036	0.002
HBase	1.020	0.200
B-Tree	0.100	0.008
TB-Tree	0.082	0.006

Table 4.4: *Insertion Latency* in *Teleco* dataset for various approaches

As explained in Section 2.1, we store only as many tuples in a template B+ tree as can be stored in an HDFS block of size 64 MB (i.e. our in-memory buffer size is fixed at 64 MB). The number of tuples that can be stored in the buffer is not constant but a function of the number of bytes that serialized tuples occupy in memory. Thus, depending on the number of bytes required by serialized tuples on average, the total number of tuples in a template B+ tree (just before it is flushed to HDFS) may vary with application domain and dataset. Figure 4.4 shows the variation of index insertion latency with the average number of tuples that fit in the buffer (or buffer capacity) in the local template B+ tree index of a data server.

We see that the latency increases as the number of tuples increases. This increase in latency can be attributed to two causes in a traditional B+ tree with a fixed fan-out. First is the increased depth of the tree, which increases the time required to find the correct leaf node for a new tuple. The second is the increased number of node splits during tree construction. While the first cause cannot be mitigated by a template-based approach, it successfully mitigates the second cause. This is the reason why on increasing the number of tuples for a fixed fan-out, the B+ tree slows down at a faster rate than TB-Tree. This in turn is the reason that the latency savings in switching from B-Tree to TB-Tree grow, both in percentage and absolute units, as the total number of tuples per tree (and therefore the buffer capacity) grows.

The fan-out of TB-Tree is another important parameter that affects performance. When the total number of tuples to be inserted in a template B+ tree is kept constant, the fan-out controls the depth of the tree; the depth of a template B+ tree as a function of fan-out f and the number n of tuples inserted is $\mathcal{O}(log_f n)$. A smaller fan-out reduces the worst-case time complexity of searching for a tuple in an in-memory index like ours ($\mathcal{O}(log_f n \times f)$). But the probability of a leaf node being stretched is higher for smaller fan-out values, as discussed in the preceding section. There is essentially a trade-off between search time complexity and maintaining the size of leaf nodes in template trees (which also affects search complexity under the hood). As an rule of thumb derived empirically, we assert that a low fan-out value works well when the data distribution has lower variance and a high fan-out value works best when the data distribution has high variance. Keeping in mind the cacheline size of 64 bytes, we measure the insertion latency of template B+ trees for fan-outs 4 and 10, representing the approximate lower bound and upper bound of node sizes in an in-memory B+ tree.

Comparison of improvement in insertion latency of a template B+ tree over a traditional B+ tree for fan-outs of 4 and 10 reveals that, for a fixed number of tuples, lower fan-outs give higher improvement. This follows intuition because for a fixed number of tuples, a higher fan-out implies lower tree height and vice-versa. Since a lower height means fewer

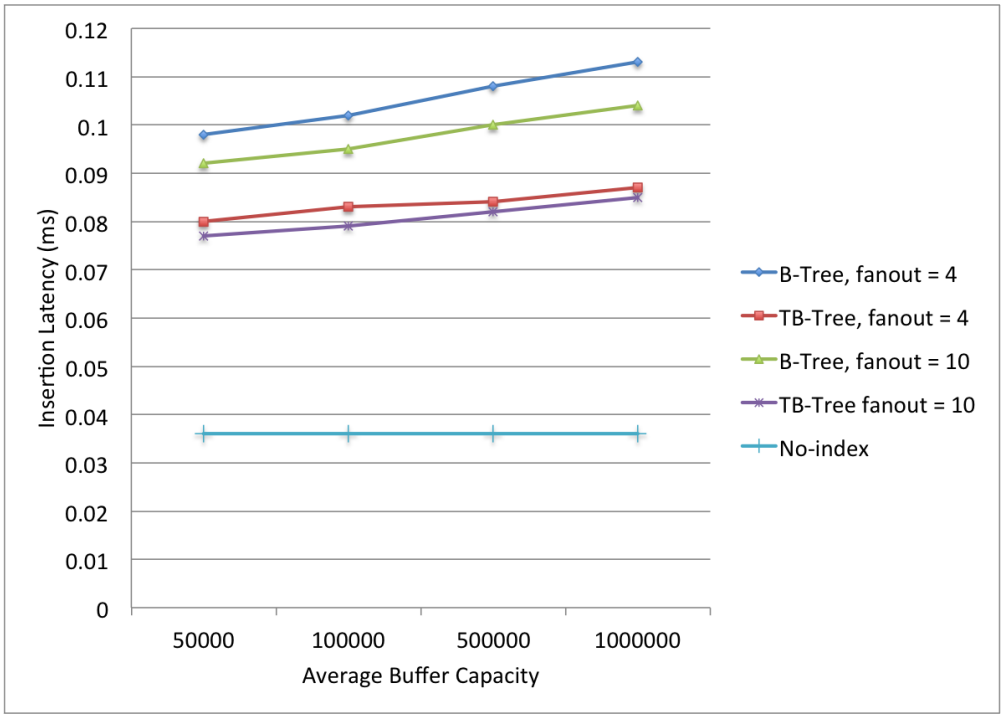

Figure 4.4: *Insertion Latency* in the *Teleco* dataset. Latency of HBase = 1.02 ms (not shown in the figure to make the trends for the other schemes prominent).

node splits, the potential scope of improvement in latency by avoiding node splits is lower. Hence, as Figure 4.4 shows, switching from B-Tree to TB-tree at lower fanouts yields higher gains.

Index insertion throughput. We study the throughput performance of traditional B+ trees against template B+ trees and template B+ trees with concurrent insertions. The insertion throughput across three datasets, averaged over 100 fillings of the local buffer per dataset, is plotted in Figure 4.5. The number of threads used for concurrent insertions is set to 16 and the number of tuples per tree is set to 500k in the above plots. We find that switching from B+ trees to template trees with concurrent insertions improves insertion throughput by roughly 1500 to 2000 tuples per second.

We see that the concurrent insertions in a template B+ tree achieve a much higher

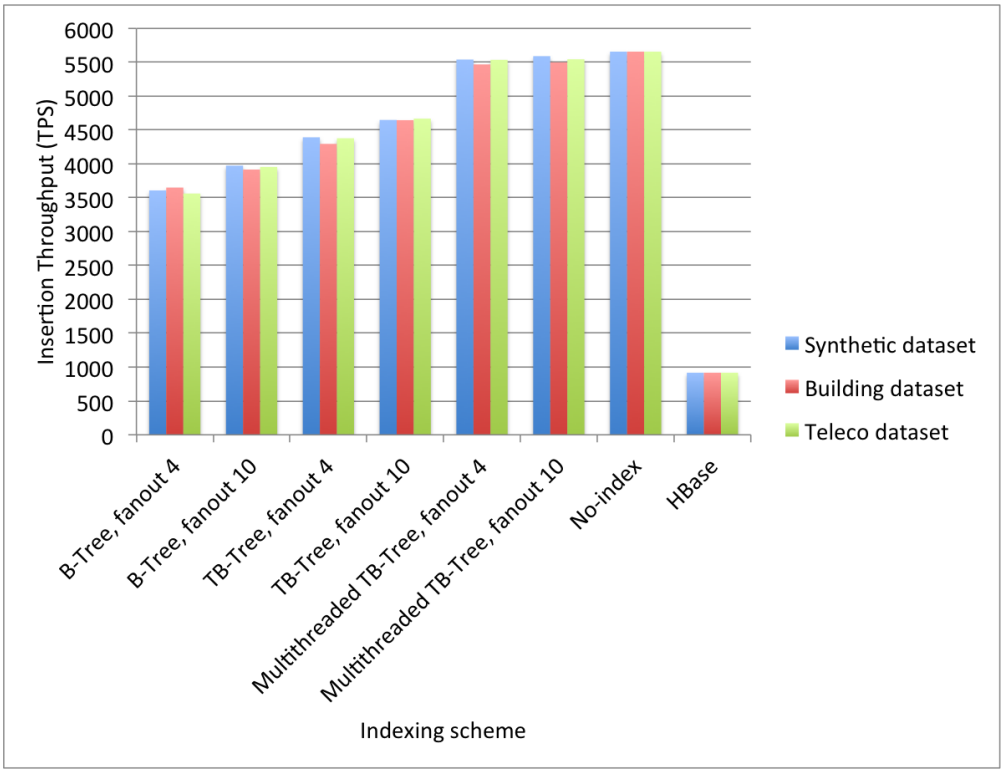

Figure 4.5: *Insertion Throughput* of various schemes under 100% insertion workload

throughput than traditional B+ trees. In fact, template B+ trees with concurrent insertions achieve almost the same throughput as a no-index datastore. The increase in throughput from traditional B+ trees to template B+ trees is commensurate with the decrease in latency. The increase in throughput from template B+ trees to template B+ trees with multi-threaded insertions is due to increased parallelism, i.e. multiple tuples are being inserted into different leaf nodes of the template tree at the same time. The improvement of performance in multi-threaded insertions is a function of the local sequence of tuples. If tuples are sequenced in such a way that all or a majority of subsequent tuples are inserted in the same leaf of a template B+ tree, then the multi-threaded insertions will not yield much improvement because the effective parallelism is greatly reduced. It is because of the same reason that for any given sequence of tuples, the throughput improvement brought about by an additional insertion thread saturates after a certain number of threads. Insertion throughput as a function of the number of threads for the *Teleco* dataset is presented in

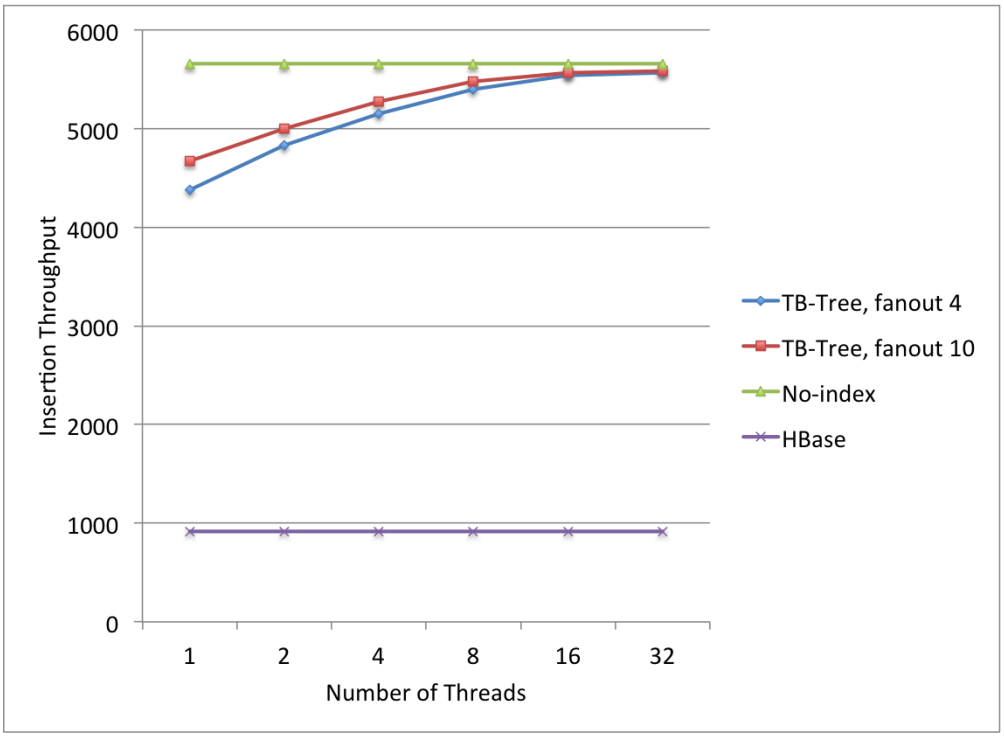

Figure 4.6: *Insertion Throughput* of TB-Tree as a function of the number of concurrent threads with 100% insertion workload

Figure 4.6.

The insertion throughput as a function of number of tuples in the index (or buffer capacity) is shown in Figure 4.7. We see that the throughput of all the tree-based indexing approaches decreases slightly as the number of tuples increases. This follows a similar reasoning as discussed in the case of insertion latency. As the number of tuples increases, the depth of the tree increases. This increases the time required to find the appropriate leaf node for the arriving tuple and also increases the number of node splits required. TB-Tree outperforms B-Tree by avoiding the node splits.

Query Completion latency We report the average query completion latency of various approaches using 60k range queries of various selectivity. These range queries are generated by sorting the datasets and then uniformly randomly choosing the lower bounds of the range queries and then choosing the appropriate corresponding upper bounds from the sorted list

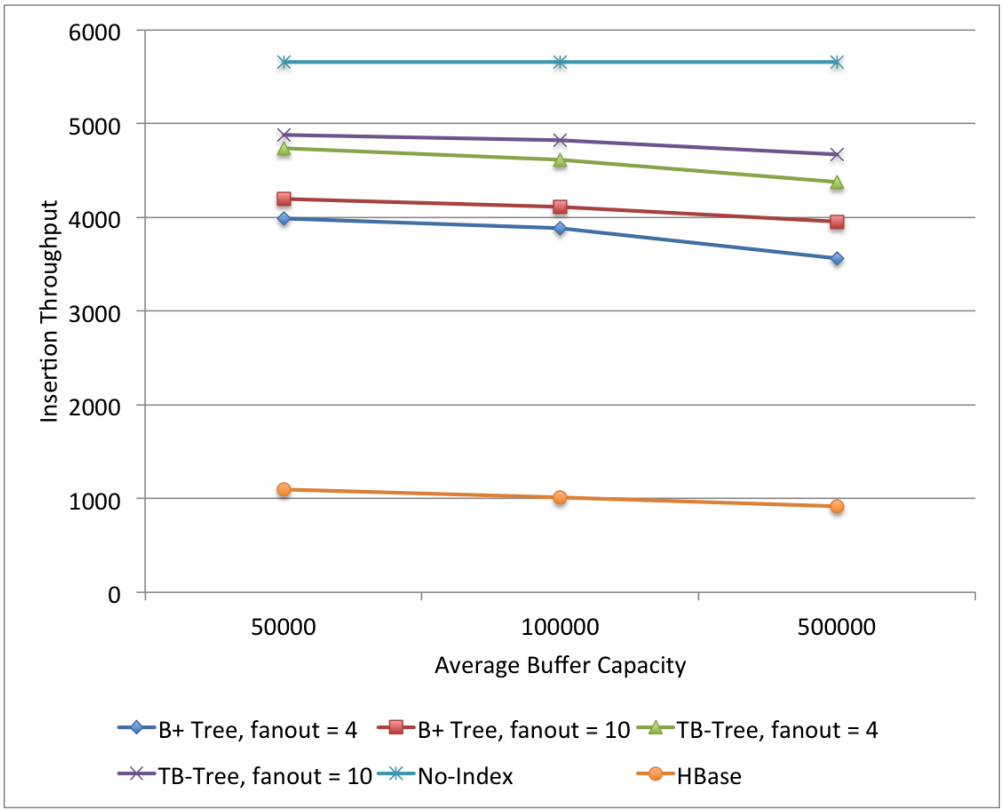

Figure 4.7: *Insertion Throughput* as a function of average buffer capacity in the *Teleco* dataset

such that the selectivity criteria is satisfied. The latencies (in ms) for traditional B+ trees and template B+ trees are shown in log scale in Figure 4.8 as a function of query selectivity. The number of tuples per buffer is fixed at 500k tuples and fanout is fixed at 10. We see that for B+ trees and template trees, the query completion time is directly proportional to the number of tuples in the query result.

We also see that range search in template B+ trees is only marginally slower than traditional B+ trees (at most 3.2% increase in latency). In the case of highly selective queries, both traditional B+ trees and template B+ trees perform better than HBase as well as the No-Index store. But HBase scales better with a decrease in query selectivity, and the latency of B-tree and TB-tree approaches no-index datastore performance when queries are not very selective (the former is only 26% better than the latter when query selectivity is 10%).

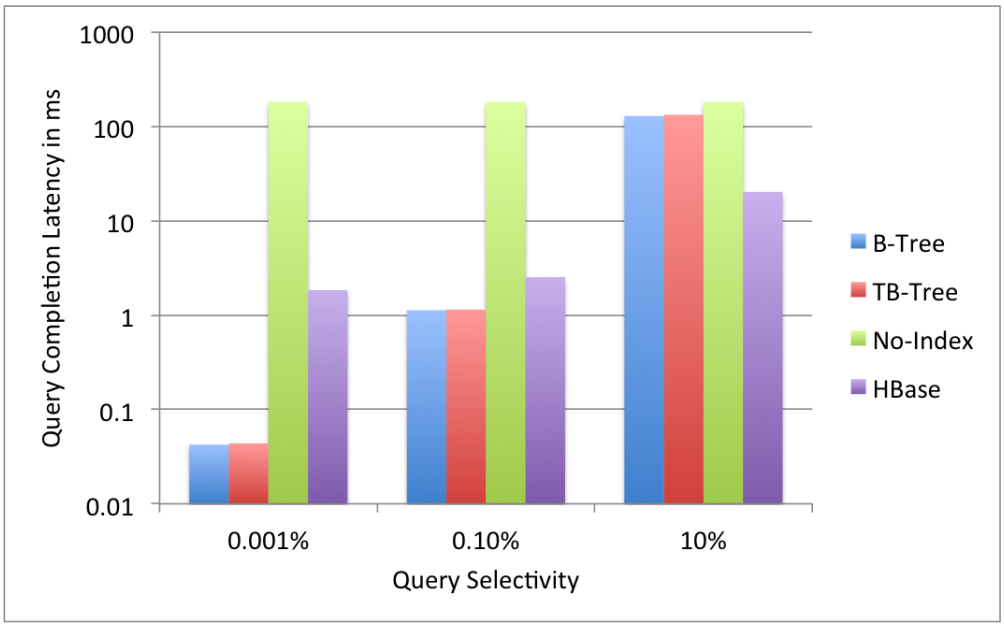

Figure 4.8: *Query Completion Latency* as a function of query selectivity

Indexing throughput under mixed workloads We present the resultant throughput for a 99-to-1 mixture of insertions to queries. We set the queries to be 1% of the workload just to magnify the effects of the query workload. In real world scenarios where the index insertions are arriving at a very high rate, the range queries generated by users will form a much lower constituent of total workload.

We compare the mixed workload throughput achieved by traditional B+ trees and TB-Trees with 16 insertion threads. The number of tuples is 500k and the fanout is 10. The insertion throughput is plotted in Figure 4.9 as a function of query selectivity. We can see that a huge increase in throughput has been achieved by multi-threaded template B+ trees. This boost in throughput has been possible because as one thread is busy satisfying a range query, the other threads can continue insertions.

Size of indices We measured the approximate size of B+ tree indices using YourKit Java Profiler 2015 [4]. These indices are created for a buffer capacity of 500k tuples and keys in the *Double* domain and the results reported are averaged over 100 flushes of local buffer to HDFS per dataset across all datasets. The memory occupied by a traditional B+ tree was

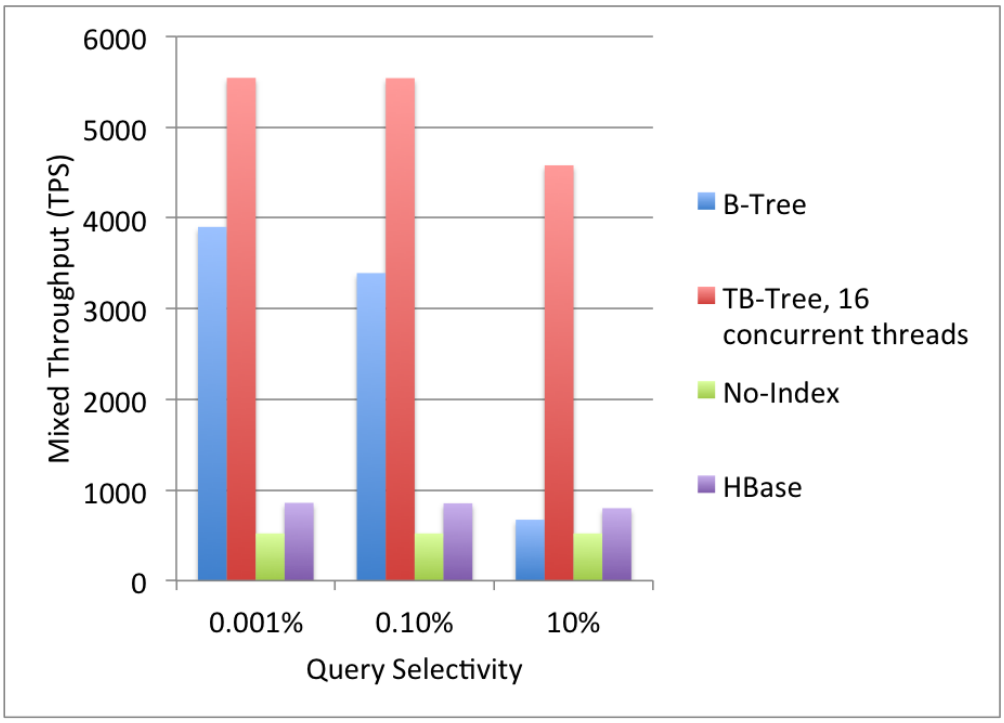

Figure 4.9: Indexing throughput under mixed workload; 99% insertions and 1% range queries

7.35 mb for fanout 4 and 6.31 mb for fanout 10. The template B+ tree occupied 7.91 mb and 6.85 mb for fanouts 4 and 10 respectively. The slightly higher memory occupancy of template B+ trees is due to additional bookkeeping variables and any overflow leaf nodes required. Since tuple values are not stored in indices (instead only their offsets in the local buffer are inserted in the index), the memory occupancies reported here are entirely indexing overheads.

Chapter 5

Conclusions

This thesis presents template B+ trees for distributed append-only stores. Template B+ trees support high throughput tuple insertion and efficient key-based retrieval. Using template B+ trees enables higher utilization of CPU resources while performing concurrent tuple insertions. Our experimental results show that compared to traditional B+ trees, template B+ trees improve insertion throughput by 40% to 57%, compared to traditional B+ tree indexes. Template B+ trees also beat the state-of-the-art system HBase, with higher insertion throughput and lower query processing latency. These initial results are very promising, and suggest a number of questions for future work to explore.

1. Domain Partitioning: The architecture proposed in this thesis does not employ any form of domain partitioning between data servers. A domain partitioning scheme divides the domain of key values into non-overlapping sub-domains such that the total load is evenly balanced between data servers. It also helps in query processing since, with domain partitioning, queries can be selectively forwarded to only those data servers which are handling the appropriate sub-domains. Our architecture achieves load balancing between data servers by forwarding tuples to data servers in a round-robin fashion. But this scheme does not help during query processing as queries have to be forwarded to all data servers. It may be possible to design an adaptive domain partitioning scheme which keeps the input load between data servers balanced at all times. The major challenge in creating such a scheme is that the template used by a template B+ tree in a data server is closely tied to the sub-domain that the data server is handling. If the domain partitioning protocol changes the sub-domain assigned to

that data server, the template has to change with it. This requires some degree of synchronisation between dispatcher servers and data servers, which is hard to achieve.

2. Quality of template B+ trees: In this thesis, we have shown that template B+ trees perform well when the overall distribution of tuples does not drift much with time. It will be valuable to fully characterize the quality of template B+ trees when subjected to a sudden change or a gradual drift in overall distribution of tuples. In this thesis, we used the percentage of highly stretched leaf nodes as a proxy for the quality of a template B+ tree. This helps us to judge the quality of template B+ trees only in a relative sense. We think that it should be possible to derive a quality metric which compares the template B+ tree with the theoretically best possible B+ tree that can be grown using the same batch of tuples. The absolute value of such a metric will have a firmer conceptual grounding and can be used to determine whether the template of a template B+ tree needs to be updated.

3. Updating the template of template B+ trees: In this thesis, we do not propose any scheme to update the template of a template B+ tree. As the overall distribution of tuples gradually drifts over time, the template needs to be updated along with it. In our present proposal, the only way to update a template is to create a new B+ tree from scratch and then flush its leaf nodes, which is not efficient. It should be possible to efficiently generate the new template directly from the previous template, using some statistical information from the most recent buffer of tuples to generate a new template.

4. Fault tolerance: We do not explore the fault tolerance mechanisms in our architecture and rely on the fault tolerance of Storm in our implementation. Without the fault-tolerance of Storm, our architecture would not be tolerant against data server failures which lead to potential data loss. In case of a data server failure, all the tuples that have been stored in the local buffer so far (and their corresponding index), but have not

been flushed to HDFS, would be lost forever. An interesting open question is the design of techniques that will make our architecture robust against failures of data servers as well as dispatchers. Replication and creating disk-based (or Zookeeper-based) logs are potential candidate schemes for this purpose.

5. Performance comparison with LSM trees and other alternatives: With promising initial performance results for template B+ trees in hand, an in-depth comparison with other indexing alternatives is a natural next step. In particular, it will be useful to compare template B+ trees to alternatives such as LSM trees in a neutral environment. Our experiments used LSM trees in their native HBase implementation, which includes many other architectural choices that affect performance and are different from Storm's.

References

[1] Apache HBase, https://hbase.apache.org/.

[2] Apache Storm, https://storm.apache.org/.

[3] Spark Streaming, http://spark.apache.org/streaming/.

[4] Yourkit Java Profiler, https://www.yourkit.com/docs/.

[5] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. BigTable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2), 2008.

[6] S. Chen, B. C. Ooi, and Z. Zhang. An adaptive updating protocol for reducing moving object database workload. *Proceedings of the VLDB Endowment*, 3(1-2):735–746, 2010.

[7] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[8] G. Graefe. Write-Optimized B-Trees. In *VLDB*, pages 672–683, 2004.

[9] G. Graefe. B-tree indexes for high update rates. *SIGMOD Record*, 35(1):39–44, 2006.

[10] A. Lakshman and P. Malik. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, Apr. 2010.

[11] V. C. Liang, R. T. B. Ma, W. S. Ng, L. Wang, M. Winslett, H. Wu, S. Ying, and Z. Zhang. Mercury: Metro density prediction with recurrent neural network on streaming CDR data. In *ICDE*, 2016.

[12] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (LSM-tree). *Acta Inf.*, 33(4):351–385, June 1996.

[13] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop Distributed File System. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[14] W. Tan, S. Tata, Y. Tang, and L. L. Fong. Diff-index: Differentiated index in distributed log-structured data stores. In *EDBT*, pages 700–711, 2014.

[15] S. Wu, G. Chen, X. Zhou, Z. Zhang, A. K. H. Tung, and M. Winslett. PABIRS: A data access middleware for distributed file systems. In *ICDE*, pages 113–124, 2015.

[16] Z. Zhang, M. Hadjieleftheriou, B. C. Ooi, and D. Srivastava. Bed-tree: an all-purpose index structure for string similarity search based on edit distance. In *SIGMOD*, pages 915–926, 2010.

[17] Z. Zhang, H. Shu, Z. Chong, H. Lu, and Y. Yang. C-cube: Elastic continuous clustering in the cloud. In *ICDE*, pages 577–588, 2013.