Journal of Information Systems Engineering and Management
2021, 6(3), em0142
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

A Comparative Study on the Performance of the IB+ *Tree* and the I2B+ *Tree*

Edgar Carneiro [1,2], Alexandre Valle de Carvalho [1,2*], Marco Amaro Oliveira [1]

[1] INESCTEC, Campus da FEUP, R. Dr. Roberto Frias, 4200-465 Porto, PORTUGAL
[2] FEUP, R. Dr. Roberto Frias, 4200-465 Porto, PORTUGAL
*Corresponding Author: alexandre.carvalho@inesctec.pt

Citation: Carneiro, E., de Carvalho, A. V. and Oliveira, M. A. (2021). A Comparative Study on the Performance of the IB+ *Tree* and the I2B+ *Tree*. *Journal of Information Systems Engineering and Management, 6*(3), em0142. https://doi.org/10.21601/jisem/11006

ARTICLE INFO

Received: 17 Sep. 2020

Accepted: 4 Mar. 2021

ABSTRACT

Index structures were often used to optimise fetch operations to external storage devices (secondary memory). Nowadays, this also holds for increasingly large amounts of data residing in main-memory (primary memory). Within this scope, this work focuses on index structures that efficiently insert, query and delete valid-time data from very large datasets. This work performs a comparative study on the performance of the Interval B+ tree (IB+ tree) and the Improved Interval B+ tree (I2B+ tree): a variant that improves the time-efficiency of the deletion operation by reducing the number of traversed nodes to access siblings. We performed an extensive analysis of the performance of two operations: insertions and deletions, on both index structures, using multiple datasets with growing volumes of data, distinct temporal distributions and tree parameters (time-split alpha and node order). Results confirm that the I2B+ tree globally outperforms the IB+ tree, since, on average, deletion operations are 7% faster, despite insertions requiring 2% more time. Furthermore, results also allowed to determine the key factors that augment the performance difference on deletions between both trees.

Keywords: data structure, indexing, B+ tree, time intervals, temporal data, performance analysis

INTRODUCTION

The classic performance analysis of index structures had its focus on disk access optimisation. Therefore, the evaluated parameters were often related to the primary goal of minimising the number of disk access operations (Cudre-Mauroux et al., 2010; Mahmood et al., 2018). Nowadays, index structures are required by new and distinct application domains, usually involving large datasets, where high-speed access to information is mandatory. Hence, in these new circumstances, evaluating disk access optimisation might no longer make sense, while the study of the performance of index structures as a function of the volume of data appears as a more adequate analysis.

The cost of both primary and secondary memory storage space has been consistently decreasing, thus allowing larger amounts of data to be captured and stored (Liu & Yuan, 2019). As a consequence, applications are required to deal with these larger amounts of data, making the space overhead related to the data structure less of a concern, with the time-efficiency gathering a more prominent role. Hence, access to data from ever-growing datasets should be as time-efficient as possible.

When considering temporal data, the *Interval B+ tree* (*IB+ tree*) is a fast data access method for the efficient handling of interval-based valid-time information (Bozkaya and Ozsoyoglu, 1998). From the performed literature review, where we examined several index structures capable of indexing valid-time information, the *IB+ tree* emerged as the most promising one. However, from the more recent work employing this index structure (Guo et al., 2014; Lock and Booss, 2010; Vutukuri, 2018), only Carneiro et al. (2020) provided an analysis of the *IB+ tree* with a focus on performance on growing volumes of data. In their work, Carneiro et al. introduce the *Improved Interval B+ tree* (*I2B+ tree*), a time-efficiency focused variant of the *IB+ tree*. This variant differs from the original index structure by reducing the number of nodes traversed in the deletion of a stored object.

Hence, in this work, we extend the work of Carneiro et al. (2020) by conducting a comparative study on the performance of the *IB+ tree* and the *I2B+ tree*. We start by analysing the main differences between both index structures and examine how those differences can impact the performances of each tree. Afterwards, we benchmark both index structures, with a focus on insertions and deletions (including single deletions and range deletions), in a multitude of scenarios and configurations. The goal is to understand if the conceptual improvement of the *I2B+ tree* corresponds to a *de facto* significant performance increase.

Copyright © 2021 by Author/s and Licensed by Veritas Publications Ltd., UK. This is an open access article distributed under the Creative Commons Attribution License which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Tested *TypeScript* open-source implementations of both the *I2B+ tree* and the *IB+ tree* are provided with the purpose of using them in the experiments and making them available for client-side applications. This choice also took into consideration the current trend of platform-independent, browser-based applications and its increased access through mobile devices.

The paper is organized as follows. Section *Related Work* summarises different structures for indexing valid-time information and presents an analysis of the *IB+ tree*. Section *I2B+ Tree* & *IB+ Tree* presents the main differences between the *I2B+ tree* and the *IB+ tree*. Section *Experiments, Results and Discussion* describes the experiments performed, as well as examines the results obtained. Lastly, Section *Conclusion* provides conclusions and identifies future work.

RELATED WORK

In this section, we identify the main index structures that are presented as capable of handling time intervals (valid-time domain). From this set of index structures, we identified the *IB+ tree* as the most promising for obtaining improved time-efficiency on growing volumes of data. Thus, details of this index structure are presented.

Valid-Time Index Structures

Through a systematic literature review on valid-time index structures, four main categories can be identified: spatial indexes storing bounding intervals in a single dimension; *B+ tree* variants; Interval tree augmentations; and others (e.g., *MPB-tree* (He et al., 2013)).

One way commonly used to represent unidimensional spatial indexes of algorithms are one-dimensional *R-trees*. Mahmood et al. (2018) and Valdés and Güting (2017) both use a one-dimensional *R-tree* for handling temporal data in their spatiotemporal frameworks. *R-trees* key idea consists of grouping objects together using a bounding interval (in one-dimensional data) and using that bounding interval to represent the group in lower depth nodes (Guttman, 1984).

Regarding *B+ trees*, there are many variants for handling temporal data. Among others, we can highlight Time Index (Elmasri et al., 1990); *IB+ tree* (Bozkaya and Ozsoyoglu, 1998); and MAP21 (Nascimento and Dunham, 1999). The Time Index comprises an access structure for temporal data, based on a versioning approach. The *IB+ tree* consists of augmenting the *B+ tree* so that the tree nodes manage interval information similarly to the Interval-tree. Moreover, Nascimento and Dunham (Nascimento and Dunham, 1999), using the approach MAP21, show how a *B+ tree* can be adapted to support the indexing of intervals by mapping the two values constituting the range into a single value.

Interval-tree (de Berg et al., 2008) augmentations represent the adaptation of a balanced tree structure to support intervals in the manner defined by the Interval tree. Carvalho et al. (2014) work is an example of augmentation by using a Red-Black Augment Interval Tree. Thus, the authors made a red-black tree capable of handling valid-time intervals. The *IB+ tree*, besides being a *B+ tree* variant, is also an example of an Interval-tree augmentation.

In the above-mentioned *others* category, we include other structures that do not belong in any of the previous categories. The Multi-dimensional Parallel Binary Tree (He et al., 2013) is an example of such. In this spatiotemporal index structure, the temporal dimension is managed through a triangular binary tree using a triangular decomposition strategy to handle the representation of temporal intervals.

Mahmood et al. (Mahmood et al., 2019), demonstrates that, for the majority of the structures, the temporal dimension is handled using a *B+ tree* variant. Regarding the comparison of some of the index structures presented above, Bozkaya and Ozsoyoglu present the benefits regarding node accesses when comparing the *IB+ tree* to the one-dimensional *R-tree* (Bozkaya and Ozsoyoglu, 1998). Henceforth, we provide a more in-depth analysis of the *IB+ tree*, since this index structure was brought to our attention, in our literature review process, as the most promising for handling valid-time intervals.

Interval B+ Tree

The *IB+ tree* consists of a time-efficient index structure that merges the principles of both *B+ trees* (Comer, 1979) and Interval-trees (de Berg et al., 2008). In more detail, it consists of an augmentation of the *B+ tree* (an N-ary tree), where each node contains the same kind of information as on Interval-trees. In this structure, there are two types of nodes: internal nodes, whose children are other nodes, and leaf nodes, whose children are intervals. Each node stores three lists: one list containing its children, another containing the ordered node keys and the last containing the maximums. Within this context, according to Bozkaya and Ozsoyoglu (1998), a key is the smallest lower bound of the respective children of the first list. Similarly, a maximum is the highest upper bound of the respective children of the first list. The order imposed by the keys sorts all lists.

Since the underlying structure is a *B+ tree*, each leaf node will contain a pointer to the right sibling. Intermediate nodes contain no pointers for any of the siblings. In the context of the *B+ tree*, a key represents a literal. Furthermore, in *B+ trees*, the insertion and deletion of nodes can lead to readjustments on the overall structure of the tree.

In the case of insertions, if a node is accommodating a new key, but the accommodation leads to the number of keys exceeding the number of allowed keys per node, the **node splits**. Splits are operations that consist of dividing the keys of a node into two new nodes. Therefore, each new node is expected to contain half the keys from the original node. After, the tree proceeds with the insertion of the new key in the node that should save it.

Reversely, the removal of a node can also trigger a rebalance of the tree. When a key is removed from a node, if the number of keys stored in the node is not bigger than half of the maximum number of keys it can store, then the tree must readjust. First, the node tries to **borrow a key** from one of its siblings. The borrowing happens if either one of the siblings contains more than half of the maximum number of keys it can store. Otherwise, the borrowing would lead to one of the siblings being unable to satisfy the minimum of stored keys condition. In such cases, where no borrowing is possible, the **nodes are merged** with one of its siblings, creating a new node that contains the keys from the merged nodes. These are

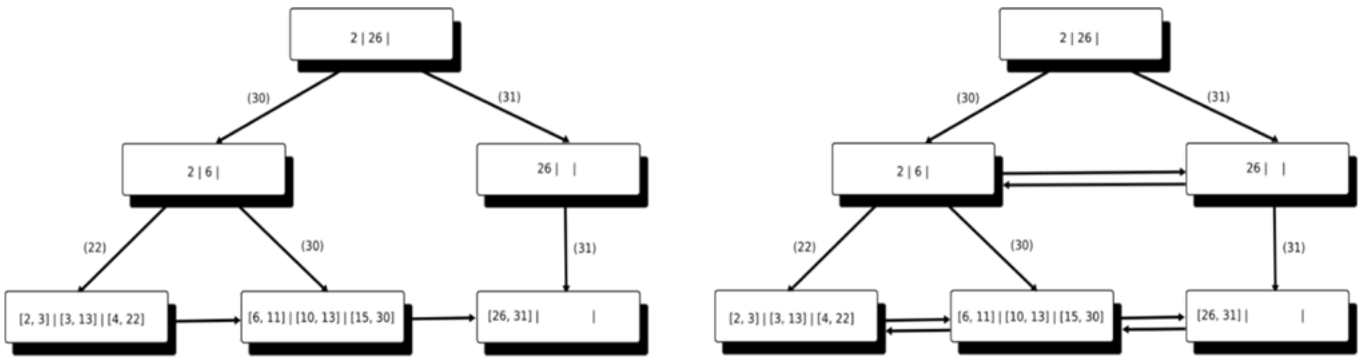


Figure 1. Structural differences between the *IB+ tree* (left) and the *I2B+ tree* (right), while indexing the same dataset

called borrow and merge operations. Consequently, other kinds of tree readjustments can occur, but we do not describe them since they do not add value to the current work. Comer's work (Comer, 1979) presents a more in-depth analysis of the entirety of these cases.

The augmentation of the Interval-tree (de Berg et al., 2008) follows some basic principles: 1) each node stores an interval, where the interval lower bound represents the key of the node - consequently, by travelling the tree in its in-order, we obtain the set of intervals, sorted by the lower bound; 2) each node also stores the maximum higher bound existent in its subtree.

Bozkaya and Ozsoyoglu (1998) also present an enhancement that allows the *IB+ tree* a more time-efficient performance. This enhancement is the **time-split** operations of intervals. This time-split enhancement consists of finding an optimum upper bound (the split point) from the intervals managed by a leaf node and split the children intervals which upper bound surpasses the split point, at that split point. For instance, consider an interval [*a*, *c*] and a split point *b*, where $a < b < c$. Then, interval [*a*, *c*] would split and generate intervals [*a*, *b*] and [*b*, *c*], with [*b*, *c*] being reinserted in the structure. The motive behind time-splits is to avoid long intervals that negatively impact structure performance.

The *IB+ tree* has two user-definable parameters: the nodes' ***order*** and the time-split ***alpha***. The *order* parameter defines the maximum number of children that a node can have. The *alpha* parameter is an empirical factor ($0 < alpha < 1$) that influences the choice of the split point for the children intervals of a leaf node. This parameter adjusts the space/query-time trade-off. Higher *alpha* values lead to higher split point values and, consequently, fewer time-splits occur, thus leading to less storage and decreased query-efficiency. Conversely, smaller *alpha* values lead to smaller split point values and, therefore, to the occurrence of more time-splits and an increase in both storage and query-efficiency. Bozkaya and Ozsoyoglu (1998) present a more detailed explanation of the time-split algorithm and the impact of the *alpha* factor.

I2B+ Tree & IB+ Tree

In this section, we compare the differences between the Improved Interval B+ tree (*I2B+ tree*) and the original Interval B+ tree (*IB+ tree*) and present the algorithms used for getting a node's sibling in each of the index structures. Moreover, we also determine the various factors that determine the performance differences on both index structures.

Main differences

As presented in the works of Carneiro et al. (2020), the *I2B+ tree* differs from the original *IB+ tree* by having every tree node storing pointers to both of its siblings, independently of it being a leaf node or an intermediary node. **Figure 1** shows the difference between both index structures by illustrating the distinct tree configurations when storing the same example dataset.

As seen in Section *Related Work*, the borrow and merge operations, involved in node deletions, serve the purpose of rebalancing the tree. Applying these operations in a node requires finding one of the node's siblings. However, in the original *IB+ tree*, for a node to proceed with either of the mentioned operations, the ancestor node that is the root of the smallest sub-tree containing the node and its sibling must be found. In the most extreme case, the ancestor of both nodes is the tree root, and finding the node sibling implies travelling to the root and back. By utilising pointers to both siblings, Carneiro et al. focus on making the original index structure more time-efficient, by discarding the need to find the ancestor of a node and its sibling. Thus, this approach optimises borrow and merge operations, which consequently improves the deletion operation time-efficiency.

In the *I2B+ tree*, storing in every node pointers to the node's siblings implies an overhead on insertions of having to set the object properties corresponding to the siblings. Additionally, on deletions, to maintain this information, when a given node disappears, the node siblings have to be updated to point to each other, instead of pointing to the node that was removed.

Regarding our implementations of the *I2B+ tree* and the *IB+ tree*, these differ significantly in the implementation of the *findLeftSibling* and *findRightSibling* methods. The *findRightSibling* implementation differs on the tree internal nodes. Below, and with the intent of better illustrating the differences between both index structures, we present the implementation of the method *findLeftSbiling* on the two index structures. An example implementation of the *IB+tree* *findLeftSibling* method consists of:

Similarly, an example implementation of the I2B+tree *findLeftSibling* method consists of:

```
private findLeftSiblingAux(currentDepth: number, isAscending: boolean):
IBplusNode<T> | null {
    if (isAscending) {
        // If is ascending set find a parent that has a left sibling
        let idxInParent: number = this.findIndexInParent();
        // No parent, so no left sibling
        if (idxInParent == null)
            return null;

        if (idxInParent >= 1)
            // It has a left sibling
            return this.parent.getChildren()[idxInParent -1]
                .findLeftSiblingAux(currentDepth, !isAscending);
        else
            // RECURSIVE: Ascend until finding a node with a left sibling
            return this.parent.findLeftSiblingAux(currentDepth + 1, isAscending);
    } else {
        // If is descending find the right most node at the same depth
        if (currentDepth == 0)
            return this;
        else {
            // Did not reach equal depth yet, descend to the rightmost child
            let children = this.getChildren();
            return children[children.length - 1]
                .findLeftSiblingAux(currentDepth - 1, isAscending);
        }
    }
}
```

As shown in the algorithms above, the IB+ tree method consists of a call to a recursive function that will repeat itself $2 * (D_A - D_N)$, where $D_A - D_N$ represents the number of depth levels between the common ancestor node (A) and the node finding the sibling (N). Conversely, the I2B+ tree method consists of obtaining an object property.

```
findLeftSibling(): IBplusNode<T> | null {
    return this.leftSibling;
}
```

Key factors impacting performance differences

Conceptually, the *I2B+ tree* significantly outperforms the *IB+ tree*, since the overall process of finding the sibling node is more time-consuming in the *IB+ tree*. For finding the sibling in the *IB+ tree*, the function responsible for finding it might have to repeat itself a considerable number of times. The higher the number of tree depth levels, the higher the average number of recursive calls the function may have to go through when deleting an interval. Thus, it is expected that deeper trees lead to more dissimilar performances when comparing both index structures.

The depth of a tree is impacted by three factors: (1) the number of intervals that the tree stores, (2) the node's order and (3) the time-split alpha. The last two factors are both tree parameters presented in section II. Regarding the number of intervals being inserted, a higher number of intervals requires more nodes to store them, thus leading to trees with higher depth. Regarding the node's order, if this parameter is set to a high value, every node becomes capable of storing more elements, thus being necessary fewer nodes to store the entirety of the inserted intervals, hence leading to a tree with fewer depth levels. Moreover, high order values also lead to nodes less frequently needing to borrow or merging with siblings, as each node can store more keys. On the contrary, if the node order is set to a small value, more nodes are necessary to store the entirety of the inserted intervals, thus leading to a tree with a higher number of depth levels. Smaller order values also lead to nodes more frequently having less than half of the order value (thus leading to borrows or merges), as each node stores fewer keys. Regarding the time-split alpha, trees employing time-splits (alpha > 0) store a higher number of intervals since these can be time-split. Hence, as already observed, by having a higher number of intervals stored, trees will have higher depth. Considering the different values the alpha parameter can assume, as introduced in Section *Related Work*, smaller alpha values lead to smaller split points, thus increasing time-split occurrences. Therefore, smaller alpha values, greater than zero, lead to higher depth trees.

The number of levels necessary to manage any given number of intervals can be obtained using $\boldsymbol{ceil}(\mathbf{log}_{\boldsymbol{o}}\,\boldsymbol{k})$ where $ceil()$ represents a function for ceiling a decimal number, o represents the tree order and k represents the number of intervals to be inserted.

EXPERIMENTS, RESULTS AND DISCUSSION

This section starts by describing the datasets used in the experiments. Afterwards, it details the experiments performed for comparing the *I2B+ tree* and *IB+ tree* performances. These experiments focused on both insertions and deletions. Each experiment includes a discussion section that interprets the results achieved.

Synthetic Datasets

To compare the performances of the *I2B+ tree* and of the *IB+ tree*, we proceeded with the generation of synthetic datasets. To assure that the generated datasets constituted viable test scenarios, we followed the recommendations presented by Theodoridis et al. (1999), meaning that we make use of mathematical data distributions for generating the synthetic datasets. Each dataset stores a fixed number of intervals. Each interval is created using two data distributions: one for computing the starting timestamp of the interval and the other for the interval duration. For creating the initial timestamps, the synthetic generator makes use of a uniform distribution, and for the duration, it uses a Poisson distribution (Katti and Rao, 1968).

Two main scenarios, with different characteristics, were created and subsequently analysed. In both scenarios, the same uniform distribution was kept for the initial timestamps: on average, at each time unit (an instant at which a time interval might begin), 10 new intervals start. However, regarding intervals duration, the two scenarios differ. In the first scenario, intervals have an average duration of 7 time units and a standard deviation of 2 time units. In the second scenario, the average duration is 1095 time units, and the standard deviation is 200 time units. By proceeding this way, the goal is to obtain two different scenarios where the number of concurrent (temporally overlapping) intervals differs significantly. Hence, we generated a first scenario containing sparse data and a second scenario containing dense data. Given the respective characteristics of each scenario, we named them accordingly: *small intervals scenario* in the first case and *big intervals scenario* for the second one. Both the nomenclature and the dataset generation methodology are exactly the same as the ones employed by Carneiro et al. (2020).

For each of these scenarios, we generated ten datasets of sequential doubling size: we started at a dataset with $1k$

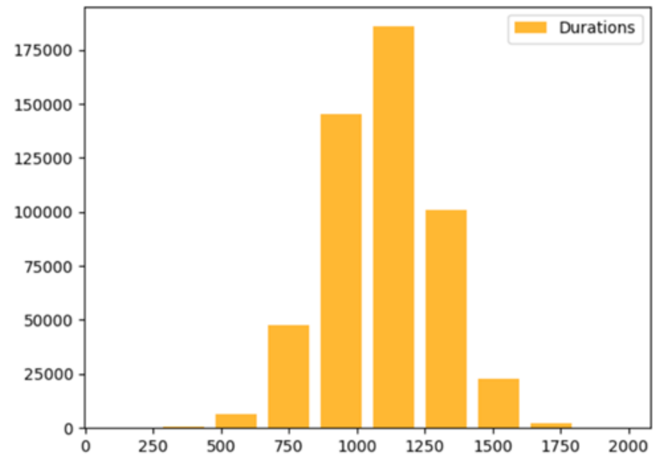


Figure 2. Example of a distribution of the durations of the intervals

intervals, then 2*k*, then 4*k*, and so on up until 512*k* intervals. **Figure 2** presents, as an example, a visual representation of the distribution of the interval durations, for the dataset with 512*k* entries in the *big intervals scenario*. The horizontal axis presents interval durations while the vertical axis presents the number of items.

Experimentation

Experiments were conducted for the three basic operations: insertions, queries and deletions. However, query tests are not detailed as the two index structures perform identically. This behaviour is expected seeing that the differences between the *I2B+ tree* and the *IB+ tree* do not interfere with queries. Moreover, the implementation of this operation in both index structures is the same.

Regarding insertions, two different tests were performed. The first insertion test, named tree insertion (T), consists of evaluating the average time it takes to construct the totality of the tree, given an input test dataset. The second insertion test (I) consists of evaluating the average time it takes to insert 100 randomly chosen intervals of the same test dataset.

Regarding deletions, three distinct tests were performed. The first (D) consists of evaluating the average time it takes to delete 100 randomly chosen intervals stored in a tree that has been previously loaded with an entire test dataset. The second, named ranged deletion (RD), consists of evaluating the average time it takes to range delete 100 randomly chosen intervals stored in a tree filled with the same previous test dataset. In this context, a range deletion consists of deleting all intervals that are fully contained by the provided interval. The third deletion test, named delete half tree (DH), consists of range deleting half of the intervals stored in the tree, by utilising a single range deletion.

Considering the results obtained by Carneiro et al. (2020), we extend on their evaluation by studying the index structures' performance using trees with the order parameter set to 4, 10 and 20, and the time-split alpha parameter set to 0 and 0.2. The order choices, namely 10 and 20, are based on Carneiro et al. findings. The order value of 4 is chosen with the intent of testing trees with more depth levels, following the rationale presented in section III. The alpha choices allow us to test the index structures both when allowing and not allowing time-splits.

To be statistically significant, we used *kruonis*[1]. With this open-source tool, each test ran between approximately 50 and 100 times (being non-deterministic as tests have a limited amount of time to run). The experiments ran in a *Node.js*[2] environment and were done in a laptop running a macOS distribution, powered by a 2.3 GHz Dual-Core Intel Core i5 and 8GB of RAM.

Insertion Comparison

In this experiment, we compare the performances of the two index structures, in the tests concerning the insertion operation (T and I). **Figure 3** shows the results obtained, in the *small intervals scenario*, using an alpha parameter of 0.2. We show these results as they are representative (the identified behaviours are similar) of the results obtained using the other alpha parameter and the *big intervals scenario*. From the analysis of **Figure 3**, we verify that trees with order set to 4

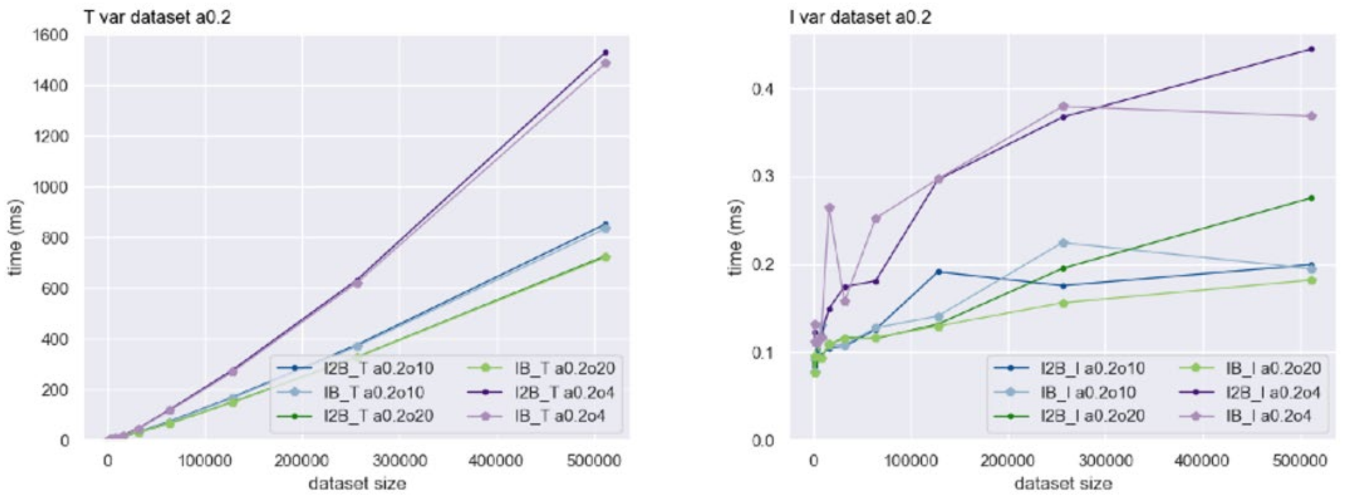


Figure 3. Tree construction test (left) and Insertion test (right), for the *small intervals scenario*, using an alpha parameter of 0.2

[1] https://github.com/most-inesctec/kruonis
[2] https://nodejs.org

take more time to construct (approximately 1500ms for the 512*k* dataset), while trees with order set to 20 are the fastest to construct (approximately 712ms for the 512*k* dataset). The results obtained in the intervals insertion test (I) corroborate the results obtained in the tree construction test (T), with the trees with the order parameter set to 4 having the worst performance (approximately 0.4ms for the 512*k* dataset) with the remaining trees, with high order values, having improved performance (average of 0.2ms for the 512*k* dataset).

Neither of the performed tests allowed us to conclude regarding the comparison of the *I2B+ tree* with the *IB+ tree*. In the tree construction test, the y-axis scale makes it impracticable to infer about the performance difference of both trees. In the Insertion test, there is not a prominent pattern, when considering the performance difference. Therefore, we opted for plotting the ratio between the *I2B+ tree* performance and *IB+ tree* performance, in the tree construction test. Moreover, we also computed and plotted a trendline using the moving average (Goyal, 2009), with a period of 4, to better understand the ratios tendency. For the remaining of this work, we refer as ratio plots to plots employing similar principles to the ones described. **Figure 4** shows the ratio plots, for the different orders (the first row of plots consists of order 4 results, the second row of order 10 results and the third row of order 20 results), in both scenarios (the left column of plots consists of the *small intervals scenario*, while the right column consists of the *big intervals scenario*), for the tree construction test (T).

From the analysis of the plots displayed in **Figure 4**, we verify that all moving average lines are above the red line indicating a ratio of 1. Moreover, all moving average lines revolve around the range of 1.01 to 1.03, with 1.02 being the most common value. We also identify a tendency for high order values to have lower ratios, while low order values appear to have higher ratios (ratios on order 20 plots are approximately 1.01, on order 10 plots are approximately 1.02, and on order 4 plots are approximately 1.03).

Interpreting the results described in this experiment, trees with low order values having worse insertion performance (I and T) are justified by the necessary procedures to create a higher number of internal nodes. Regarding the results obtained in the ratio plots, we can conclude that the *IB+ tree* outperforms the *I2B+ tree* on insertions. The results indicate that, on average, the *IB+ tree* is 2% more time-efficient on insertions than the *I2B+ tree*. The difference between ratio values on trees with distinct order parameter values is justified by higher-order trees having to manage more nodes, and, consequently, of maintaining pointers to the sibling nodes.

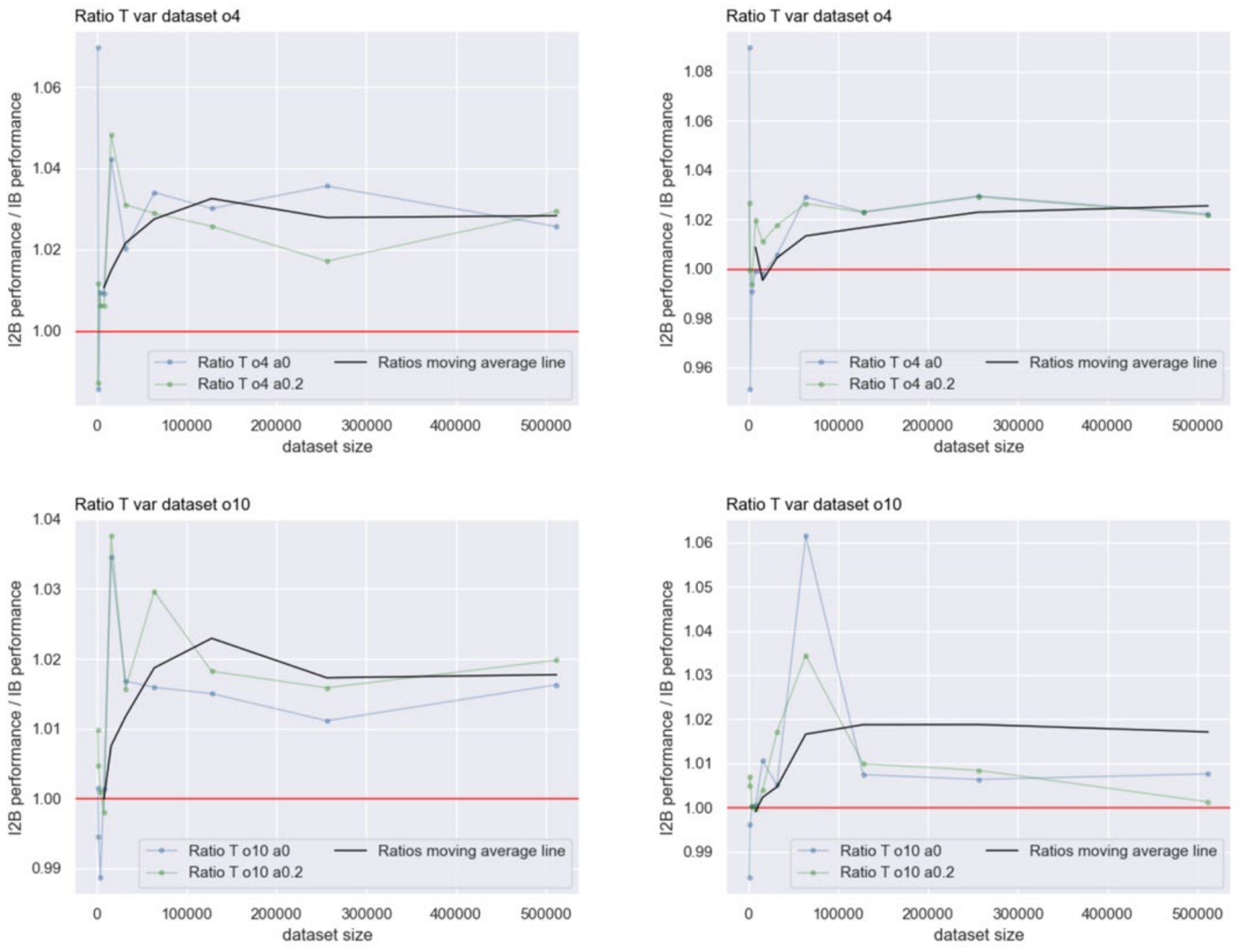
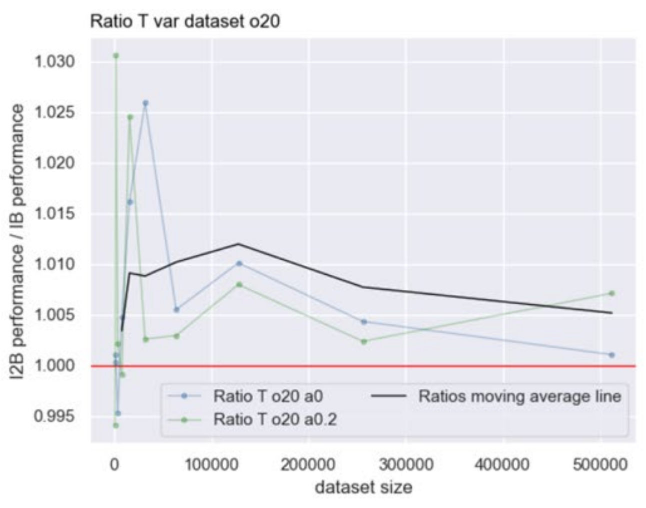


Figure 4. Ratio plots of the tree construction test (T), for the different orders (4, 10 and 20), in both scenarios



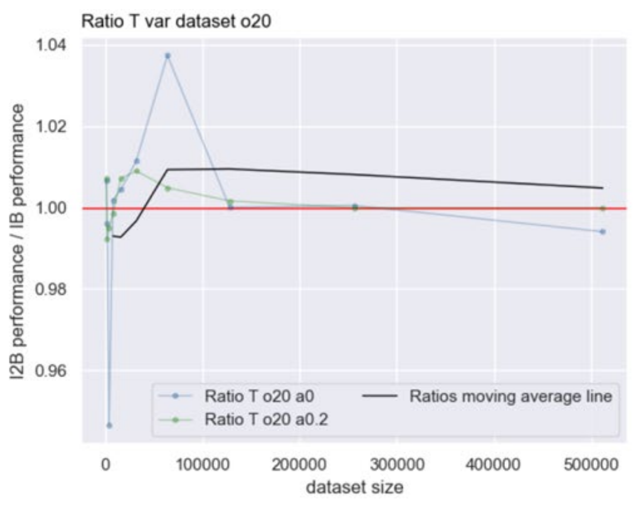
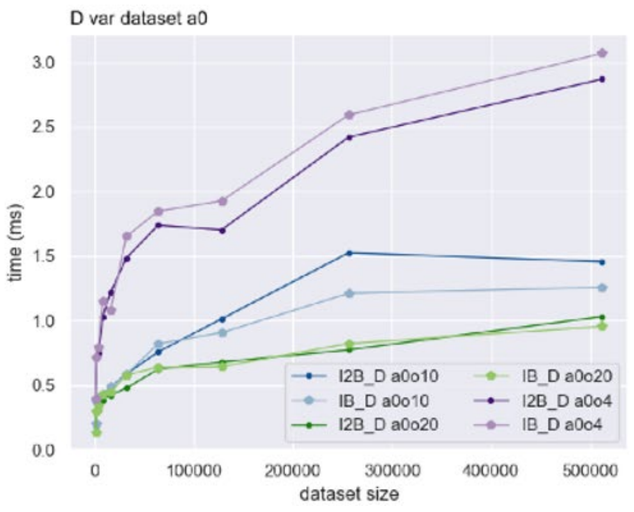
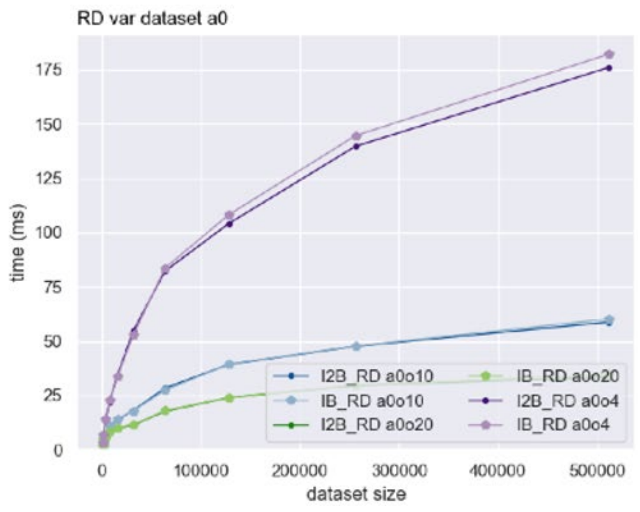


Figure 4 (Continued). Ratio plots of the tree construction test (T), for the different orders (4, 10 and 20), in both scenarios





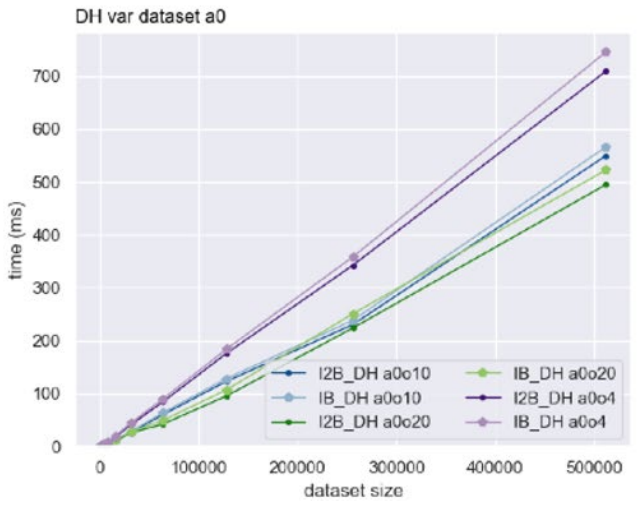


Figure 5. Deletion tests (D, RD and DH), in the big intervals scenario, using an alpha of 0

Deletion Comparison

In this experiment, we compare the two index structures regarding their performances on deletions (D, RD and DH tests). **Figure 5** shows the results obtained, in the *big intervals scenario*, using an alpha parameter of 0 (zero), i.e., no time-splits. Once more, we present these results as they are representative of the results obtained using an alpha parameter of 0.2 and the *small intervals scenario*. From **Figure 5**, we verify that the trees with the order parameter set to 20 are the fastest on all deletion tests, with an average (between the *I2B+ tree* and *IB+ tree* results) of 1.0ms on D, 34ms on RD and 508ms on DH, for the 512*k* dataset. On the contrary, the trees with the order parameter set to 4 are the slowest on all deletion tests, with an average of 2.9ms on D, 183ms on RD and 726ms on DH, also in the 512*k* dataset. Moreover, by observing

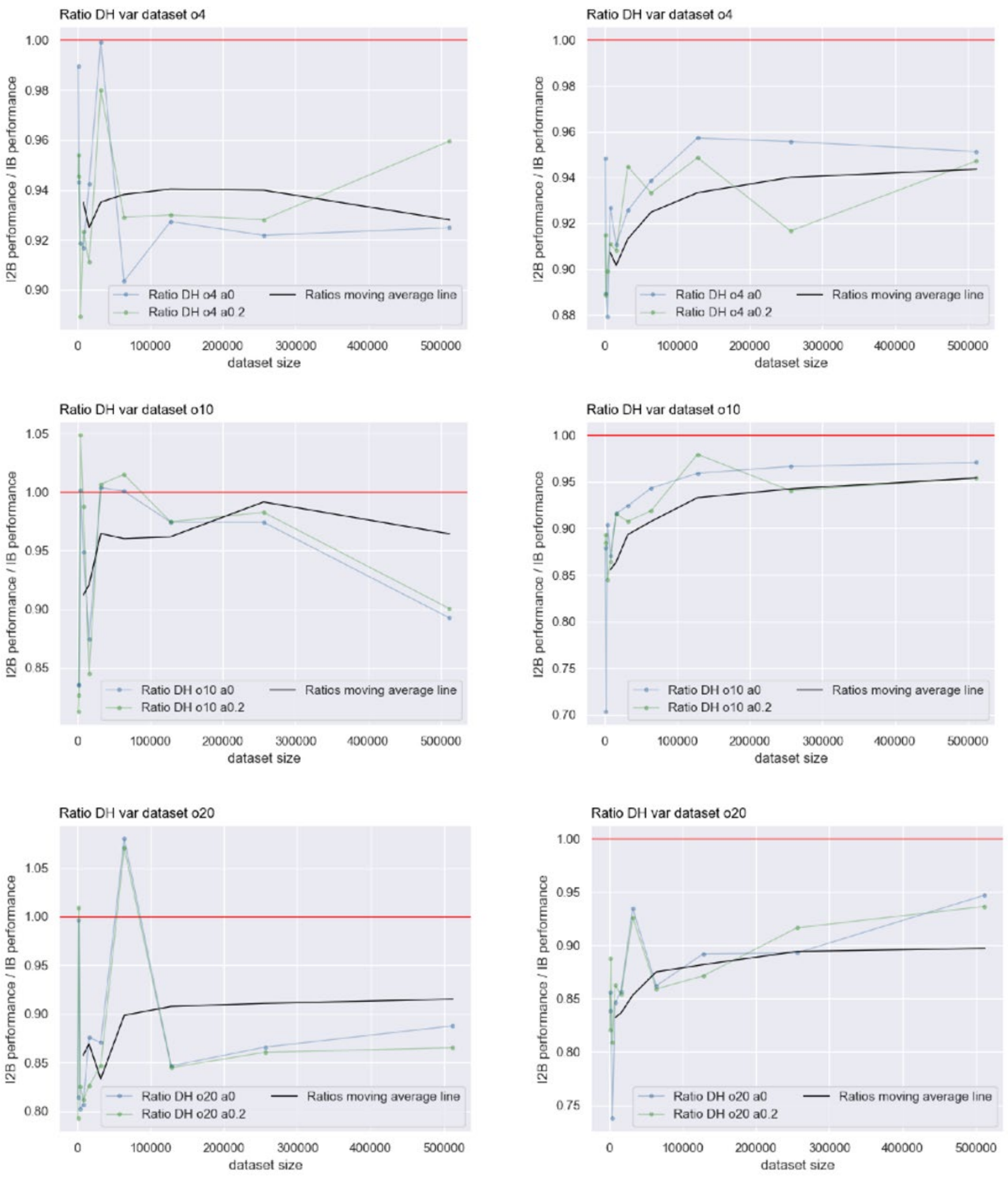


Figure 6. Ratio plots of the delete half tree test (DH), for the different orders (4, 10 and 20), in both scenarios

the obtained results we can identify the *I2B+ tree* as being more time-efficient than the *IB+ tree*.

However, to achieve more precise conclusions, a more detailed analysis of the comparison of both index structures on deletions is performed. Similarly, to the previous experience, we once more employ ratio plots to analyse the results obtained in the tests that consist of deleting half of the tree (DH). **Figure 6** shows the ratio plots, for the different orders (the first row of plots consists of order 4 results, the second row of order 10 results and the third row of order 20 results), in both scenarios (the left column of plots consists of the *small intervals scenario*, while the right column consists of the *big intervals scenario*).

From the plots displayed in **Figure 6**, we verify that, independently of the scenario, all moving average trend lines are below the line indicating a ratio of 1 (red line). In the first row of the ratio plots (**Figure 6**), on trees with the order parameter set to 4, we verify that the ratio values mostly vary between 0.92 and 0.94. On the second row of plots (order of 10), on the *small intervals scenario*, the ratio varies from 0.92 to 0.99, with 0.96 being the most common value. Conversely, on the *big intervals scenario*, the ratio fluctuates between 0.86 and 0.95 and tends to 0.95. On the third row of plots (order of 20), the ratio values range between 0.84 and 0.91 and stabilise on 0.90.

Interpreting the results described in this experiment, trees with low order values having worse deletion performance on all tests are justified by each node storing less information (maximum of 4 keys). As a consequence, trees are deeper, seeing that there is an increase in the number of nodes. Using the formula presented in Section *I2B+ Tree* & *IB+ Tree* on the 512*k* dataset, the trees with order 4 need a total of 10 depth levels, while trees with order 10 need 6 depth levels and trees with order 20 need 5 depth levels. Consequently, the disparities between the index structures are more prominent on trees with order 4, since fewer keys on each node lead to more borrow and merge operations, which in average take more time to conclude as the tree is deeper. From the results shown in **Figure 6**, we conclude that the *I2B+ tree* regularly outperforms the *IB+ tree* on deletions. On average, for the performed tests, the *I2B+ tree* takes 0.93 of the time it takes the *IB+ tree* to perform the same deletion. Hence, we can infer that, for the performed tests, the *I2B+ tree* is, on average, 7% more efficient on deletions than the *IB+ tree*. Variations of the 0.93 ratio depend on the dataset characteristics and the tree order parameter choice. Moreover, we also conclude that the usage of time-splits has no perceptible impact in the ratios of the two index structures, despite the usage of time-splits leading to an increase of the number of stored intervals (as a consequence of the storage of the *CompoundIntervals* presented by Carneiro et al. (2020)).

With the outcomes obtained from the analysis of both the insertion comparison experiment (Section *Insertion comparison*) and the deletion comparison experiment (Section *Deletion comparison*), we can conclude that the *I2B+ tree* globally outperforms the *IB+ tree* when considering time-efficiency: despite the *IB+ tree* being, on average, 2% more efficient on insertions, the *I2B+ tree* is, on average, 7% more efficient on deletions. Thus, in usage scenarios where a considerable amount of insertions, queries and deletions occur, we show that the performance improvements provided by the *I2B+tree* overcome the ones granted by the *IB+ tree*.

CONCLUSION

In this work, we studied the fundamental differences between the *IB+ tree* and the *I2B + tree*. Furthermore, we determined how different order parameter values can impact and accentuate the performance difference between both trees. To analyse the index structures performance differences, we carried out a comparison study, using multiple datasets with growing volumes of data, in two distinct scenarios (one depicting sparse data and the other dense data), with various tree parameters configurations (the time-split alpha parameter was set to 0 and 0.2, while the order parameter was set to 4, 10 and 20). Results showed that, on insertions, the *IB+ tree* is, on average, 2% more time-efficient than the *I2B+ tree*. However, on deletions, we verified that the *I2B+ tree* is, on average, 7% more time-efficient. Hence, when the concern is the index structure time-efficiency, the *I2B+ tree* appears as the best alternative, from the ones studied. An open-source implementation of the *I2B+ tree* is available at *https://github.com/most-inesctec/I2Bplus-tree, while the IB+ tree* is available at *https://github.com/most-inesctec/IBplus-tree*. Future work includes (1) benchmarking the *I2B+ tree* with other valid-time index structures, as a function of the volume of data, and (2) the study of how the *I2B+ tree* fares in the spatiotemporal context, coupled with a spatial structure.

Author contributions: All co-authors have involved in all stages of this study while preparing the final version. They all agree with the results and conclusions.

Funding: This work is financed by the ERDF – European Regional Development Fund through the Operational Programme for Competitiveness and Internationalisation - COMPETE 2020 Programme and by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia within project PTDC/CCI-INF/32636/2017 (POCI-01-0145-FEDER-032636).

Declaration of interest: The authors declare that they have no competing interests.

Ethics approval and consent to participate: Not applicable.

Availability of data and materials: All data generated or analyzed during this study are available for sharing when appropriate request is directed to corresponding author.

REFERENCES

Bozkaya, T. and Ozsoyoglu, Z. M. (1998). Indexing valid time intervals. In G. Quirchmayr, E. Schweighofer, & T. J. Bench-Capon (Eds.), *Database and expert system applications* (pp. 541-550). DEXA 1998. Lecture Notes in Computer Science, vol 1460. Springer, Berlin, Heidelberg. https://doi.org/10.1007/BFb0054512

Carneiro, E., Carvalho, A. V. d. and Oliveira, M. A. (2020). I2B+tree: Interval B+ tree variant towards fast indexing of time-dependent data. *2020 15th Iberian Conference on Information Systems and Technologies (CISTI)*, Sevilla, Spain, 2020, pp. 1-7. https://doi.org/10.23919/CISTI49556.2020.9140897

Carvalho, A. V. d., Oliveira, M. A. and Rocha, A. (2014). *Improvements to efficient retrieval of very large temporal datasets with the TravelLight method.* s.l., 9th Iberian Conference on Information Systems and Technologies (CISTI). https://doi.org/10.1109/CISTI.2014.6876986

Comer, D. E. (1979). Ubiquitous B-Tree. *ACM Computing Surveys, 11*(2), 121-137. https://doi.org/10.1145/356770.356776

Cudre-Mauroux, P., Wu, E. and Madden, S., 2010. Trajstore: An adaptive storage system for very large trajectory data sets. *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*, March, pp. 109-120. https://doi.org/10.1109/ICDE.2010.5447829 de Berg, M., van Kreveld, M., Overmars, M. and Schwarzkopf, O. (2008). *Computational Geometry: Algorithms and Applications*. Springer-Verlag. https://doi.org/10.1007/978-3-540-77974-2

Elmasri, R., Wuu, G. T. J. and Kim, Y.-J. (1990). *The time index: An access structure for temporal data.* Brisbane, Queensland, Australia, 16th International Conference on Very Large Data Bases.

Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1994). *Design patterns: Elements of reusable object-oriented software.* s.l.: Addison-Wesley Professional Computing Series, Pearson Education.

Goyal, M. (2009). *Numerical Methods and Statistical Techniques Using 'C'.* s.l., Laxmi Publications.

Guo, T., Papaioannou, T. G. and Aberer, K. (2014). Efficient indexing and query processing of model-view sensor data in the cloud. *Big Data Research, 1*, 52-65. https://doi.org/10.1016/j.bdr.2014.07.005

Guttman, A. (1984). R trees: A dynamic index structure for spatial searching. *ACM SIGMOD Record, 14*. https://doi.org/10.1145/971697.602266

He, Z., Kraak, M.-J., Huisman, O., Ma. and Xiao, J. (2013). Parallel indexing technique for spatio-temporal data. *International Journal of Photogrammetry and Remote Sensing, 78*, 116-128. https://doi.org/10.1016/j.isprsjprs.2013.01.014

Katti, S. K. and Rao, A. V. (1968). Handbook of the poisson distribution. *Technometrics, 10*(2), 412-412. https://doi.org/10.1080/00401706.1968.10490580

Liu, Q. and Yuan, H., 2019. A high performance memory key-value database based on redis. *Journal of Computers, 14*, 170-183. https://doi.org/10.17706/jcp.14.3.170-183

Lock, H. C. and Booss, D. (2010). *Indexing stored data.* US, Patent No. 7,761,474.

Mahmood, A. R., Aly, A. M., Kuznetsova, T., Basalamah, S. and Aref, W. G. (2018). Disk-Based Indexing of Recent Trajectories. *ACM Transactions on Spatial Algorithms and Systems, 4*(3), 7. https://doi.org/10.1145/3234941

Mahmood, A. R., Punni, S. and Aref, W. G. (2019). Spatio-temporal access methods: a survey (2010 - 2017). *Geoinformatica, 23*(1), 1-36. https://doi.org/10.1007/s10707-018-0329-2

Nascimento, M. A. and Dunham, M. (1999). Indexing valid time databases via B/sup +/-trees. *IEEE Transactions on Knowledge and Data Engineering, 11*(6), 929-947. https://doi.org/10.1109/69.824609

Reinsel, D., Gantz, J. and Rydning, J. (2018). *The digitization of the World – From edge to core,* s.l.: Seagate, IDC Information and Data.

Theodoridis, Y., Silva, J. R. and Nascimento, M. A. (1999). On the generation of spatiotemporal datasets. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and LectureNotes in Bioinformatics), 1651*, 147-164. https://doi.org/10.1007/3-540-48482-5_11

Valdés, F. and Güting, R. (2017). Index-supported pattern matching on tuples of time-dependent values. *GeoInformatica, 21*. https://doi.org/10.1007/s10707-016-0286-6

Vutukuri, T. R. (2018). *Q+ IB+ tree: Indexing technique for moving regions* (PhD thesis), Southern Illinois University.