CS561 Project: A Dual Tree System

Jianqi Ma
Boston University

Ning Wang
Boston University

Lanfeng Liu
Boston University

ABSTRACT

B^+ tree is a common technique used for building index in database systems. A general purpose of an index is to bring "order". For an ordered/nearly-ordered workload, B^+ trees however doesn't exploit the existing "sortedness" of the data. In this research project, we build a dual tree systems, which contains two B^+ trees to leverage the advantages of B^+ tree along with the "sortedness" of the data when performing insertion operations. We use the workload generator to conduct our experiments on workload with different "sortedness", and try to show in which cases a dual-tree system is worth the trade-off. For the detail of the project, see link: "https://github.com/CS561-DualTree/dual_tree"

1 INTRODUCTION

1.1 Motivation

A traditional B^+ tree insertion can be extracted to 2 phases. The first phase focus on find the corresponding leaf node. When the database has an insert, it needs to traverse from the root to find the correct position of the leaf node that contains the key for this insert. Once you find the node, you can simply insert that into this node. Then the second phase is invoked, which is splitting B^+ tree nodes. In order to insert into some full node, a split on this node is required. One problem with this vanilla B^+ tree is that phase one is always going to happen for every insert it gets, which is clearly not necessary for ordered/nearly-ordered workload, because with a order workload, every insertion must happen at the end of the right most leaf node. So having some way of using the "sortedness" of workloads would be really helpful in reducing insertion overheads.

1.2 Problem Statement

Our goal for this project is to try to build some variation of the B^+ tree that utilize the "sortedness" of its workload to avoid unnecessary insert overheads in the first phase of insertion. Moreover, since the read performance is negatively impact for sure using a dual tree system, we also aim to determine in which case this system are worthy in terms of read and write trade-offs.

1.3 Contributions

(1) Build a basic dual-tree system that consist of a sorted tree and an unsorted tree. Unsorted tree is implemented as a vanilla B^+ tree. Sorted tree is implemented with a special insert method that directly append key to the rightmost leaf node and split accordingly.
(2) Based on these basic dual-tree, we add some other components including a heap buffer, an outlier detector and a query buffer. First two components are optimizations for insertions while the last one help to speed up queries. Each components has one or more corresponding tuning knobs so that the system can adjust itself with respect to different workload.
(3) Conduct experiments with insertion workloads of different "sortedness", and report how different levels of "sortedness" could affect the write performance of this dual-tree system as well as how our system could exploit the sortedness in the data set.

2 BACKGROUND

Previous research [1] had explored the advantage of near sortedness in query evaluation. They formally defined what it means for a relation to be near sorted with 2 parameters, where k is the number of elements that are out of order and l is how far the out of order elements are from their sorted positions. To take advantage of the sortedness of data, they designed algorithms for operations such as natural join, set operation and sorting to be executed more efficiently. In addition, they developed efficient approximation algorithm to determine the sortedness of a relation so they can decide when to use their algorithm. However, they did not address how insertion might also benefit from nearly sorted data.

3 ARCHITECTURE

The dual B^+ tree system consists of a sorted tree and an unsorted tree. Here sorted and unsorted refer to the "sortedness" of inserted workload. Namely, we insert in-order data to the sorted tree and insert out-of-order data to the unsorted tree. As stated in Section 1.3, both trees in the dual-tree system are vanilla B^+ tree except that a special insert method is required for the sorted tree to bypass the search phase of a traditional B^+ tree insertion. We also have an extra layer on top of the two trees to decide which tree to insert for a order/nearly-ordered workload.

Database systems are always about trade-offs. In our dual tree system, we also introduce some tuning knobs that help adjust the system to different workloads. Subsections of Architecture will follow this format: First we introduce a component or a strategy, and then we list and explain its corresponding tuning knobs. We will describe details of the component or the strategy, and how corresponding tuning knobs can affect it. Before diving into different components of the system, a summary of the dual-tree system is needed.

The figure 1 below shows the overview of the dual-tree system. It consists of 5 main components along with some insertion and query strategies. The $heap\ buffer$ is a minimum-heap that buffers inserted tuples. The $outlier\ detector$ is responsible for checking whether a new inserted tuple is an outlier with respect to the sorted tree. The $sorted\ tree$ and the $unsorted\ tree$ is the main components of the dual-tree system, the first one receives tuples with sortedness while the second one receives other tuples. The $query\ buffer$ plays an important role when querying tuples, it supports query policies like MRU(Most Recently Used) to help decide query which tree first. We also implemented parallel query(query both trees concurrently), but the result turns out to be slower than the single thread query, it will be discussed in the experiment section.

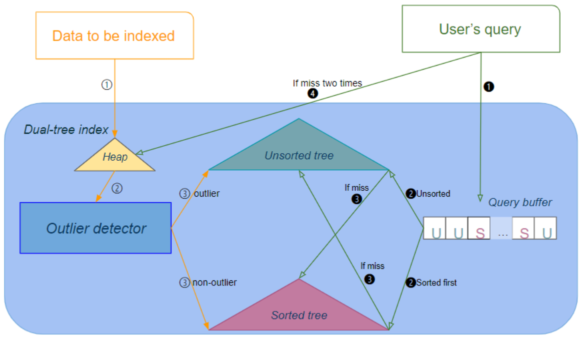


Figure 1: The overview of the dual-tree system

3.1 Insertion in Sorted Tree

3.1.1 Basic insertion. Before discussing these insertion components, we take a look into the basic insertion involving only two B^+ trees. First we implement a new method named *insert_to_tail* that only appends tuples to the B^+ tree, and thus the splitting of leaf nodes and internal nodes will occur only at the rightmost one in each layer. The *sorted tree* only calls *insert_to_tail* method for inserting, and the *unsorted tree* only calls standard insertion method for inserting. Hence, for every newly added key, we first check whether it is greater than the maximum key of the *sorted tree*, if it is, insert it to the sorted one, else insert it to the unsorted one.

However, there are two obvious drawbacks of these "dumb" method. First, the space utility of the *sorted tree* is low, because all nodes except the tail leaf node are at most half-full. Second, If a relatively great key is inserted to the *sorted tree*, it will keep many keys away from the *sorted tree*. Hence, we implement two optimizations along with two corresponding tuning knobs:

- *SORTED_TREE_SPLIT_FRAC*: The value of this knob should be chosen from $[0.5, 1)$, it expresses in ratio how many tuples will remain in the original node after splitting it. By adjusting this knob, users can move as few keys as possible to the new tail leaf.
- *ALLOW_SORTED_TREE_INSERTION*: The value of this knobs should be chosen from $\{true, false\}$. When it is false, only appending tuples to the tail leaf of the sorted tree is allowed. When it is true, the dual-tree system will record the minimum key of the tail leaf of the sorted tree, thus if any new key falls between this key and the maximum key of the sorted tree, it can be directly inserted to the tail leaf. This will cost more than simply appending a new key since movement of keys is inevitable, however it still save us some cost compared to inserting to a B^+ tree because no search of the tree is required. Besides, by doing so, more keys can be put into the *sorted tree*

3.1.2 Heap buffer. Even though optimizations above could improve the performance, but it is still not enough. For example, if there is an array {700, 701, 1, 2, 3, 4, 5, ..., 1000} which will be inserted to the dual-tree system, only about one third of all keys will be put into the *sorted tree*. To solve this problem, we use a minimum heap buffer for insertion. Here we provide a new tuning knob:

- *HEAP_SIZE*: The value of this knob should be an non-negative integer. When the value is no more than 1, no heap buffer is used. The value represents how many keys it can store.

However, the size of the heap buffer should not be too large, because the cost of maintaining a heap is non-negligible. In our test, the good a heap buffer of size 128 brings will be offset by the bad it brings when the size of the data set is 100 thousand keys or more.

3.1.3 Outlier detector. A heap buffer does help us in some cases, but the case in the figure 2 illustrate a situation where heap buffer provides limited improvement. In this figure, a minimum heap buffer which has a size of 3 is used for insertion, as the process proceeds to the key 57, two outliers have been inserted to the *sorted tree*, which causes the next three or even more keys to be inserted into the *unsorted tree*. Thus, we introduce the outlier detector for preventing outliers from being inserted to the *sorted tree*

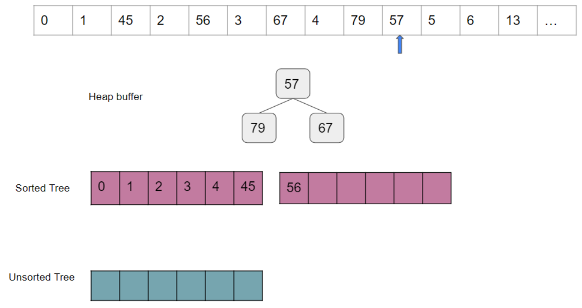


Figure 2: The problem with buffer

To detect an outlier, here we choose to use the average distance between every two consecutive keys of the *sorted tree* as the metric. The easiest way of detecting outliers with this metric is to compare the average distance with the distance between a new key and the maximum key of the *sorted tree*, the new key is allowed to be inserted into the *sorted tree* only if the distance is equal or less to the average distance. For example, if we use the outlier detector in figure 2, key 45 and key 56 will be inserted into the *unsorted tree*. Still, this easiest way has a very obvious drawback: When the average distance stay in a low value, it is possible that no key will be inserted. For instance, if we replace the key 5 with the key 7 in the figure 2, no more keys will be inserted to the *sorted tree* after inserting the key 4. Hence, we should tolerate some keys that are not too far away from the maximum key. A tuning knob is used to adjust the tolerance level:

- *INIT_TOLERANCE_FACTOR*: The value of this knob should be a non-negative number. If it is set to 0, the outlier detector is disabled. Assume the distance between the new key and the maximum key of the *sorted tree* is $dist_{new}$, and the average distance is $dist_{avg}$, when

  $$dist_{new} \leq dist_{avg} \cdot tolerance\_factor$$

  the new key will be appended to the sorted tree. Here the value of $tolerance\_factor$ is equal to the *INIT_TOLERANCE _FACTOR* if no modification are made to the $tolerance\_factor$.

However, a fixed tolerance factor can also bring us a problem: the average distance will keep growing if unluckily a serious of large keys are inserted to the *sorted tree*. The figure 3 shows the situation:

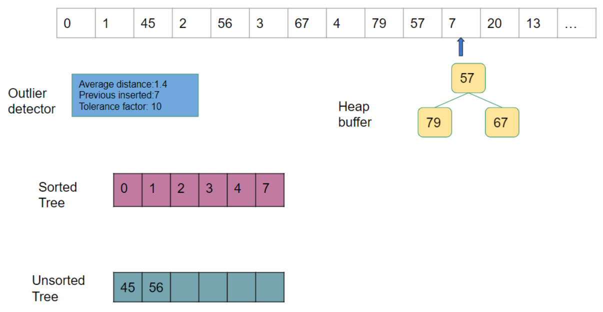


Figure 3: The problem with the easiest detector

After inserting the key "7", the average distance become 1.4, which means the real outlier key "20" will be inserted into the sorted tree because $1.4 \cdot 10 > 20 - 7$, and the average distance will grow again. If the average distance keeps growing, more and more outliers will be inserted into the sorted tree, and the dual-tree system will fail to use the sortedness of the data set. Therefore, to stop the average distance from continuous expansion, a new tuning knob is provided:

- *EXPECTED_AVG_DISTANCE*: The value of this knob should be any number that greater than 1. If the value is less or equal to 1, then no *EXPECTED_AVG_DISTANCE* is used. Otherwise, every time we update the average distance, we also update the *tolerance_factor* following the formula below:

$$tolerance\_factor^{`} = tolerance\_factor \cdot \frac{expected\_avg\_distance}{average\_distance}$$

In the real implementation, since the average distance cannot be exactly equal to the expected one, we make some concessions. If the average distance is greater than the expected one by a limited value(like 0.5 in our implementation) or less than the expected one, we consider the average distance as having reached the expectation, thus we will change the tolerance factor back to its initial value(*INIT_TOLERANCE_FACTOR*).

3.2 Query in Dual $B^{+}Tree$

3.2.1 Naive Implementation. Since the Dual $B^{+}Tree$ System optimize insertion performance by introducing a new tree, some overhead while querying is inevitable. The most naive version of query is to query both tree on after the other and terminate at found. This is not enough because essentially going through both of the trees would create 2x latency of the single $B^{+}Tree$ system.

3.2.2 Tree Selection. In order to reduce query time, we decide to add a tree selection module on top of the actual querying process. Intuitively, we begin with querying the tree/buffer with the smallest size first, so that if we hit the data being queries, we can avoid to go look it up in the larger tree/buffer. However, experimental results show that doing so doesn't really make a difference. The query process still have an around 1.9x latency of a single $B^{+}Tree$ for sequential and random workload with random data (with 50% noise and 50% window size). The other way of doing this is to query the tree/buffer with the largest size first. This way the probability of hit the data being queries with one query is higher. Namely, the probability of querying multiple times for one query request is lower than query the smaller tree/buffer first. This approach works really well. Based on our experiment, sequential and random query on nearly sorted data (with 10% noise and 50% window size) can achieve around 1.1x latency of a single $B^{+}Tree$. Sequential and random query on random data (with 50% noise and 50% window size) can achieve 1.5x latency of a single $B^{+}Tree$. More detailed results are discussed in Section 4.

3.2.3 MRU Buffer. To apply the same idea of exploiting metadata of datasets, we decide to add a Query buffer to our Dual $B + Tree$ System. The query buffer is just a array with size set by a tuning knob QUERY_BUFFER_SIZE. Query buffer isn't really storing any queries. Instead of that, it stores which tree did the latest QUERY_BUFFER_SIZE number of queries found the data it queried. With this metadata of the query workload, we can then determine which tree to query first statistically by choosing the one that is accessed more often. We believe this MRU could be useful for periodic query workload (workload that periodically access the same part of data), but we haven't conduct experiments against periodic query yet.

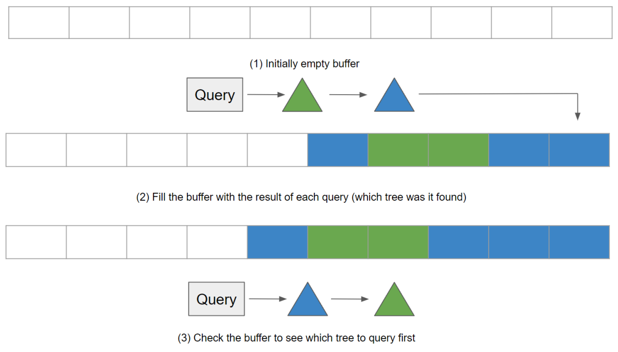

(1) Initially empty buffer
Query
(2) Fill the buffer with the result of each query (which tree was it found)
Query
(3) Check the buffer to see which tree to query first

Figure 4: MRU Buffer Workflow

3.2.4 Parallel Query. To fully speed up the query execution, we also tried the parallel query: one thread queries the sorted tree, while the other one queries the unsorted tree. However, after trying 3 different implementations of parallel query, the performance still not meets our expectation.

At first, we try to start a thread for every query, and terminate the thread after the query finishes. The problem brought by this method is obvious: the cost of creating and terminating one thread is negligible, but we have to create and terminate a thread for every query, which will cause a huge cost, and finally slow down the query process. we simply test the performance and it turns out that the time cost by this method is much more than the time of single query thread.

In the second strategy, except the main thread, we only create a thread one time for querying the unsorted tree. We use a message

Knob name	Function	Domain	Type
SORTED_TREE _SPLIT_FRAC	Decide how many keys remain in the original node after splitting.	$[0.5, 1)$	Insertion knob
ALLOW_SORTED _TREE_INSERTION	Allow insertion to the tail leaf of the sorted tree.	$\{true, false\}$	Insertion knob
HEAP_SIZE	Define the size of the heap buffer, 0 means no heap buffer is used.	$\{1, 2, 3...\}$	Insertion knob
INIT_TOLERANCE _FACTOR	Define initial tolerance factor. If it is 0, then the outlier detector is disabled.	$(0, +\inf)$	Insertion knob
EXPECTED_AVG _DISTANCE	The expectation of the average distance of the sorted tree. If it is less or equal to 1, the tolerance factor is fixed.	$(1, +\inf)$	Insertion knob
QUERY_BUFFER _SIZE	Query buffer size for determine which tree to query first statistically, if it is zero, Query buffer is disabled.	$[1, +\inf)$	Query knob

Table 1: Tuning knobs of dual-tree system

queue to block the second thread: every time the dual-tree system receives a query, it adds the query key to the message queue, and the second thread will take the key out; if the queue is empty, then the second thread will be blocked. By using this strategy, even though there are multiple queries received by the dual-tree system at one time, it can still handle all of them, which means the system is able to handle concurrent queries. The performance does increase a lot compared to the first strategy, but it doesn't outperform the latency of querying single B^+ tree either.

Thinking that the higher latency of the second strategy is possibly brought by blocking and notifying the second thread many times, in the third strategy, we stop blocking the second thread, instead, we make it constantly check the message queue, which means the second thread is now busy waiting. The third strategy do improve the performance a lot, which makes the latency only about 1.7 times higher than the latency of querying in the single tree, and it is a huge improvement compared to the previous two strategies. However it still doesn't meet our goal, because theoretically at least the latency should not be greater than that of the single tree. We didn't figure out why the performance of two thread is even slower, one possible reason we think is that two threads can contend for the cache storage, which makes the cache update much more times, thus downgrades the performance.

4 EXPERIMENTS

4.1 Benchmark Explanations

The workload is generated according three parameters: *domain size*, *noise*(%) and *Window Threshold*(%). *Domain size* is the size of generated data, if the *domain size* is 1000, then the generated data contains integers from 0 to 999. *Noise* is the percentage of our of order elements, and *window threshold* is the window(as percentage of the total elements) within which out of order element can be placed from its original location. So a 5% noise and 5% window threshold means 5% of the total domain size of elements are out-of-order and each of them is placed within 5% window from its original position.

For query experiments, we used random query workload and sequential query workload. Random query workload we used have the same size as the insertion workload, and are contained of the same elements. The only difference is that before we query, we would randomly shuffle the workload. Sequential query workload, on the other hand, is just the insertion workload in sorted order.

4.2 Insertion Performance of raw dual-tree system

We compare the raw dual-tree system with a single B^+ tree with respect to the insertion performance. In the raw dual-tree system, we only use two trees without any additional component: one tree for inserting unsorted tuples and another for inserting sorted tuples. In the data stream, when the key value of the newly arrived tuple is greater than the maximum key value of the sorted tree, we insert the new tuple into the sorted tree, otherwise we insert it into the unsorted tree.

The Figure 5 and 6 show the results of comparing the raw dual-tree system and the single B^+ tree. We use 1M integers(from 0 to 999999) as the input data set, and change one of the $k(noise)$ and $l(window\ threshold)$ while the other remains the same.

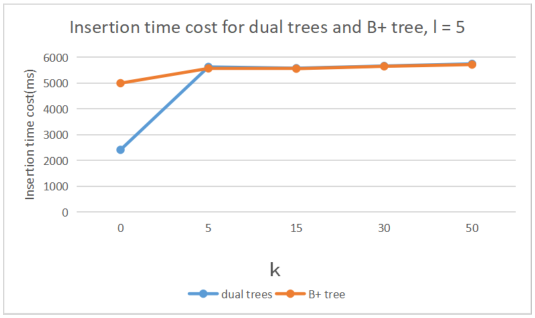


Figure 5: Compare insertion time when l is 5

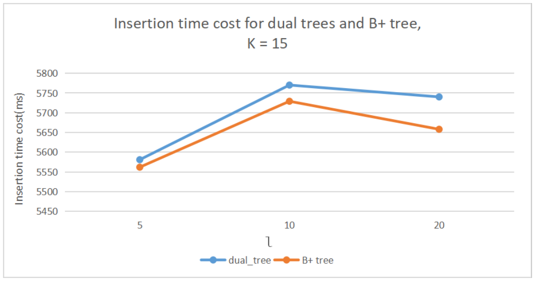
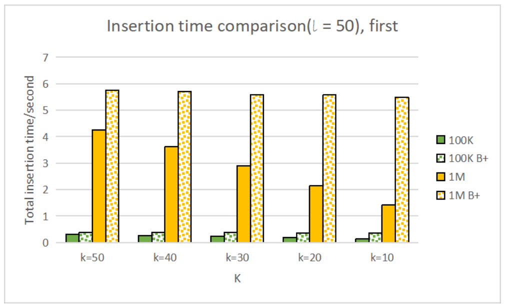
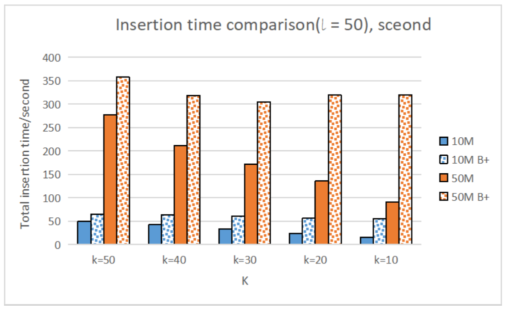


Figure 6: Compare insertion time when k is 15



(a) data size = 100K, 1M



(b) data size = 10M, 50M

Figure 7: Insertion time comparison between dual-tree and a single B^+ tree

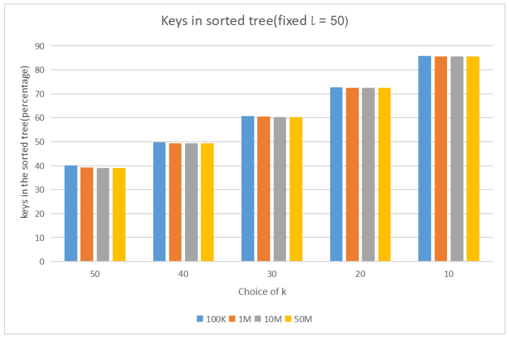


Figure 8: Compare number of keys with decreasing k

As shown in the Figure 5, when $k = 0$, which means the input data set is sorted, the cost of raw dual tree is nearly half of the B^+ tree. With the growing k, the performance of the B^+ tree barely changed, this is because the insertion of a B^+ tree follows the same pattern regardless of the sortedness of the input data. However, when the data is not strictly ordered, the performance of the dual-tree system becomes the almost same as the B^+ tree, because most tuples will not be inserted into the sorted tree in the system even though there are only a few outliers in the data set. One outlier is able to prevent 2.5% of the tuples from being inserted into the sorted tree. In figure 6, because the k is 15, the performance of the dual-tree system cannot be better than the single tree. In fact, it could be even worse than the single B^+ tree since the dual-tree system has some overhead for determining which tree the new tuple should be inserted into.

4.3 Query Performance

4.4 Insertion Performance of optimized dual-tree system

Currently in the experiment, the configuration of all 5 insertion tuning knobs is:

- $SORTED\_TREE\_SPLIT\_FRAC = 0.9$
- $ALLOW\_SORTED\_TREE\_INSERTION = true$
- $HEAP\_SIZE = 16$
- $INIT\_TOLERANCE\_FACTOR = 100$
- $EXPECTED\_AVG\_DISTANCE = 2.5$

To fully test our optimized dual-tree system, we test it using data sets of different size, and for each size, using different k and l. We choose 100K, 1M, 10M and 50M as the size of data sets. When testing on different k, fix l to 50%; when testing on different l, fix k to 35%. For each combination, we conduct 5 independent tests, and use the average for all results

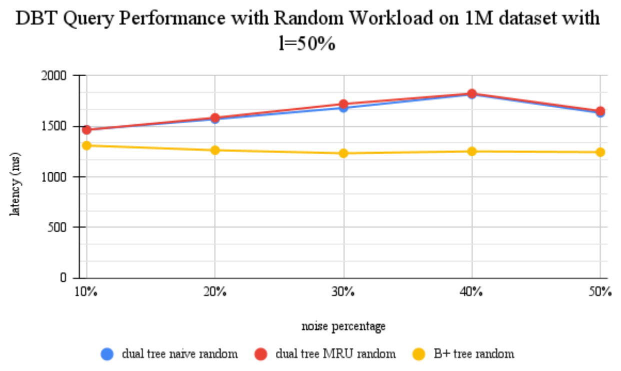
Comparing with the single B^+ tree, our dual-tree system could save a lot of time for insertion. Figure 7 shows the result of comparing the insertion time of the dual-tree system and a single B^+ tree. In both figure 7a and figure 7b, the dotted bars represent the total insertion time of a single B^+ tree, while the filled bars represent the total insertion time of the dual-tree system. The shorter the bar, the less time it takes.

As the figures show, our dual-tree system always outperforms a single B^+ tree in terms of insertion. As the value of K decreases, the insertion time of a single B^+ does not vary much, while the insertion time of the dual-tree system decreases with the value of K. Another interesting point is that , for the same dataset size, the insertion time of dual-tree system is much less(less than half) than that of a B^+ tree when K equals to 10 or 20. Combine these two points, we can say that our dual-tree system does make good use of the sortedness in the dataset.

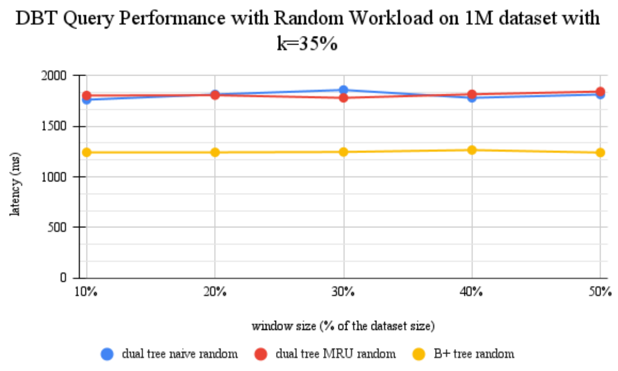
The figure 8 shows the result of how the number of keys(in percentage) in the sorted tree changes with the decreasing k. Bars of different color represent different data size.

Focusing on the single color, e.g. the blue bars, an obvious trend is that the less k, the more keys are inserted into the sorted tree. However, even the k is up to 50, for every dataset size, nearly 40% of all keys are still inserted into the sorted tree, which is much better than the raw dual-tree system. Moreover, with the growing dataset size, the percentage of keys in the sorted tree merely decrease at most 0.9%(for dataset size of 100K) and at least 0.3%(for dataset size of 50M), which means our dual-tree system has the ability to scale up.

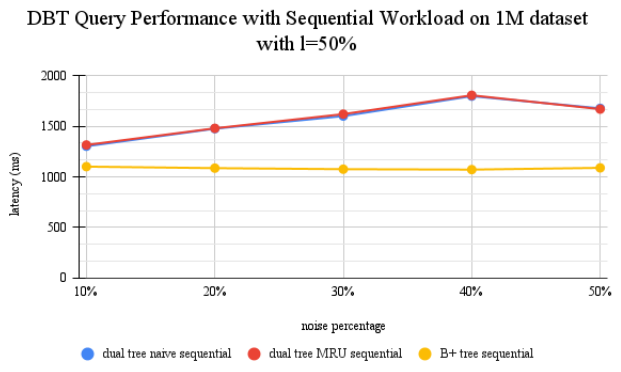
To find how the value of j could impact the performance of the system, we use a new workload generator which is still based on the k-l sortedness but use the number of keys instead of the percentage to represent k and l. The reason why we use a new generator is that when it is represented by percentage, the minimum value is 1%. When the domain size is large, like 1 million, then the minimum distance between the position of an unsorted key and its original



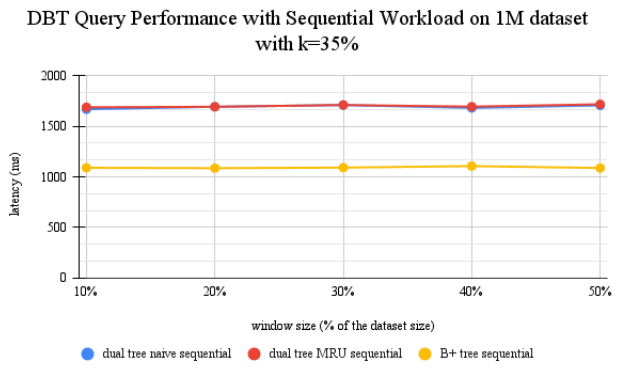
(a) Random Query with varying noise k



(b) Random Query with varying window size l



(c) Sequential Query with varying window size l



(d) Sequential Query with varying window size l

Figure 9: Dual $B^{+}Tree$ Query Performance for 1M dataset

	l = 0.5M	l = 0.1M	l = 1K	l = 100
# of keys in sorted position	250407	250290	250931	251574
# of keys in sorted tree	251451	254260	456345	958196

Table 2: Comparing keys in sorted position and in sorted tree

position would be 10000 which is still a large l. In the table 2, the second row is the number of keys in the sorted position of the data set, and the third row is the number of keys in the sorted tree. When the value of l is relatively large, as the first two columns show, the number of keys in the sorted tree is only slightly more than the number of keys in sorted positions. However, when the value of l is rather small, as in the last two columns, the number of keys in the sorted tree increases a lot.

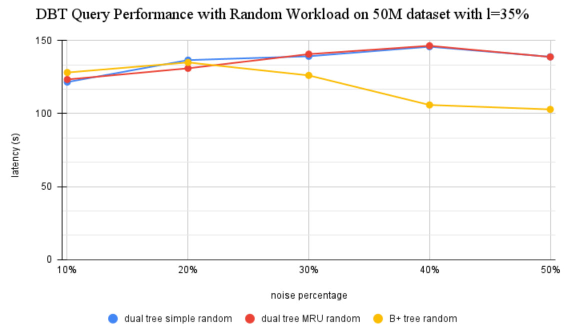
We used the following tuning knob configuration for query experiments:

- $SORTED\_TREE\_SPLIT\_FRAC = 0.9$
- $ALLOW\_SORTED\_TREE\_INSERTION = true$
- $HEAP\_SIZE = 16$
- $INIT\_TOLERANCE\_FACTOR = 100$
- $EXPECTED\_AVG\_DISTANCE = 2.5$
- $QUERY\_BUFFER\_SIZE = 20$

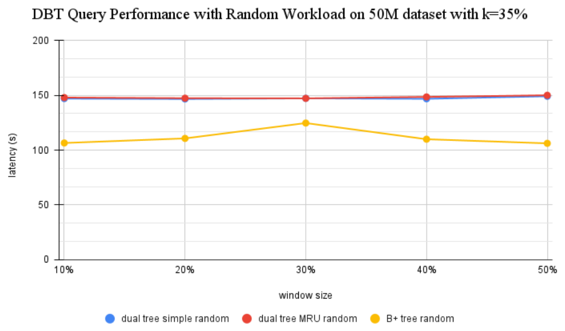
Our query experiment set up is similar to insert experiment. We test the Dual $B^{+}Tree$ system against 100K, 1M, 10M, and 50M sized workloads. For each size, we generate 5 different workloads based on different combination of noise percentage(k) and window size(l), and query against all 5 workloads then register the mean as its latency. Since Most of the experimental results are showing similar trends, we are only going to show test results from 1M and 50M workloads.

As shown by Figure 9, we can see that when the data is nearly sorted (k=10%), we can reduce the overhead of query two trees to around 1.1x baseline latency for both random and sequential queries. However, as the noise percentage k increases, the accuracy of predicting which tree to query start to drop, and therefore we can see the increase in query overhead. Worst case scenario, we can still get a 1.5x baseline latency for random query and 1.8x latency for sequential query. On the other hand, window size doesn't really change the performance of query, which could due to our outlier detection mechanism is efficient. The Implementation of MRU buffer doesn't really provide any performance boosts in random query and sequential query. However, we think MRU buffer could work better with other workloads which we have not tested yet.

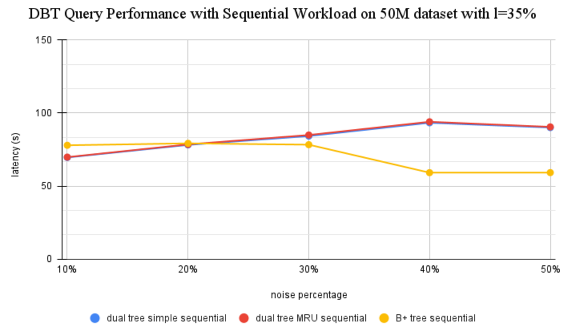
For a large enough dataset, as shown in Figure 10, the query performance for nearly sorted workload is very similar. We believe



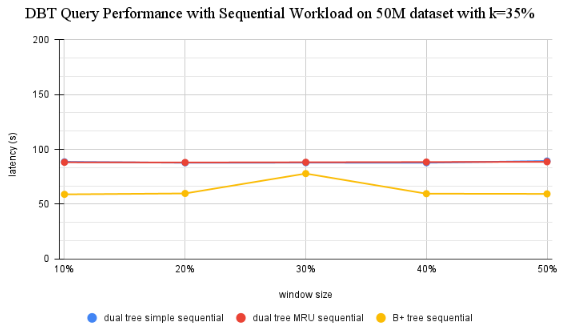
(a) Random Query with varying noise k



(b) Random Query with varying window size l



(c) Sequential Query with varying window size l



(d) Sequential Query with varying window size l

Figure 10: Dual B^+Tree Query Performance for 50M dataset

this is due to the significant size difference between the two trees in dual B^+ tree. As k increases, aka, workload becomes less sorted, their difference gets larger, where dual B^+ tree gets higher query response time but single B^+ tree gets lower.

5 CONCLUSION

We proposed a new data structure called dual B^+ tree to specifically improve the insertion time for nearly sorted workload. Dual B^+ tree consists of two trees, where one tree is used to insert in order elements and the other tree the rest. For the in order tree, which we call the sorted tree, elements are inserted to the tail so it's more efficient. To improve the performance for less sorted or unsorted workload, we implemented a heap buffer and an outlier detector to make sure more keys are inserted into the sorted tree. To benchmark its performance against a single B^+ tree, we generated workloads of varying sortedness. As a result, the dual B^+ tree has consistently better insertion time than single B^+ tree, even for completely unsorted workload. The difference becomes larger as the workload gets more sorted. However, one major drawback of the dual B^+ tree is that query time is higher. The simple query method results in a 10% to 40% increase in cumulative query response time than single B^+ tree. We proposed a query algorithm called MRU query that might help for mostly repetitive queries but according to our initial experiment, it makes little difference for random and sequential queries. Theoretically, taking advantage of parallelism where we query both trees at the same time could potentially improve the performance. But our experiments indicated that parallel query is significantly slower for some unknown reason. Future research could build upon this idea to improve query performance.

There are still several outstanding issues waiting to be solved in the future. For query, how to boost the parallel query is still unknown, a possible solution is to bind each thread with some particular cache storage, so that there will be no cache contention when running two threads for querying. For insertion, although it can make sure that almost all sorted keys can be inserted into the sorted tree, the system still cannot handle a data set with a large l, a new mechanism is needed, like sampling before inserting so that we can know the approximate value of l and adjust the tolerance factor of the outlier detector.

REFERENCES

[1] S. Ben-Moshe, Y. Kanza, E. Fischer, A. Matsliah, M. Fischer, and C. Staelin. *Detecting and Exploiting Near-Sortedness for Efficient Relational Query Evaluation.* In Proceedings of the International Conference on Database Theory (ICDT), pages 256–267, 2011