I2B+tree: Interval B+ tree variant towards fast indexing of time-dependent data

Edgar Carneiro[a,b], Alexandre Valle de Carvalho[a,b], Marco Amaro Oliveira[a,b]
edgar.f.amorim@inesctec.pt, alexandre.carvalho@inesctec.pt, mao@inesctec.pt

[a]INESCTEC, Campus da FEUP,
R. Dr. Roberto Frias, 4200-465 Porto, Portugal
[b]FEUP,
R. Dr. Roberto Frias, 4200-465 Porto, Portugal

Abstract — **Index structures are fast-access methods. In the past, they were often used to minimise fetch operations to external storage devices (secondary memory). Nowadays, this also holds for increasingly large amounts of data residing in main-memory (primary memory). Examples of software that deals with this fact are in-memory databases and mobile device applications. Within this scope, this paper focuses on index structures to store, access and delete interval-based time-dependent (temporal) data from very large datasets, in the most efficient way. Index structures for this domain have specific characteristics, given the nature of time and the requirement to index time intervals. This work presents an open-source time-efficiency focused variant of the original Interval *B+ tree*. We designate this variant Improved Interval *B+ tree* (*I2B+ tree)*. Our contribution adds to the performance of the delete operation by reducing the amount of traversed nodes to access siblings. We performed an extensive analysis of insert, range queries and deletion operations, using multiple datasets with growing volumes of data, distinct temporal distributions and tree parameters (time-split and node order). Results of the experiments validate the logarithmic performance of these operations and propose the best-observed tree parameter ranges.**

***Keywords - Data structure; Indexing; B+ tree; Time intervals; Temporal data; Performance analysis*.**

I. INTRODUCTION

When the majority of index structures emerged, databases were mostly stored in disk drives. Therefore, the classic performance analysis of index structures has its focus on disk access optimisation. Hence, the parameters evaluated are often related to the primary goal of minimising the number of disk access operations [1, 2]. However, nowadays, index structures are required by new and distinct application domains, usually involving large datasets. Its usage has evolved for being employed in a plethora of situations (e.g. client-side spatiotemporal applications, mobile applications that deal with temporal data) where high-speed access to information is mandatory. In these new circumstances, evaluating disk access optimisation might no longer make sense. Conversely, it becomes adequate to evaluate the performance of index structures as a function of the volume of data.

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 [3]. 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 maintain logarithmic performance.

It is a known fact that the rate at which data is being generated nowadays supersedes any before [4]. Thereupon, for these large datasets, it is also necessary to have fast access methods that support the efficient management of information elements, while maintaining its indexed characteristics.

In the widespread domain of spatiotemporal information, data is generally decomposed into spatial data and temporal data. The most common approach to handle data from this multidimensional domain is to employ two separate index structures: one for managing temporal data and the other for managing spatial data [5]. Hence, index structures for maintaining temporal data become relevant. However, a limited set of data structures has efficient support for managing the valid-time (the temporal range at which information is considered to be valid) associated with the handled information.

The *Interval B+ tree* (*IB+ tree*) is a fast data access method and promising index structure for the efficient handling of interval-based valid-time information [6]. Our literature review concerned a broader problem regarding spatiotemporal data management, where fast access plays an important role. Therefore, within this scope, we reviewed fast access methods for time-dependent data, among which emerged the *IB+ tree*. Consequently, we examined more recent work employing this index structure [7, 8, 9]. However, we were unable to find *IB+ tree* analyses with a focus on performance on growing volumes of data. The same can be stated about an evaluation

regarding time-efficiency on the use of the time-split operation: an optimisation presented by the Bozkaya and Ozsoyoglu that focus on improving the *IB+ tree* overall performance [6].

In this work, we present a time-efficiency focused variant to the *IB+ tree*, the *Improved IB+ Tree (I2B+ tree)*. This variant differs from the original *IB+ tree* by reducing the number of nodes traversed in the deletion of a stored object. Moreover, we establish and describe an empirical evaluation of this index structure in scenarios with distinct dense and sparse temporal datasets. The goal is to analyse the *I2B+ tree* performance on growing volumes of data.

A tested *TypeScript* open-source implementation of the *I2B+ tree* is provided with the purpose of using it in the experiments and making it 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 II summarises different structures for indexing valid-time information and presents an analysis of the *IB+ tree*. Section III presents the *I2B+ tree*. Section IV describes the experiments performed, as well as the results obtained. Section V analyses the obtained results. Lastly, section VI provides conclusions and identifies future work.

II. Related Work

In this section, we identify index structures that are capable of handling time intervals (valid-time domain). Then, of those, we describe in detail the index structure we identified as being the most promising for obtaining improved time-efficiency on growing volumes of data.

A. Valid-time index structures

Through a systematic literature review on valid-time index structures, we identified four main categories: spatial indexes storing bounding intervals in a single dimension; *B+ tree* variants; Interval tree augmentations; and others (e.g. *MPB-tree* [10]).

The structure most commonly used to represent unidimensional spatial indexes of algorithms are one-dimensional *R-trees*. Mahmood et al. [11] and Valdés and Güting [12] 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 [13].

Regarding *B+ trees*, there are many variants for handling temporal data. Among others, we can highlight Time Index [14]; *IB+ tree* [6]; and MAP21 [15]. 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 [15], 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 [16] augmentations represent the adaptation of a balanced tree structure to support intervals in the manner defined by the Interval tree. Carvalho et al. [17] 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 [10] 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. [5], 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 before, Bozkaya and Ozsoyoglu present the benefits regarding node accesses when comparing the *IB+ tree* to the one-dimensional *R-tree* [6]. Henceforth, we provide a more in-depth analysis of the *IB+ tree*, since this index structure emerged, from our review, as the most promising for handling valid-time intervals.

B. Interval B+ tree

The *IB+ tree* consists of a time-efficient index structure that merges the principles of both *B+ trees* [18] and Interval-trees [16]. 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 [6], 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 remotion 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 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 [18] presents a more in-depth analysis of the entirety of these cases.

We identified a problem in the node removal process that led to the deletion operation performance being slower than what was needed. We further detail this problem and our respective solution in the following section.

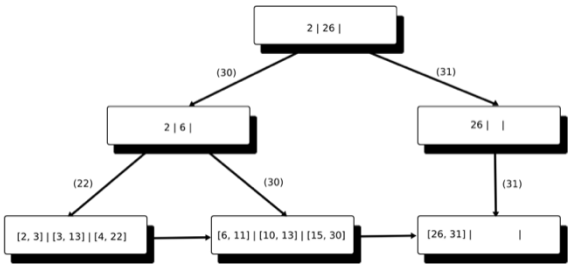


Figure 1. Original Visual representation of an *IB+tree* example.

The augmentation of the Interval-tree [16] 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.

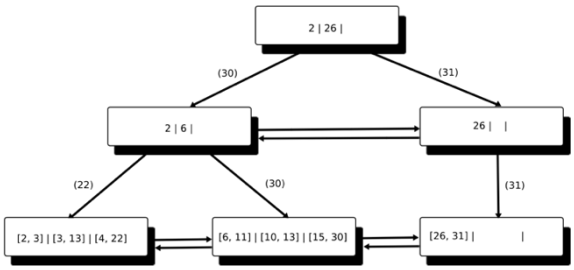


Figure 2. Visual representation of dataset in Figure 1 using the *I2B+ tree*.

Bozkaya and Ozsoyoglu [6] 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 tradeoff. 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 [6] present a more detailed explanation of the time-split algorithm and the impact of the *alpha* factor.

III. IMPROVEMENTS ON THE *IB+ TREE*

In this section, we present our proposed improvements with respect to the original implementation of the *IB+ tree*. Next, we analyse other implementation details, such as the tree mechanism for handling time-split intervals.

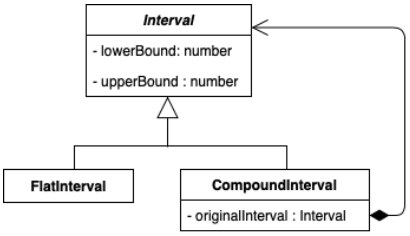


Figure 3. Simplified UML of the used Interval types.

A. *I2B+ tree main differences*

In section II, we presented the borrow and merge operations and the respective inner workings. As described, these operations require the access of a node to its siblings. Consider Figure 1, where a possible IB+ tree is visually represented. In the original structure, in the event of a deletion of the interval with lower bound 26 and upper bound 31, the leaf node containing it would try and borrow a node from its left sibling. To do so, considering the pointers between nodes of B+ trees, the application would have to travel to the ancestor node spawning a sub-tree containing both nodes. In Figure 1, this would imply travelling to the root and back from the root to the left sibling node.

With the goal of making these scenarios more efficient, and since our primary focus is time-efficiency (rather than storage space efficiency), we propose to modify the pointers to the sibling nodes that each node manages. Figure 2 visually represents the same scenario presented in Figure 1, but with our modified version of the *IB+ tree*. In our adaptation, a node, be it a leaf or an intermediary, always stores pointers for its siblings. Therefore, in borrow and merge operations, the process of accessing a sibling node becomes more straightforward. Furthermore, these pointers are easily maintainable, given the circumstances in which a node can appear or disappear.

B. *Other nuances*

An index structure should always abstract its external interface from the adopted inner data structure and internal data representation. Therefore, even though in the *IB+ tree* the inputted time intervals are possibly time-split, when a query is performed over the index structure, the result should contain the original input intervals that match the search criteria.

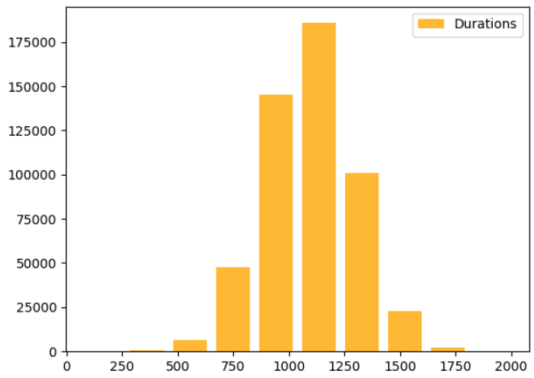


Figure 4. Example of a distribution of the durations of the intervals.

To satisfy this requirement, we propose the adoption of the *Composite* design pattern [19] for managing the internal representation of the inputted intervals. With this adoption, we consider two possible types of intervals: the *FlatIntervals* and the *CompoundIntervals*. *FlatIntervals* are the original intervals, used in insert operations. *CompoundIntervals* are the result of the application of a time-split operation to an Interval (be it a *FlatInterval* or a *CompoundInterval*). Hence, a *CompoundInterval* saves a pointer referring to the original interval that was subjected to the time-split operation. Figure 3 presents a visual representation of the different existent Interval types. Only the original *FlatIntervals* are returned when retrieving results as part of a query operation, using the pointer stored by the *CompoundIntervals*.

IV. EXPERIMENTS AND RESULTS

This section starts by describing the datasets used in the experiments. Afterwards, it describes the performed experiments, followed by the results obtained by the *I2B+ tree*. The discussion on these results will be presented in the next section.

A. Synthetic Datasets

To evaluate the performance of the *IB+ tree* variant, we proceeded with the generation of synthetic datasets. To assure that the generated datasets constituted viable test scenarios, we followed the norms presented by Theodoridis et al. [20], meaning that we make use of mathematical data distributions for generating the 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 [21].

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. What we try to accomplish with this configuration is to obtain two different scenarios where the number of concurrent intervals differs significantly. Hence, obtaining 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.

For each of these scenarios, we generate ten datasets of sequential doubling size: we start at a dataset with 1*k* intervals, then 2*k*, then 4*k*, and so on up until 512*k* intervals. Figure 4 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.

B. Experimentation

Experiments were conducted for the three basic operations: insertions, range queries and deletions. Regarding insertions, two different analyses were performed. The first insertion analysis, named tree insertion, consists of evaluating the average time it takes to construct the totality of the tree, given a test dataset. The second insertion analysis consists of evaluating the average time it takes to insert 100 randomly chosen intervals pertaining to the same test dataset. The range query experiment consists of evaluating the average time it takes to find the stored intervals that belong to a query range. Hence, it is an analysis of the range query performance. We chose to test with range query rather than single query, with the intent of integrating the analysis of an operation over a range. Lastly, the deletion analysis consists of evaluating the time it takes to delete 100 randomly chosen intervals stored in the tree. In order to be statistically significant, we use *Benchmark.js*[1]. With this tool, each test ran between approximately 50 and 100 times (being non-deterministic as it depends on the test performance).

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.

C. Tuning the order parameter

After running our *I2B+ tree*, in a multitude of distinct situations, crossing different characteristics, we proceeded with the comparison of the performance. Figure 5 shows the results for the tree insertion test regarding the two 512*k* datasets of both the *small intervals scenario* and *big intervals scenario*. We chose to present the largest dataset as being the most representative since the performance behaviour achieved across the datasets with other sizes is identical. In Figure 5 and for the following presented figures, the *a* label stands for the *alpha* parameter value (as described in section II.B). The *T* in the labels of the plots stands for the tree insertion test.

From the analysis of the plots present in Figure 5, we verify that independently of the dataset size, the structure with *alpha* set to 0, is always the fastest when it comes to the insertion operation of the entire dataset of intervals. Furthermore, from the line plots shapes in Figure 5, we observe that the order values ranging between 10 and 25 outperform the remaining order values. Therefore, for the remaining of the experiments, we focus on analysing configurations with the *order* parameter within range 10 to 25.

D. Tuning the alfa parameter

We also tried to analyse the performance tests from a perspective that would allow the tuning of the *alpha* parameter. However, the identification of optimal values for the *alpha* parameter is not as evident as in the *order* parameter.

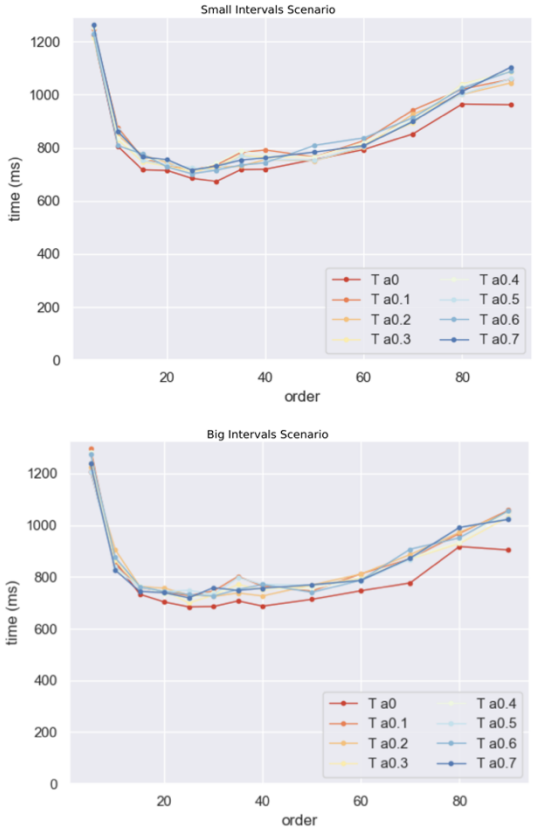


Figure 5. Tree insertion test, on the dataset with 512*k* intervals, in the identified scenarios.

We verified, through the empirical evaluation of the obtained results, that on insertion operations, the configuration with the *alpha* parameter with value zero outperformed the remaining values. This performance advantage is more emphasised on datasets with a smaller number of intervals (in both scenarios). In the remaining operations (although more obvious on deletions), when the *alpha* parameter is set to zero the structure seems to have slightly worse performance. Regarding the comparison of the other alpha values, from a general point of view, the alpha values of 0.2, 0.3 and 0.4 appear to be the most consistent and time efficient. Further results are shown in the GitHub referenced in section VI.

Figure 6 shows the performance results achieved with the insertion tests and with the 512*k* dataset, in the *small intervals scenario*, that highlights the characteristics of these alpha parameters. In Figure 6, and for the following presented figures, the *o* label stands for the *order* parameter value (as described in section II.B). The *I* in the labels of the plots stands for the insertion test.

E. Performance analysis

In this section, we analyse the time complexity of the proposed *I2B+ tree*, using the *Big O* notation, by evaluating the performance when varying the sizes of the test datasets. Since our improvement to the *IB+ tree* focuses on improving the deletion operation performance, the results obtained for insertions and range queries are also valid for the original structure. Figure 7 shows the performance of insertions, range queries and deletions over different increasing dataset sizes. In this set of tests, the selected *alpha* parameter was set to 0.2, on the two scenarios (*small intervals* and *big intervals*) and the different values of the *order* parameter belong to the identified optimum range. We restrain the configurations chosen in order to reduce visual clutter. In Figure 7, the *SI* and *BI* labels

identify, respectively, the *small intervals* and *big intervals* scenarios. The *RS* in the labels of the plots stands for the range query test.

From the analysis of Figure 7, we can verify that range queries and deletion have logarithmic performance, independently of the scenario. Insertion performance appears to be independent of the scenario and is the overall fastest operation. Additionally, the dataset size impact on insertions performance is minimal when compared to the other operations. Indeed, insertion behaviour is closer to a constant performance rather than a logarithmic one, since

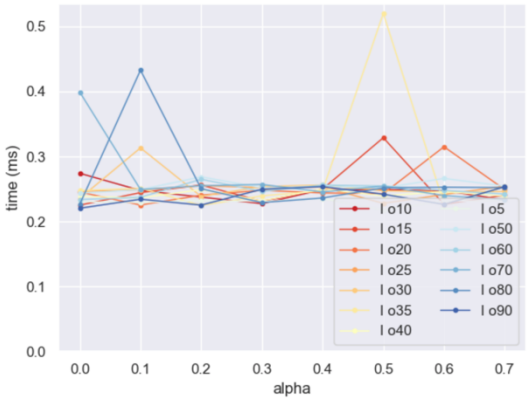


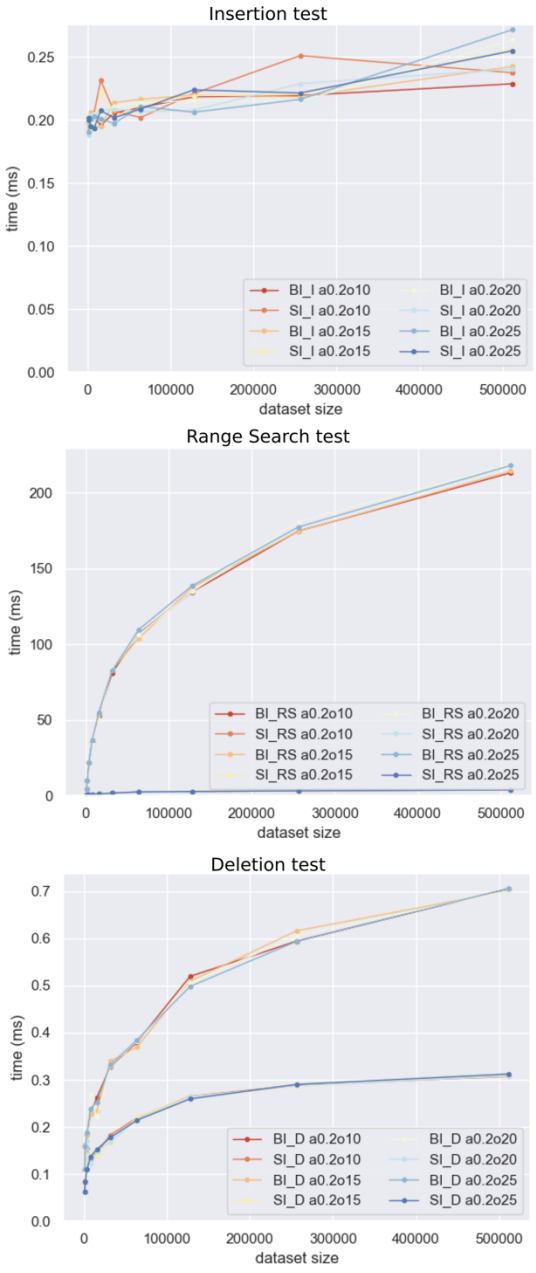
Figure 6. Insertion test, on dataset with 512*k* intervals, in the *small intervals scenario*.

a dataset 256 times bigger only represents an increment of 0.3 milliseconds on its performance, that corresponds to an increment of approximately 11.1% of the time spent. Range query is the more scenario-dependent operation. On the *big intervals scenario*, the range query operation is approximately fifty times slower than on its counterpart. Deletion operation performance also depends on the input interval duration, although not as heavily as range queries. On the *big intervals scenario*, the deletion operation is approximately four times slower than on the *small intervals scenario*.

V. DISCUSSION

In this section, we interpret the results achieved from the experiments. We verified that the *order* choice has a higher impact on performance than *alpha* does. Furthermore, the identified optimum range of *order* values can be explained by the consequences of choosing less-balanced values. Smaller *order* values lead to worse performances because of an increase of the tree depth and respective overhead for managing rebalancing tasks. Higher *order* values lead to worse performances because of the process of linearly iterating over more extensive lists of elements stored in a node. On the other hand, smaller *alpha* values (excluding zero) having better performances are justified by Bozkaya and Ozsoyoglu [6]: 'Higher values will lead to fewer splits and hence less storage expansion, but also to worse query efficiency. Smaller values will lead to better query efficiency but increase storage requirements'. Regarding the performance of configurations with the value of the *alpha* parameter set to 0, the performance on insertions is better since there is not the overhead of handling time-splits and consequent insertions (of the time-split intervals) that will occur.

The overhead of creating and managing time-splits is also the motive why configurations with the *alpha* parameter set to zero have better performance on smaller datasets. However, the inexistence of time-splits also explains the disadvantage of these trees with the query and deletion operations. The verified logarithmic time-efficiency of the *IB+ tree* is explained by the underlying structure: the *B+ tree*. The *B+ tree* also performs the mentioned operations with similar time-efficiency thanks to the branching-factor and rebalancing techniques [18].



VI. Conclusion

In this work, we presented a time-efficiency focused *IB+ tree* variant, the *Improved IB+ tree (I2B+ tree)*. Our structure differs from the original one by maintaining pointers to the siblings of each node. To evaluate the *I2B+ tree* performance, we tested it with distinct test scenarios, characterized by different characteristics and using synthetically generated datasets. From the results, we empirically verified and demonstrated that the *I2B+ tree* behaves logarithmically on searches and deletions and, with the insertion operation tending to a quasi-constant performance. From our experiments, we determined appropriate boundaries for the *order* and *alpha* parameters and provided justifications for the behaviours achieved. We consider the current study an improvement on the knowledge and performance regarding *IB+ trees*. The open-source implementation of the structure, as well as a more detailed analysis of the experiments developed, is available at *https://github.com/EdgarACarneiro/I2Bplus-tree*. Future work includes benchmarking the *IB+ tree* variant with other valid-time index structures and try and ascertain if there is any structure that excels. Furthermore, we expect to evaluate the behaviour of the index structure in a context oriented to in-memory databases and continuous data generation (e.g. data streams).

Acknowledgment

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).

References

[1] P. Cudre-Mauroux, E. Wu, and S. Madden, "Trajstore: An adaptive storage system for very large trajectory data sets," in *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*, pp. 109–120, March 2010.

[2] Mahmood, A. Aly, T. Kuznetsova, S. Basalamah, and W. Aref, "Disk-based indexing of recent trajectories," *ACM Transactions on Spatial Algorithms and Systems*, vol. 4, pp. 1–27, 09 2018.

[3] Q. Liu and H. Yuan, "A high performance memory key-value database based on redis," *JCP*, vol. 14, no. 3, pp. 170–183, 2019.

[4] D. Reinsel, J. Gantz, and J. Rydning, "The Digitization of the World –From Edge to Core," tech. rep., Seagate, IDC Information and Data, 112018.

[5] A. Mahmood, S. Punni, and W. Aref, "Spatio-temporal access methods:a survey (2010 - 2017)," *GeoInformatica*, 10 2018.

[6] T. Bozkaya and M. Ozsoyoglu, "Indexing valid time intervals," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 1460,pp. 541–550, Springer Verlag, 1998.

[7] H. C. Lock and D. Booss, "Indexing stored data," July 20 2010. US Patent 7,761,474.

[8] T. Guo, T. G. Papaioannou, and K. Aberer, "Efficient indexing and query processing of model-view sensor data in the cloud," *Big Data Research*, vol. 1, pp. 52–65, 2014.

[9] T. R. Vutukuri,*Q+ IB+ Tree: Indexing Technique for Moving Regions*. PhD thesis, Southern Illinois University at Edwardsville, 2018.

[10] Z. He, M.-J. Kraak, O. Huisman, X. Ma, and E. Xiao, "Parallel indexing technique for spatio-temporal data," *International Journal of Photogrammetry and Remote Sensing*, vol. 78, pp. 116–128, 04 2013.

[11] A. Mahmood, A. Aly, T. Kuznetsova, S. Basalamah, and W. Aref, "Disk-based indexing of recent trajectories," *ACM Transactions on Spatial Algorithms and Systems*, vol. 4, pp. 1–27, 09 2018.

[12] F. Valdés and R. Güting, "Index-supported pattern matching on tuples of time-dependent values," *GeoInformatica*, vol. 21, 01 2017.

[13] A. Guttman, "R trees: A dynamic index structure for spatial searching," vol. 14, pp. 47–57, 01 1984.

[14] R. Elmasri, G. T. J. Wuu, and Y.-J. Kim, "The time index: An access structure for temporal data," in *VLDB*, 1990.

[15] M. A. Nascimento and M. H. Dunham, "Indexing valid time databasesvia b+-trees," *IEEE Trans. on Knowl. and Data Eng.*, vol. 11,p. 929–947, Nov. 1999.

[16] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars, *Computational Geometry: Algorithms and Applications*. Springer Berlin Heidelberg, 2008. Section 10.1: Interval Trees, pp. 222-227.

[17] A. V. de Carvalho, M. A. Oliveira, and A. Rocha, "Improvements toefficient retrieval of very large temporal datasets with the travellight method," *2014 9th Iberian Conference on Information Systems and Technologies (CISTI)*, pp. 1–7, 2014.

[18] D. Comer, "Ubiquitous b-tree," *ACM Comput. Surv.*, vol. 11, pp. 121–137, 1979.

[19] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series, Pearson Education, 1994.

[20] Y. Theodoridis, J. R. Silva, and M. A. Nascimento, "On the generation of spatiotemporal datasets," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and LectureNotes in Bioinformatics)*, vol. 1651, pp. 147–164, Springer Verlag, 1999

[21] S. K. Katti and A. V. Rao, "Handbook of the poisson distribution," *Technometrics*, vol. 10, no. 2, pp. 412–412, 1968.

[1] https://benchmarkjs.com

[2] https://nodejs.org