CS561 : *Benchmarking* Dual B+ trees for Near-Sorted Workloads

Jingyi Huang
Boston University

Meng Heng Lee
Boston University

Shaolin Xie
Boston University

Abstract

B+ tree has long been a popular database architecture for storing large amounts of data that cannot be stored in the main memory due to the fact that it utilizes indexing and a high fanout to reduce the number of I/O operations needed and achieve more efficient search. Moreover, the internal nodes of a B+ tree only contain the keys, ensuring that all the pointers to value are stored at leaf nodes that remain at the same height. The leaf nodes are linked using a linked list, which enables the B+ tree to do sequential search as well as random access. However, for insertions, regardless of the orderliness of the datasets that have been fetched, a B+ tree will eventually need to do a full scan and insert the data in its associated nodes, which leads to extra insertion run time.

To address this challenge, in this paper, we implemented a new key-value storage architecture, Dual B+ tree[1], that utilizes two B+ tree and investigate how this can help improve the run time of insertion.

1 Introduction

1.1 Motivation

We were inspired by another research paper [1], where it provides theoretical foundations for improving query evaluation over possibly nearly-sorted relations. While constantly maintaining sorted database relations can result in insertions and updates to become inefficient, there are still order-preferring operations whose evaluation is far more efficient with the relations sorted. However, the sorting process is time consuming and requires large numbers of I/Os to achieve. The paper shows how order-preferring operations is not optimal for the nearly-sorted relations. We, building on top of that, want to show specifically how nearly sorted relations are not optimal or ideal for B+ tree indexing structure either. Moreover, it inspired us to implement Dual B+ tree that can possibly overcome this issue.

1.2 Problem Statement

The extra run time exists because when insertion in B+ tree, The efficiency in searching and inserting of a general B+ tree is not related to the orderliness of the data. That is, if data arrives in order from small to large, instead of simply inserting the data to the right-most leaf of the tree, B+ tree will do the full searching and inserting process regardless, which would ultimately result in longer run time than there needs to be.

1.3 Contributions

Our contributions are as follows:

(1) We implemented a prototype of Dual B+ tree.
(2) We scale queries to large data workloads while considering the new trade-offs of new implemented tree architecture.
(3) We analyze dual B+tree, with random workloads, to understand its performance and new functionality in the context of improving the run time.
(4) We implemented an outlier detection algorithm and a lazy move algorithm to reduce the effect of outliers which would decrease the insertion performance.
(5) We benchmarked the performance for insertion and point queries with nearly sorted workloads from the work load generator with specific noises and windows thresholds.

[1]Project github repository: https://github.com/Jingyi-H/cs561_dual_bptree.git

2 Background

In this section, we provide an overview of B+ tree and Dual B+ tree, including their algorithm and their insertion/deletion run time and the definition of nearly-sorted relations.

2.1 B+ Tree

A B+ tree consists of a root, internal nodes and leaves. The root might be either a leaf of node with two or more children. B+ tree eliminates the search time of a record, compared to the normal B-tree, by storing data pointers only at the leaf nodes. Moreover, the leaf nodes are all linked together to provide range queries to the records.

2.1.1 ***Querying B+ Tree.*** We traverse through the root node to find the pointer that will direct to the lead node that might contain the search key. Once we are on the leaf node, we do a sequential search to find the search key. In average the run time complexity should be $O(log\ n)$.

Algorithm 1 B+ Tree Point Query

```
current node ← root node
while current node != leaf node do
    for i in current node do
        if k < i then
            current node ← left child node
        else if k == i and k < k + 1 then
            current node ← right child node
        end if
    end for
end while
for j in current node do
    if j == k then
        return k
    end if
end for
return NONE
```

2.1.2 ***Properties for insertion B+ tree.*** We discuss the insertion on random key and have it separated into two cases. In average it should have a time complexity of $O(log\ n)$.

Case1: Overflow in leaf node. If the leaf node is equal to the order or the node, we split the leaf node into two nodes. First node

will contains $\lceil (m-1)/2 \rceil$ values where we denote the order as m and second node contains the remaining values. We then **copy** the smallest search key value from second node to the parent node in order to end the insertion process.

Case2: Overflow in non-leaf node. The overflow in non-leaf node works quite different from the leaf node overflow. If the non-leaf node is full. We split the non-leaf node into two nodes. Again the first node should contains $\lceil (m-1)/2 \rceil$ values, and the second node contains the rest. However, instead of copying the smallest search key we **move** it to the parent node.

Algorithm 2 B+ Tree Insertion

```
if bucket is full then
    split the bucket
    move half of elements to new bucket
    copy middle element into the parent
    while (parent is full) do
        split the bucket
        move middle element into parent
    end while
else
    add element to bucket
end if
```

2.2 Nearly-Sorted Relations

2.2.1 ***Definition: Sorted Relation*** A relation of n tuples is said to be sorted if for any two indices i, j, where $1 \leq i \leq j \leq n$, we have $R[i] \leq R[j]$ [1].

2.2.2 ***Definition: k-Close to Sortedness*** A relation is k-close to being sorted when it is possible to remove k tuples from the relation to achieve a sorted relation. Reversely, if a relation is not k-close to being sorted, then we say the it is k-far from being sorted [1].

2.2.3 ***Definition: l-Globally Sorted*** A relation is l-globally sorted when for every pair of tuples that is out of order in R , the difference between the index of the two tuples is not greater than l [1].

2.2.4 ***Definition: (k, l)-Nearly Sorted*** Nearly Sorted relation combines two of the above relation types and captures the notion of being nearly-sorted.

Given a non-negative k and l, we say a relation is k-close to being l-globally sorted if there exists a set of indices I where $|I| \leq k$, so that for any two indices $i, j \in [n]$ where $i \leq j - l$, $i \notin I$ and $j \notin I$ we have $R[i] \leq R[j]$ [1].

3 Dual B+ Tree Designs

The dual B+ tree structure is composed of two B+ trees which are called the "ordered tree" and the "unordered tree". The ordered tree is used to store data that arrives in a "sorted" behavior, and the unordered tree stores the unordered data. To determine the orderness of a data point, we proposes an outlier detection model. When building a dual B+ tree for an nearly sorted workload, we use such model to decide in which tree the data points are to be inserted. For example, if we have a dataset {1, 2, 3, 100, 4}, we should insert the outlier 100 to the out-of-order tree even though it arrives "in order". The design of the dual tree allow us to store almost-sorted workloads in an efficient manner.

3.1 Ordered Tree Insertion Algorithm

When making insertions on an in-order key to the ordered tree, we use a insert_by_max() function that is adapted from the B+ tree algorithm. Since the given key arrives in a sorted order, meaning that it is greater than the current maximum key in the ordered tree, we can confidently insert it to our tail leaf. By only making insertions to the tail leaf, we avoid the cost of locating the leaf to be inserted, which leads to a better efficiency in dual B+ tree insertions. The algorithm of dual B+ tree looks as follows:

Algorithm 3 Dual B+ Tree Insertion Algorithm

```
if key is not outlier then
    sorted_tree.insert_by_max()
else
    unsortedtree.insert()
end if
```

The pipeline of insert_by_max method is as follows: If the insertion does not trigger a leaf node split, we simply add it to the tail leaf. Otherwise, we will do the necessary split on the tail leaf and update the internal nodes.

3.2 Basic Dual B+ Tree

The idea of our Basic Dual B+ Tree was straight forward as we compare the inserted value with the maximum key($tail\_max$) and minimum key($tail\_min$) of sorted tree's tail leaf using two functions, $insert\_by\_max$ and $insert\_by\_min$ functions, where $insert\_by\_max$ directly insert the value into the end of the tail node whereas the $insert\_by\_min$ function requires a full scan of the tail node and insert the value accordingly. If the value is greater than the maximum key, we insert the number into in ordered tree using the $insert\_by\_max$ function. If the value is in between min and max key of the tail node, the $insert\_by\_min$ function will be called. Otherwise, it will be inserted into out-of-order tree. Algorithm 4 and Algorithm 5 shows the pseudo code of the basic dual b+ tree insertion and $insert\_by\_min$ function.

Algorithm 4 Basic Dual B+ Tree Algorithm

```
Require: Key K, Value V
if K > tail_max then
    sorted.insert_by_max(K, V)
else if K > tail_min then
    sorted.insert_by_min(K, V)
else
    unsorted.insert(K, V)
end if
```

Yet, when we consider this workload {1, 2, 3, 16, 17, 18, 4, 5, 6, 7}, and we assume that the order of the node to be 6. Thus after the insertion of 18, the node will splits into two node, where the new tail node has the maximum key of 18 and minimum key of 16. This means that the {4,5,6,7} will all be inserted into the out-of-order tree. Scale up of workload will eventually result in greater size of out-of-order tree. This situation is not ideal for our research

Algorithm 5 Insert to Tail by Tail Minimum

Require: Key K, Value V
  $i = max\_index$
  data = sorted.tail.data
  while $data[i].key > K$ **do**
    $data[i+1] \leftarrow data[i]$ // move the elements backward
    $i \leftarrow i - 1$
    $data[i] \leftarrow pair(K, V)$
  end while

since we want to maximize the size of the in-ordered tree, where it uses the *insert_by_max* algorithm. In order to solve this problem, outlier detection needs to be implemented to prevent such situation to happen when inserting. Calculating the standard deviation of the tail node could be a possible outlier detection algorithm to be integrated into the Dual B+ Tree architecture.

3.3 Outlier Detection Algorithm Using Standard Deviation

We choose to implement the outlier detection algorithm using standard deviation, and we call this method SD. Standard deviation(StdDev) indicates the spreadness of the data, thus should provide a good estimation on whether a key fits in the distribution of the sorted tree. The SD algorithm is as follows:

Algorithm 6 Dual B+ Tree Insertion with SD Strategy

Require: Key K, Value V
1: **if** $K \geq tail\_max$ **then**
2: **if** $K < tail\_max + StdDev$ **then**
3: $sorted.insert\_by\_max(K, V)$
4: $tail\_max = key$
5: $updateStdDev(K)$
6: **else**
7: $unsorted.insert(K, V)$
8: **end if**
9: **else if** $K \geq tail\_min$ **then**
10: $sorted.insert\_by\_min(K, V)$
11: update $tail\_min$
12: **else if** $K < tail\_min$ **then**
13: $unsorted.insert$(K,V)
14: **end if**

The intuition behind the outlier detection algorithm is to create an upper bound for the key value that is allowed to be inserted into the sorted tree, where the bound is calculated by adding standard deviation(StdDev) to the tail max. If key is out of bound or smaller than the tail min, we simply insert it to the unsorted tree; otherwise we either insert the key by tail min or tail max, depending on whether it is greater than tail max.

We update StdDev when we make and update to the tail max. We calculate the StdDev as follows:

Algorithm 7 UpdateStdDev Method

1: $StdDev = (tail\_max - tail\_min)/sqrt(12)$

We choose to calculate uniform distribution standard deviation since its distribution resembles the most to our nearly sorted workloads. By default, we insert the first key of the workload to the sorted tree, and the StdDev value is maintained above $sqrt(12)$. With such design, the StdDev will scale as the sorted tree grow. Note that we calculate StdDev on the tail leaf instead of the entire sorted tree in order to prevent dramatic scaling of the standard deviation, which could leads to an overly big boundary that categorize extremely big outliers to the sorted tree.

3.4 Lazy Move Strategy

In the modified insert algorithm in section 3.2, we add one more case to the base model: when the key to insert K is within the range $(tail\_min, tail\_max)$. We design another strategy for the case. This strategy reduces impacts of outlier by removing the outlier in the sorted tree instead of detecting beforehand.

Algorithm 8 Dual B+ Tree Insertion with Lazy Move Strategy

Require: Key K, Value V
1: **if** $K \geq tail\_max$ **then**
2: $sorted.insert\_by\_max(K, V)$
3: **else if** $K \geq tail\_min$ **then**
4: $outlier\_K, outlier\_V \leftarrow sorted.replaceOutlier(K, V)$
5: $unsorted.insert(outlier\_K, outlier\_V)$
6: **else if** $K < tail\_min$ **then**
7: $unsorted.insert$(K,V)
8: **end if**

The lazy move strategy moves the outlier from the in-order tree to the out-of-order tree. Then a question arises: how to determine an outlier. As depicted in section 3.2, the *insert_by_min* algorithm finds the target slot for K and moves the keys backward from the target slot. The lazy move strategy identifies the key lying in the target slot as an outlier. The pseudo-code is as follows:

Algorithm 9 Lazy Move Strategy

Require: Key K, Value V
1: $i = 0$
2: data = sorted.tail.data
3: **while** $data[i].key < K$ **do**
4: $i \leftarrow i + 1$
5: **end while**
6: $outlier \leftarrow data[i]$
7: $data[i] \leftarrow pair(K, V)$
8: **return** $outlier$

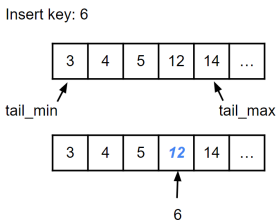


Figure 1: Lazy Move Example

As illustrated in Figure 1, we are going to insert key $K = 6$ to the tail node of the sorted tree. The lazy move algorithm will compare

K with the keys in the tail and find the first element $K' > K$. In this case, 12 is identified as the outlier. Then it will insert K' to the unsorted tree and overwrite K' by K in the tail of the sorted tree. However, it will not do anything to the elements after 12 – in this example, the 14 will remain unchanged. The order of the tail node is guaranteed in such way while the outlier is removed from the sorted tree.

Switching the tree for outlier also creates overhead. The idea of laziness is to remove the fewest elements in the node while decreasing most impacts of outliers. Therefore, the dual b+ tree can benefit from the nearly-sorted workload without moving around a lot of entries between the two trees. Furthermore, the lazy strategy will minimize the effect by some noisy element within the range $(tail\_min, tail\_max)$.

3.5 Benchmark Explanations

To test the dual B+ tree implementation, we generate test data using a workload generator. The workload generator generates the data points with an input size n, a noise level k, and a window threshold l, which allow us to generate nearly-sorted datasets. Specifically, we create a workload with the input size that has the noise level percent of out of order data points where the out-of-order data is placed within a threshold percent window from its original sorted location. By conducting insertions and point queries on the workloads, we compare the run time of the operations between a normal B+ tree and the dual B+ tree.

We have designed three sets of workloads of different sortedness each with 10 million keys where each sortedness is tested three times. The first set is when window k = 5 and l varies from 0, 1, 3, 5, 10, 25, 50; the second set is when l = 5 and k varies from 0, 1, 3, 5, 10, 25, 50; and the third set has (k,l) as (0,5), (1,1), (3,3), (5,5), and (50,50). The design of the experiment tests how our algorithms perform when we scale window size or noise percentage, and provides an overall picture of how much efficiency our implementations give comparing to the normal B+ tree.

4 Results

In this section, we present our experimental results and describe our observations.

In Figure 2, we show the insertion latency of each implementations of Dual B+ tree and also the B+ Tree, which is the control group, for workload of 1 million entries with incriminating noise and window percentage. The noise and window percentage varies from (0,0), (1,1), (3,3), (5,5) up to (50,50).

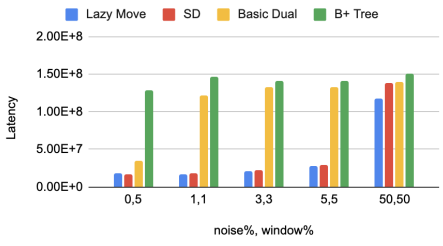
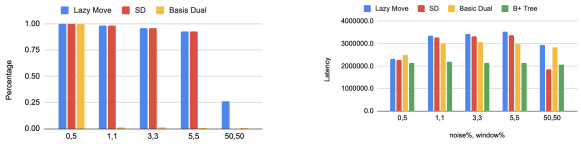


Figure 2: Overall Insertion Performance

It is worth mentioning that the performance of the Lazy Move and Standard Deviation implementations have a better performance in terms of latency compared with Basic Dual B+ Tree and Normal B+ Tree especially when noise level and window threshold are set small. However, when both thresholds were being increased to 50%, the performance decrease gradually for all of them. But still, the Lazy Move strategy has a better performance than normal B+ tree by reducing the latency for up to 22%.

By looking into the sorted tree size of each dual tree implementations (Figure 3(a)), all three dual B+ trees with the the integration of outlier detection algorithm work well on workloads with small noise and window. The percentage of keys that were stored in the sorted tree, in average, is as high as 90% of the entire workload when noise and window threshold are below 5%. However, when both noise and window thresholds become large enough, the SD approach is unable to capture in-order keys efficiently, while lazy move method is still able to capture good amount of sorted keys.



(a) Overall Tree Size **(b) Overall Query Time**

Figure 3: Overall Tree Size and Query Time

We performed basic query for four of the implementations. Since Dual B+ Tree needs to search for both out-of-ordered tree and in-ordered tree, Thus they run relatively slower than the Normal B+ Tree as shown in Figure 3(b).

4.1 Basic Dual B+ Tree

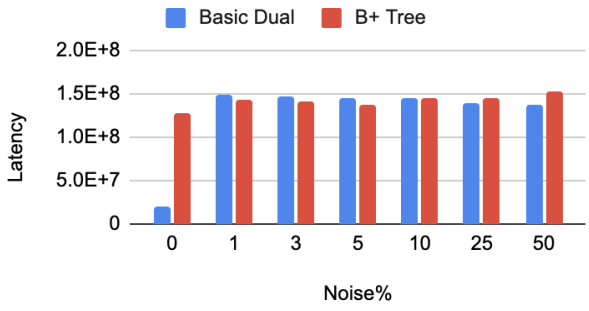


Figure 4: Basic Dual B+ Tree Insertion Performance vs. Noise %

In Figure 4, we set the fixed variable, window threshold, as 5%, and change the noise percentage accordingly. As shown in the figure, when the workload been fetched in is sorted. The insertion time decreases gradually when we compared it with B+ Tree, due to the fact that all of the keys were being inserted into in-ordered tree using insert_by_max() function. Basic Dual B+ Tree outperforms Normal B+ Tree when the noise percentage was high enough. 25% of noise was the bottom line where the Basic Dual B + tree performs

Table 1: Basic Dual B+ Tree Tree Size

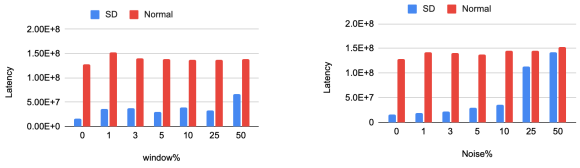
Noise, Window	Sorted Tree Size	Out of Order Tree Size
0,1	100000000	0
1,1	128767	9871233
3,3	98782	9901218
5,5	89818	9910182
50,50	51901	9948099

faster insertion than normal architecture, which demonstrates that it can be relatively more efficient when the workload has a higher degree of orderless.

According to Table 1, only 10% of the total workload was being inserted into the in-ordered tree. Since the outlier of the workload are still been added to the in-ordered tree, just as mentioned in the section 3.2.

4.2 Outlier Detection with Standard Deviation

In Figure 5(a), we show the performance of SD method as window threshold scales. It shows that the SD approach is able to provide better insertion performance than the normal B+ tree when window level changes and noise remains at 5%. Figure 5(b) shows the performance of SD method as noise percentage scales. In this figure, we see that SD latency increases as noise size increases, but when noise is 25% and 50%, its latency dramatically increases.



(a) Noise=5% **(b) Window=5%**

Figure 5: Dual B+ Tree with SD vs. B+ Tree for Insertions

We further look at the size of the percentage of keys that were inserted to the unsorted tree.

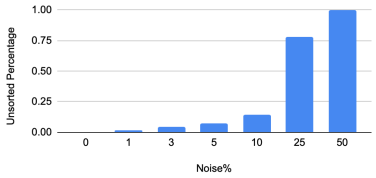
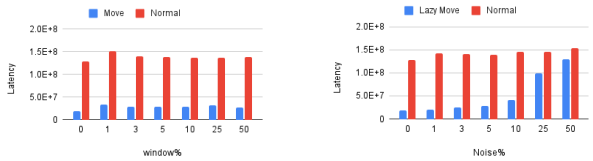


Figure 6: SD Method Unsorted Keys Percentage at window=5%

In Figure 6, we see that the percentage of sorted keys is close to the noise level when noise is lower than 25%, but it significantly increased when noise reaches 25%, which means that when noise level becomes high, SD approach is only capable of identifying a few sorted keys. The reason behind such outcome is that standard deviation is not able to scale elegantly with big noise in the workload. Since we always insert the first key that's read into the sorted tree, we are thus unable to insert any key that is smaller than this key into the sorted tree. When 50% of the keys are out of order, SD is also incapable of scaling smoothly since keys don't arrive in a sorted order, which ultimately prevent even more sorted keys from falling into the detection boundary.

Therefore, the SD approach performs well with workloads that have relatively low noise, but fails to perform ideally when noise becomes big.

4.3 Lazy Move



(a) Noise=5% **(b) Window=5%**

Figure 7: Dual B+ Tree with Lazy Move vs. B+ Tree for Insertion

Figure 7 compares the insertion performance of dual B+ tree with lazy move and a normal B+ tree, which is tested with the same workload as the previous experiments.

It can be observed in Figure 7a that the performance is stable corresponding to the window size, with only a slight increase for window size below 3%. As discussed in section 3.4, noise within a smaller window can lead to frequent moving between the dual tree. However, due to the laziness of the move strategy, the performance is overall stable.

With the move strategy, most of the outliers can be identified and will be removed from the sorted tree. As more keys in the workload will be directly inserted into the tail of the sorted tree – which is much more efficient compared to insertion to unsorted tree – the latency is significantly reduced, as shown in Figure 7b.

5 Conclusion

Lazy move method is able to provide a better performance in every workload, even given a close to completely random workload. When noise level is lower 25%, SD and Lazy Move methods are almost equally efficient, even though SD method fails to effectively detect outliers when noise is higher due to the limited scalability of standard deviation. In conclusion, the move strategy provides a smart way to deal with outliers and effectively reduce insertion latency in varied workloads, and the outlier detection methods with standard deviation is able to provide efficient insertions on workloads with small noise Overall, our implementations of the dual B+ tree structure are efficient design for insertion operations for nearly sorted workloads.

6 Future Work

As discussed previously, the reason for introducing the dual B+ tree is to get most benefits from nearly-sorted workload at inserts. Our work focused on the optimization on insertion to the dual structure. The possible next step is to optimize query design balancing the trade-off between reads and writes for the dual B+ tree.

7 Reflections

We have learned a lot from this project by overcoming challenges and the project have inspired us to come up with new designs. We enjoyed and appreciated the process of researching related backgrounds, designing and testing our algorithms, and analyzing results. We have become much more proficient in writing C++ and become very familiar with the B+ tree algorithms. Working in a team setting to corporate with teammates to design and split the workload is also an experience that we all benefit from.

8 Reference

[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