Journal of Information Technology Management
ISSN-OnLine: 2980-7972
University of Tehran

Incorporating Retroactive Operations in Large Temporal Databases Using Retroactive B-Tree

Santosh Kumar Verma*

*Corresponding author, Assistant Prof., Department of Computer Science and Engineering, JK Lakshmipat University – Jaipur, Rajasthan, India. E-mail: santosh.verma@jklu.edu.in

Suman Saha

Assistant Prof., Department of Computer Science and Engineering, Jaypee University, Anoopshahr, UP, India. E-mail: suman.saha@mail.jaypeeu.ac.in

Sanjay Goel

Prof., Department of Computer Science and Engineering, Jaypee University, Anoopshahr, UP, India. E-mail: goelsan@yahoo.com

Journal of Information Technology Management, 2025, Vol. 17, Special Issue, pp.87-107.
Published by the University of Tehran, College of Management
doi: https://doi.org/10.22059/jitm.2025.102923
Article Type: Research Paper

Received: January 17, 2025
Received in revised form: March 03, 2025
Accepted: June 13, 2025
Published online: August 01, 2025

Abstract

Temporal databases, quickly rising in size, are distinguished by their capacity to maintain the older version of data objects against actions on them, allowing logical deletions. Queries for historical data are particularly costly due to the linear scanning of temporal versions. Temporal data structures like time-split B-Tree or multiversion B-Tree are working underlying the state-of-the-art temporal databases. So far, most efficient temporal data structures are partially persistent or fully persistent, but none of them support retroactive queries. On the other hand, efficient temporal indexing is required to address bulk loading in a real-life application. To the best of our knowledge, there is no efficient solution for bulk loading and updating retroactive index structures. This article seeks to offer a new data structure, the Retroactive B-Tree (RBT), to facilitate retroactive operations in temporal databases as well as bulk loading. It presents theoretical and empirical research and analysis of the suggested data structure and its relevant operations. The experiments were conducted to demonstrate the performance of the proposed retroactive B-Tree in terms of execution time, I/O complexity, space complexity, and bulk loading. The obtained results show that indexing

with a buffer is the most powerful model for existing temporal databases for implementing a retroactive B-Tree. The tree of lists architecture is observed as an I/O efficient data structure for all variants of temporal indexing for large databases.

Keywords: Indexing, Retroactive Query Answering, Temporal Databases, Retroactive B-Tree, Persistent B-Tree, Temporal Database Indexing Problem.

Introduction

Aspects of the real world are stored in the databases. Dynamic characteristics are challenging to record and manage in traditional databases. The most recent version of the modelled environment can only be stored using conventional database models. Future updates overwrite the current record. As a result, it cannot access past data for predictive or trend analysis. This limitation can hinder the ability to make informed decisions based on historical trends. To overcome this challenge, temporal databases have been developed. These databases are designed to handle dynamic data and allow for easy access to historical records for analysis.

By utilizing these advanced database technologies, organizations can gain valuable insights from past data, predict future trends, and make more informed decisions. This enables them to stay ahead of the competition and adapt quickly to changing market conditions. Ultimately, by leveraging these modern database solutions, businesses can unlock the full potential of their data and drive innovation in their operations.

Retroactive indexing is necessary for dynamic applications that demand ad hoc version management, audit trail, backwards-compatible updates, and mistake repair. Users of the application can study records through temporal queries to verify the propagation of changes, execute rollback operations, and/or analyze previous observations. The databases that provide the attributes help us analyze and model the entity's dynamic evolution and locate causal or temporal relationships between them. Due to the high cost and limited memory capacity of disc technology, historical database searches were initially not feasible. However, large storage capacity will become more accessible as external memory technology advances in the future (George Copeland, 1980). More information can now be stored for a very long period. As a result, database systems with temporal support or version management have become increasingly popular. However, recent bibliographic research demonstrates that handling temporal data has become more critical. Effective temporal data processing enables various fascinating real-world applications, including advanced business analysis, fraud detection, automation, and more. A time series database (TSDB) software system is tailored for storing and delivering time series using related pairs of time and value. In several disciplines, time series are also known as trends. Early time series databases were frequently used to support industrial applications capable of reliably retaining recorded values from sensory equipment.

However, they are used to supporting a far larger range of applications nowadays. Science and web applications both benefit from using a time-series database. With certain inherent restrictions that will be covered later, IBM (Cynthia et al., 2010), Teradata (Anton et al., 2016), Oracle (Ravi, 2007), and Microsoft (Lomet et al, 2008) have provided temporal database systems. The retroactive B-Tree is a novel data structure that this article presents to facilitate retroactive queries in temporal databases. This article makes the following key contributions: (a) using bulk operations for retroactive index structures, this work defines the first retroactive B-tree, an asymptotically optimal data structure. The bulk operation shows that previous techniques required significant I/O costs for external memory access. The objective here is to reduce the number of I/O operations with the proposed paradigm; (b) The research findings show that our recommended techniques greatly outperform the current state of the art in terms of performance.

The rest of the paper is structured as follows: The related work section summarizes prior works that are still relevant and discusses the critical characteristics of temporal databases and the shortcomings of the current state of the art that necessitate novel indexing methods. Limitations with the existing state of the art present the problem formulation. A proposed novel indexing model, Retroactive B-Tree, is shown in the proposed model section. The proposed method for bulk loading is applied, and the outcomes are shown in the results section. The last section of this paper provides a conclusion of the overall observations, implementations, and limitations of the research carried out.

Literature Review

We go through the pertinent state-of-the-art challenges, related index structures, and bulk-loading and bulk-updating procedures in this part. Consider the processing of temporal inquiries as well, according to the state of the art. Traditional database systems usually maintain a single version of the data. However, several applications, including those for engineering design and statistics, moving-object records, governance, finance, and legal records, as well as applications for inventory control and scheduling, require access to the historical data (Rudolf & Mario, 1977; Gultekin & Richard, 1995; Taghva et al., 2016; Harshit & Santosh, 2021; Ruijie et al., 2022). These applications need a custom database structure to handle multiversion data and maintain excellent query performance and space constraints.

The design of database structures for storing multiversion data has advanced significantly, according to research in the field (Daniar & Bernhard, 2013; Lars Arge et al., 1999; Tuukka et al., 2008; Tuukka et al., 2009; Lomet & Betty, 1989; Betty et al., 2004). For storing multiversion data, a comprehensive form of B-Tree and B+Tree is implemented (Rudolf & Mario, 1977; Daniar & Bernhard, 2013; Lars Arge et al., 1999; Tuukka et al., 2008; Douglas, 1979; Lars Arge, 2003; Lars Arge et al., 2003). These thorough implementations provide

multiversion concurrency management and are effective for precise match queries. One can also leverage snapshot isolation to provide multiversion concurrency control (Hal et al., 2007; Alan et al., 2005). However, these structures were insufficient for range queries since it could be necessary to manually scan each consecutive entry of a certain version, adding to the time and space complexity. It is further noted that none of the models has perfect logarithmic execution times for all operations.

The eternal database prototype employs the time-split B+tree in (Lomet et al., 2005); however, it does not condense pages. A multiversion structure that ensures logarithmic execution durations for all operations, but does not offer concurrency control, is the multiversion B+ tree in (Peter et al., 1996). Other MVBT variations have been implemented by Tuukka et al. (2008) and Jaluta et al. (2005), offering full concurrency control but with slower updating operations. Several persistent models of index structures, bulk loading, and bulk updates have been reported in the literature. Partial persistence is a concept that is frequently utilized in computational geometry (Michael et al., 1993; Marios et al., 2006) and algorithm design. In databases, the problem with partial persistence is sometimes referred to as versioning, transaction time, and system time (Lars Arge et al., 2003). The partial persistent model enables the operations (Leveldb, 2011; Krishna & Michels, 2012) in the present version of the temporal database only. Commercial organizations have developed multiversion systems, including IBM (Cynthia et al., 2010), Teradata (Anton et al., 2016), Oracle (Ravi, 2007), and Microsoft (Lomet et al, 2008). Yufei & Dimitris (2001), Rui & Martin (2010), and Elisa et al. (2012) have developed several persistent B-trees, allowing range queries over older versions and providing similar query performance to a fundamental B+tree. Only a partial-persistent model was used in the investigation of persistent B-trees. The design of moving object database indexes is significantly impacted by MVBT and other partly persistent B-trees (Marios et al., 2006). Retroactively speaking, the issue with bulk loading and updates has not yet been sufficiently rectified. The first bulk loading method puts all active nodes into memory using a partly persistent B-tree (Tuukka et al., 2009). Since dynamic databases are significantly larger than the memory capacity, this assumption isn't always true. Data tuples are organized as sorted maps in key-value stores like HBase (2017) and levelDB (2011), which also offer effective key range queries. However, it is impossible to utilize range searches over non-key properties, such as temporal queries. However, these systems leverage the LSM tree rather than the traditional B+ tree, which reduces the cost of updates. The updates in LSM Tree still need to be merged with preceding data, resulting in a significant data merging overhead and limiting the throughput for insertion. Time series databases, like BTrDb (Michael & David, 2016), Druid (Fangjin et al., 2014), Gorilla (Tuomas et al., 2015), Waterwheel (Li Wang et al, 2018), and MVlevelDB (Zhao et al., 2021), are made for low-latency, real-time queries on time series data. However, due to the absence of secondary range indexes, they do not provide effective range searches over non-temporal properties.

Limitations of the Existing State of Art

The insights from the literature review on the current state of the art are presented in this part. Our observations are also listed in Table 1. Temporal databases have the following characteristics: they allow logical deletions, they preserve the earlier version of data objects against operations on them, and their size increases quickly. Queries become quite costly since older versions are linearly scanned. Range queries must be supported across a certain subset of past versions and on a particular past version. Traditional B-trees and their variants are not appropriate for efficiently indexing temporal databases because they can index only one database snapshot. As a result, several temporal extensions for B-trees (Ruijie et al., 2022; Daniar & Bernhard, 2013; Tuukka et al., 2008; Tuukka et al., 2009; Peter et al., 1996; Lehman & S Bing, 1981) have been created. The MVBT (Tuukka et al., 2008; Tuukka et al., 2009) is the first partially permanent index structure enabling temporal key-range queries, insertions, updates, and removals. However, the MVBT or the time-split B-tree (Ruijie et al., 2022; John et al., 2016) does not support efficient algorithms for bulk loading and bulk updates compared to MVBT+ and MVBT-LRU (Tuukka et al., 2008, 2009). The design of efficient algorithms for partially persistent B-Trees has been reported in the past, while very few research articles have defined retroactive indexing problems. The persistent changes deal with the current or a past version and require a more straightforward treatment, whereas the retroactive changes deal with both the current and past versions. This article discusses bulk loading and updating across retroactive B-Trees supporting the key and time range query.

Table 1. Limitations of the Existing State of Art

Existing Systems	Query Efficiency		Base	Limitations
	Key Range	Time Range		
HBase (2017)	Yes	No	LSM Tree	No Support for Indexes
LevelDB (2011)	Yes	No	Used with Hive or Mapreduce.	It may cause memory issues and no support for query optimization.
BTrDb (Michael & David, 2016)	No	Yes	Berkeley Tree	Supports Point query over a time range
Druid (Fangjin et al., 2014)	No	Yes	Rest API	Supports Point query over a time range, Slow, Update Latency is high.
Gorilla (Tuomas et al., 2015)	No	Yes	In-memory TSDB	Supports Point query over a time range, Slow, Update Latency is high.
Waterwheel (Li Wang et al, 2018)	Yes	Yes	LSMV Tree	Proposed models are based on the template B+ tree and applied to IoT-based data streaming. While the Chrono size is too short, about 200ms, the data streaming rate was also slow.

MVlevelDB (Zhao et al., 2021)	Yes	Yes	LSMV Tree	Proposed models are based on the template B+ tree and applied to IoT-based data streaming. While the Chrono size is too short, about 200ms, and the data streaming rate was also slow.
MVBT (Tuukka et al., 2009)	Yes	No	B+ Tree	Does not support the Time range query and is also unable to manage Buffer tree splitting and synchronization with version management.

Indexing Problem in Temporal Databases

This paper presents a retroactive B-tree that efficiently supports mixed operations such as insert, delete, update, and query over bulk-loaded data. This section outlines several indexing challenges observed in the literature: (a) A major issue is the lack of attention to data splitting and distribution among multiple nodes as the size of the versioned database grows. This study addresses the need to synchronize node expansion with version management. (b) The concepts of versioning, transaction time, and system time are commonly used to describe partial persistence issues in databases (Arge, 2003). (c) To the best of our knowledge, no prior research has explored a retroactive multiversion B-tree. In this study, we compare our empirical findings with the current state-of-the-art approaches.

It has also been observed that the concept of multiple discrete time conceptions for keeping or querying the temporal database has not been effectively addressed in research articles on this subject, utilising any form of the B-Tree. Time is typically seen as a continuous variable. The perspective of time must be discrete for a time model to be implemented on discrete computer hardware. This viewpoint is what our paper adopts. A conceptual view is given in the preliminaries section.

Methodology

This section presents the implementation of a retroactive B-Tree supporting retroactive queries. This computational model works on a single processor configured with a limited internal memory capacity and a huge external memory space. The proposed retroactive model is achieved with the modification of the B-Tree algorithm licensed to SPDXLicense-Identifier-GPL-2.0 (Spdx, 2020). A sample B-Tree of SPDX is depicted in Figure 1. The suggested B-Tree data structure is bound by the following: Support a single process execution. The maximum key size is 120 bytes, the maximum data page size is 64K, and the maximum page size is 65024 bytes. The proposed temporal B-Tree is shown in Figure 2 with two versions of the B-Tree. We obtained the retroactivity on the basic B-Tree model by maintaining the various versions with the help of the following five algorithms.

1. Initialization

Algorithm 1 for retroactive B-tree initialization works by initializing the timeline variables and memory size.

Algorithm 1: Retroactive B-Tree Initialization

---

procedure SET INITIAL(Btree < T > initial)

call timeline.clear()

size = 0

initial = initial

make operation(Operations<T>Operation::INITIAL, 0, -1)

end procedure

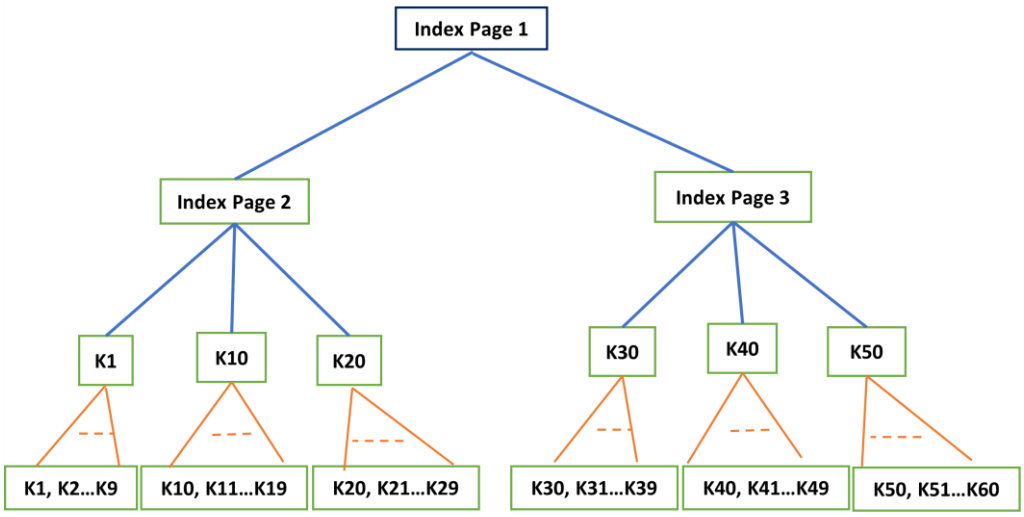

Figure 1. Traditional B-Tree

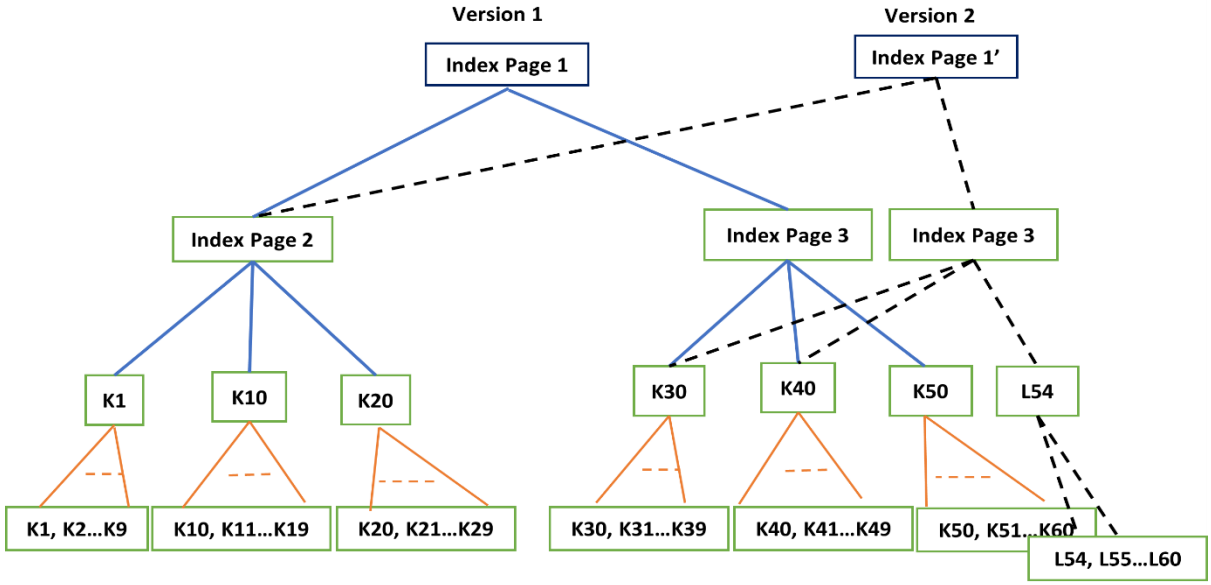

Figure 2. Retroactive B-Tree, dotted lines indicate the version

2. Insertion

Algorithm 2 for retroactive B-Tree insertion works by inserting a value "k" into a specified version and propagating the same till the current version of the tree by using algorithm 5 for updating. Algorithm 5 is responsible for updating the version information of the nodes in the B-Tree as the insertion operation propagates through different versions. This ensures that the changes made during retroactive insertion are reflected accurately in each version of the tree.

The process starts by inserting the value "k" into the specified version of the tree using Algorithm 2. This insertion may require splitting nodes and creating new ones to maintain the B-Tree properties. Once the value "k" has been successfully inserted into the specified version, Algorithm 5 is used to update the version information of all affected nodes. The update process involves traversing from the leaf node where "k" was inserted up to the root of the tree, adjusting version numbers as needed. By doing so, each version of the B-Tree reflects the correct state after inserting "k", allowing for efficient querying and modification of past versions.

Algorithm 2: Retroactive B-Tree Insertion Algorithm

---

procedure INSERT(Tk, int version = −1)
make operation(Operations<T> {Operation::INSERT, k}, version)
end procedure

---

3. Deletion

The process to remove an element from any specified version of the retroactive B-Tree is given in Algorithm 3. After deleting the desired element, the algorithm creates a new version of the specified current version of the tree by using algorithm 5. This new version of the tree will reflect the changes made by deleting the element, while still maintaining the retroactive property of being able to query the state of the tree at any point in time. The update process ensures that all previous versions of the tree remain unchanged, allowing for efficient and consistent access to historical data. By following these algorithms, users can easily manipulate and track changes in the retroactive B-Tree structure without compromising its integrity or performance.

Algorithm 3: Retroactive B-Tree Deletion Algorithm

```
procedure REMOVE(Tk, int version = −1)
make operation( Operations<T> {Operation::REMOVE, k}, version)
end procedure
```

4. Update Operation

The update operation for the retroactive B-Tree, to modify an existing value in the tree by a new value, is given in Algorithm 4. After updating the desired element, the algorithm creates a new version of the specified current version of the tree by using algorithm 5. This new version of the tree will reflect the changes made during the update operation, ensuring that the updated retroactive B-Tree remains consistent and up to date.

The algorithm considers the structure of the tree and ensures that all necessary adjustments are made to maintain its properties. By following this process, users can easily modify values in a retroactive B-Tree without compromising its integrity.

Algorithm 4: Retroactive B-Tree Update Algorithm

```
procedure UPDATE(ToldK,TnewK, int version = −1)
make operation( Operations<T>{Operation::UPDATE, oldK, newK}, version)
end procedure
```

5. Retroactive B-Tree Version Update

It is expected to maintain an updated version of the retroactive B-Tree during operations such as inserting, deleting, or updating over a timestamp. The retroactive B-Tree version update algorithm rolls back to the specified version under an operation, creates a new version of the B-Tree on the specified timestamp, and propagates the result update until the current version. This process ensures that the retroactive B-Tree remains consistent and accurate throughout all operations, allowing for efficient retrieval of data at any point in time. By constantly updating and propagating changes to each version of the B-Tree, users can easily access historical data and track changes over time. This level of detail and precision is crucial for applications that require a high level of data integrity and historical accuracy. The retroactive B-Tree version update algorithm plays a key role in maintaining this level of consistency and reliability within the system.

Algorithm 5: Retroactive B-Tree Version Update

```
procedure MAKE OPERATION(Operations<T> op, int version=-1)
version = (version == −1||version < size)? version=size:version
if version=size then
timeline.emplace back(vector<Operations<T >>{op})
inc()
else timeline[version].emplace back(op)
end if
end procedure
```

This process ensures that the retroactive B-Tree remains consistent and accurate throughout all operations, allowing for efficient retrieval of data at any point in time. By constantly updating and propagating changes to the tree, users can access historical versions of the data while still being able to make real-time updates.

This level of flexibility and control over the data structure is crucial for applications that require a high level of data integrity and consistency. The retroactive B-Tree version update algorithm plays a key role in maintaining this balance between historical accuracy and real-time updates.

Experimental Setup

In this section, we present the details about the experimental setup and the various datasets through which empirical analysis of the proposed data structure is carried out.

1. Setup

The B-Tree algorithm licensed to SPDX-License-Identifier- GPL-2.0 (Spdx, 2020) is used to implement all algorithms in C. By adding one more term "t" to an index item, RBT is added on top of the current B-Tree. The empirical evaluation is carried out on an ASUS-VivoBook-R542U workstation with an Intel Core i7-8550U and 8GB of memory running Windows OS (Win10), a magnetic disk (HGST HTS541010B7E610, 1TB), and an SSD (Lexar SSD, 256 GB). We solely used the raw device interface in our research to prevent the operating system from affecting the results. We conducted our experiment using pages of 4KB, 8KB, and 16KB in size. In addition, we set the buffer cache size to be the block size of 512 KB, and the ratio of the L1 cache is set to be 32KB, as 512 KB/16 processes. So, the size of a block for the node size of 25 is calculated as 32/25, i.e., 1.28. Thus, we get the maximum number of degrees for the memory B-Tree to be 1280. For each data set in our examination, we randomly selected 10 versions and calculated their estimated checkout time. The OS page cache was cleared before each run, and each experiment was performed five times. We took the average of the remaining three trials after excluding the two extreme numbers from the five trials due to experimental variation. Table 2 represents the running time of the B-Tree bulk loading operations on the setup mentioned above.

2. Data

Our experiments use workloads like those used in earlier experimental investigations using versioned databases (Cynthia et al., 2010; Betty et al., 2004; Elisa et al., 2012). We have considered the index bulk loading workload in this experiment. We look at how the quantity of update and delete operations affects I/O loading performance and space use. We have six files to load: db50, udb0, udb25, udb50, udb75, and udb100. The performance of the retroactive B-Tree has been evaluated using the synthetic data set of randomly generated 1,000,000,000 operations in each file. A larger workload than the specified is not considered due to expensive execution time and constraints with the below setup. We divided the dataset by a specific proportion of insertions, updates, and search queries to analyze the execution time of the proposed algorithms.

Case 1: The dataset is defined with 50% insert, 50% update, and 0% search query operations.

Case 2: The dataset is defined with 60% insert, 30% update, and 10% search query operations.

Case 3: The dataset is defined with 40% insert, 40% update, and 20% search query operations.

The operation distribution is subjective and may vary depending on the test cases and the experimental setup.

3. Experiments

We have conducted several experiments to assess the effectiveness of the suggested retroactive data format. This experiment's main goal is to examine data structures under diverse data set configurations, temporal environment factors, and multiple models. Details of the experiments and data are given in the following subsections.

A. Experiment-1

An experiment was conducted to measure the execution time of the Retroactive B-tree (RBT) under three different data distributions. The execution time of the RBT over a set of operations for the different degrees of the node is shown in Table 2.

B. Experiment-2

Experiment on architecture, list of the trees vs tree of the lists for three different data distributions with the Retroactive B-Tree. Figure 4 represents the first retroactive tree architecture, where every new version is connected as a node in a linked list. The headed node tree is the recent version of the tree, while the last node is the first version of the tree. The second architecture depicted in Figure 3 represents a single tree, and leaves relate to a list of operations per node. The immediate nodes connected to the leaves are the current version, while the last nodes, the deepest nodes, are the initial versions of the operations in the tree. The I/O time per operation of both variants for different node degrees is shown in Tables 3 and 4.

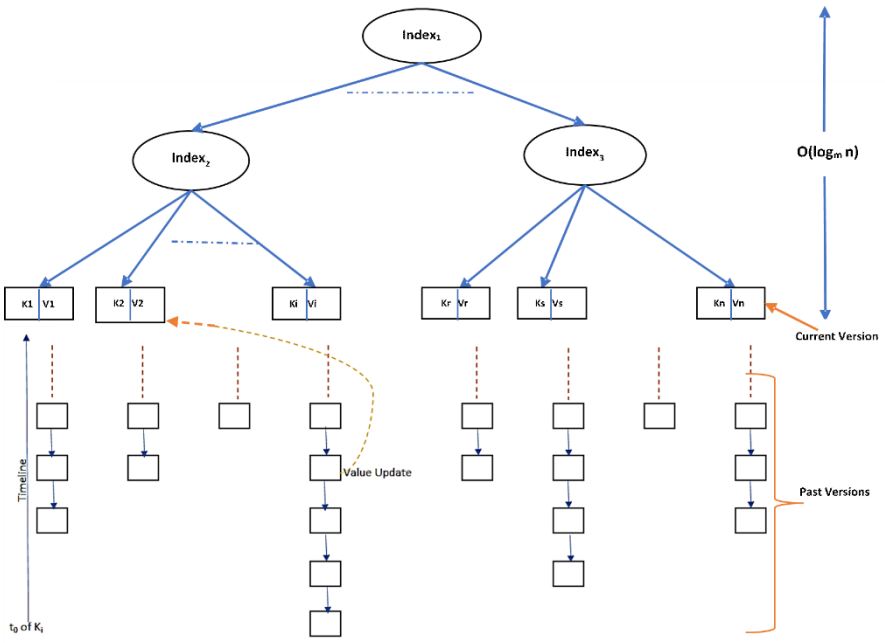

Figure 3. Tree of List Architecture

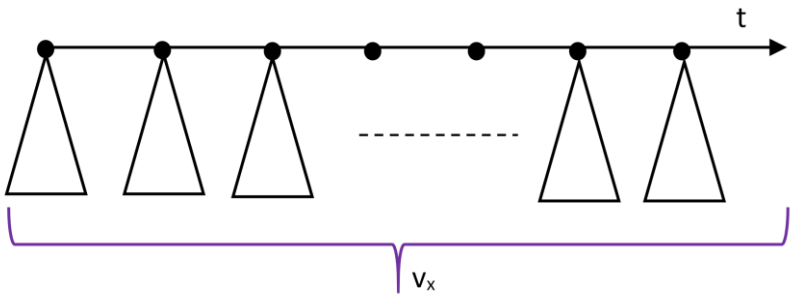

Figure 4. List of Trees Architecture; every triangle represents a tree at the ith timestamp

Results

This section presents the experimental results of the discussed setup and a comparison with the existing state of the art. In this part, we compare our new bulk loading technique on retroactive B-Trees with iterative MVBT-LRU and MVBT+ loading in terms of performance. For each workload file, Figure 5 shows the total number of I/Os needed to load MVBT+, MVBT-LRU, and RBT. We employed a fixed memory capacity of 3.2 MB and a page size of 8 KB.

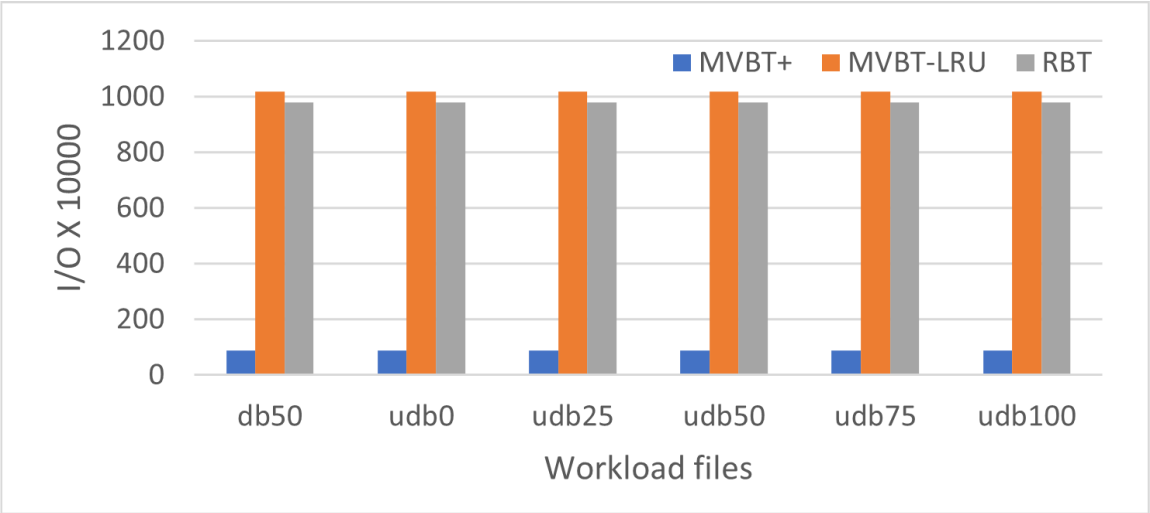

Figure 5. I/O performance comparison: Performance of MVBT-LRU, MVBT+, and RBT during loading (on a logarithmic scale) (memory size is 3.2 MB, the page size is 8 KB considered)

The results of our comparison show that our new bulk loading technique on retroactive B-trees outperforms both iterative MVBT-LRU and MVBT+ loading in terms of total number of I/Os required. This indicates that our technique is more efficient at handling workloads and managing memory resources effectively. Additionally, the fixed memory capacity and page size used in our experiments provide a consistent basis for comparison across different loading techniques. Overall, these findings suggest that our new bulk loading technique offers a promising solution for optimizing performance in retroactive B-tree structures.

The results are presented on a logarithmic scale for several I/Os. RBT outperforms MVBT and is substantially closer to MVBT-LRU. The ratio of I/Os needed to load MVBT-LRU and RBT as a function of memory size is shown in Figure 6. RBT yields its greatest results when used for 200 pages. RBT's I/O performance increases as memory capacities increase and gets a little bit closer to MVBT-LRU. RBT works better since there are fewer live versions. The RBT outperformed MVBT+ for updates using file udb100 by a factor of 15.

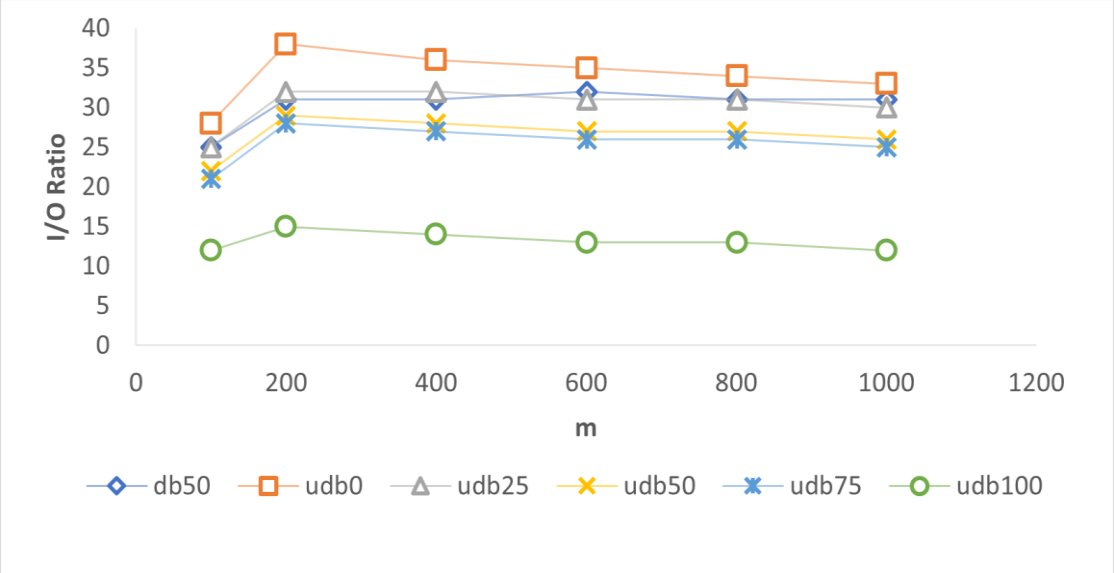

Figure 6. I/O performance comparison: The I/O Ratio of MVBT-LRU and RBT as a function of memory capacity, where page size = 8KB

As memory size increases, RBT's performance improves even further, narrowing the gap between RBT and MVBTLRU. Overall, the results suggest that RBT is a strong contender in terms of efficiency and scalability compared to other data structures like MVBT and MVBT-LRU. Figure 7 shows the I/O ratio as a function of the page size for loading the udb50 data set. There are 400 pages in total set aside in the memory capacity. The middle curve shows the ratio of I/Os needed for loading MVBT+ to those needed for loading RBT. It is demonstrated that the I/O performance increases linearly with page size. RBT performs quicker than MVBT+ for pages that are 16 KB in size. The top curve shows the worst-case scenario for MVBT without an LRU buffer. The graph displays the differences between MVBT+ and RBT in relative performance increases.

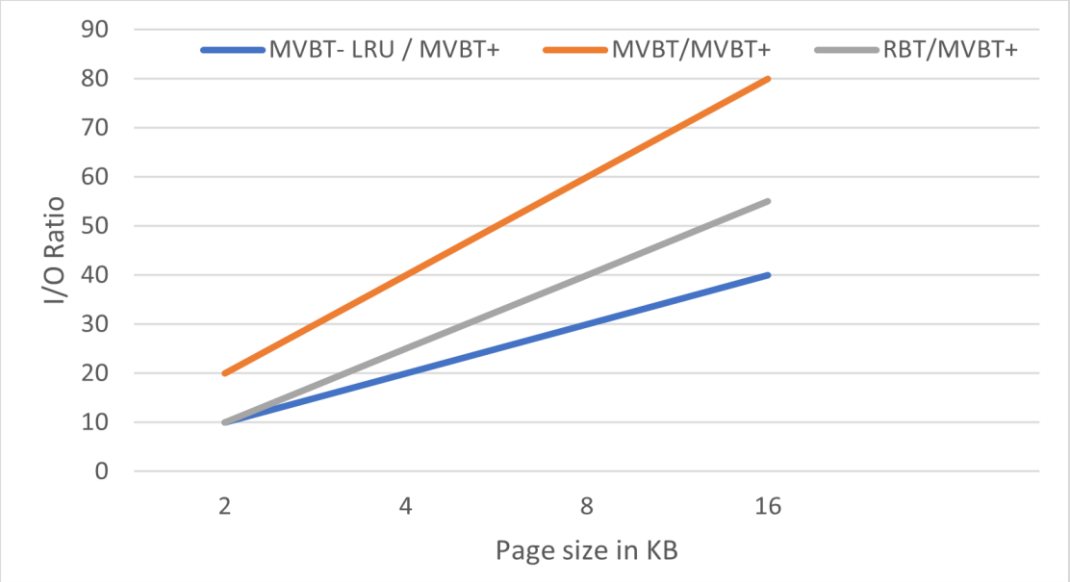

Figure 7. I/O ratio for the udb50 data set as a function of page size, memory capacity m = M/B = 400

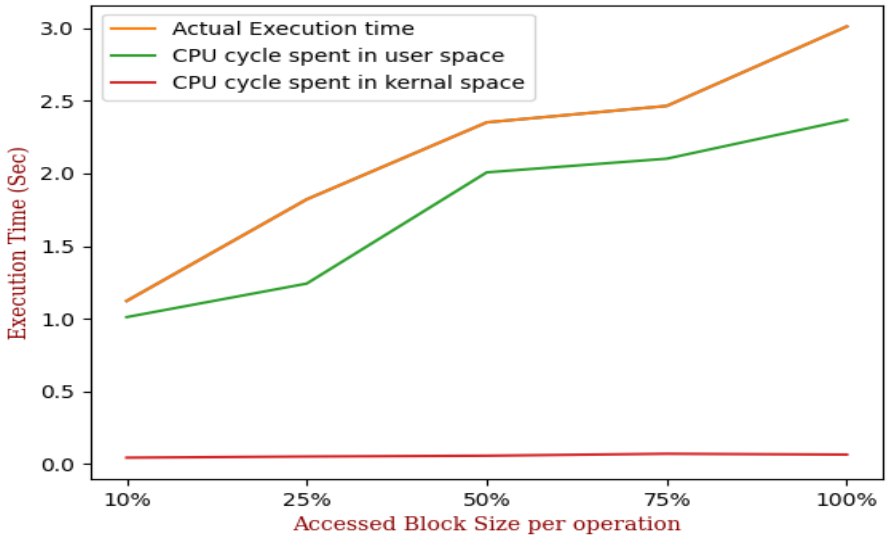

Figure 8. Experiment #1: Retroactive B-Tree Execution Time over Operations

Along with loading, we also ran several experiments to determine the I/O effectiveness of bulk updates on a certain RBT, MVBT+, and MVBT-LRU. We initially made 500,000,000 updates (or 50% of the total updates) for each data set. Then, we processed the remaining updates using a series of bulk updates (with a given batch size). Figure 8 shows the graph of the execution time of the retroactive B-Tree as a function of batch size corresponding to Experiment 1. The execution time over a set of operations for different node degrees is shown in Table 2. Two hundred pages were designated as the memory size.

Table 2. Experiment #1: Retroactive B-Tree Operation Execution Time

Degree	ActualTime	UserTime	SystemTime
120	1.8233	1.9120	0.0447
320	2.1223	1.9427	0.0588
640	2.8520	2.9980	0.0582
960	3.2650	3.3621	0.0817
1280	3.5117	3.3689	0.0660

Another experiment is conducted to find the optimal solution for space complexity. Two proposed architectures of retroactive data structures in Experiment 2 have been used. Figure 4 represents the first retroactive tree architecture, where every new version is connected as a node in a linked list. The head node tree is the recent version of the tree, while the last node tree is the first version of the tree. The second architecture depicted in Figure 3 represents a single tree, and leaves relate to a list of operations per node. The immediate nodes connected to the leaves are the current version, while the last nodes, the deepest nodes, are the initial versions of the operations in the tree. The I/O time per operation of both variants for different node degrees is shown in Tables 3 and 4. The theoretical I/O complexity is O(k*logB n) for search or update and uses O(n/B) blocks for a list of trees architecture as shown in Figure 4, where k is the number of nodes in the linked list. And for the second architecture, as the tree of the lists, the expected amortized I/O complexity is O((logmn)/B +z), where z is the cost to maintain the list in the leaf.

Table 3. Experiment #2: I/O time per Operation in List of Tree Architecture

Degree	I run (µs)	II run(µs)	III run(µs)
120	45.497	45.874	47.156
320	49.308	44.281	45.552
640	46.076	46.33	52.186
960	47.645	46.754	46.056
1280	43.245	47.28	47.45

Table 4. Experiment #2: I/O time per Operation in Tree of List Architecture

Degree	I run (µs)	II run(µs)	III run(µs)
120	4.23	4.43	3.65
320	4.83	5.09	5.64
640	5.165	5.45	5.30
960	4.80	5.96	6.03
1280	5.48	7.76	6.96

Discussion

This section presents the observations from our experiments with the proposed retroactive B-Tree for indexing problem handling in temporal databases. The proposed data structure is compared with the best-known existing state-of-the-art MVBT-LRU and MVBT+. The data structure was evaluated on different parameters such as its execution time on bulk loading, I/O complexity (I/O ratios), and space complexity. Our intention through the experiments was to have a comparative analysis of the retroactive data structures under various setups of the

dataset, temporal environment variables, and different models. The result section presents the experiment's results over the execution time of the proposed retroactive data structure in Table 2. The I/O ratio of MVBT-LRU is observed to be comparatively faster than the retroactive B-Tree. But the amortized I/O complexity of the retroactive B-Tree is better than the MVBT and MVBT+. As our objective was to find a suitable retroactive data structure for efficient indexing with a lower I/O bound, we recommend a retroactive B-Tree.

In the second experiment, we compared the I/O complexity of two proposed architectures of retroactivity in data structures. We have tested the list of trees and tree of lists models for implementing a retroactive B-Tree. The purpose of this experiment is to observe the minimum I/O access and space complexity. The list of trees model requires more space as it maintains a replica of the previous version, including the update in every new version. While the tree of lists maintains a tree at the end of a chronon, every new operation is inserted at an appropriate position in the lists appended with the tree's leaves. This model does not require keeping a replica of the tree structure. The I/O time taken by both the architectures for the insert operation is tabulated in Tables 3 and 4 for the list of trees and tree of lists models, respectively. The result in Table 4 also includes the time to place the operation in the list, i.e., the time taken to maintain the skip list. The result of this experiment depicts that the tree of lists model is more promising than the list of trees model, as it takes less execution time for I/O operation and less space for maintaining the data structure.

The only limitation with the tree of lists architecture is that it follows the lazy search operation because the lists maintain the operations while the search is applicable in the stable tree. All the operations maintained in lists must be settled, meaning the tree must complete a chronos duration for a relevant search result.

Conclusion

The research in this paper is carried out on designing, implementing, and evaluating a retroactive data structure to address the indexing problems in temporal databases of the modern era. We have attempted to model a retroactive B-tree to address issues such as solving the indexing problem of large databases with minimal I/O bounds, managing historical transactions, and pruning past errors from faulty versions of the database. This paper introduces the Retroactive B-tree (RBT), the first retroactive data structure that allows bulk loading in an asymptotically optimal number of I/Os while maintaining all worst-case performance requirements.

It is comparable to the multiversion B-tree in that it provides all performance guarantees, like MVBT-LRU. MVBT-LRU loading is significantly quicker—by a factor linear to the page capacity—than earlier loading techniques, such as loading using iterative updates. Performance analysis is carried out based on the execution time of bulk loading, I/O

complexity, and space complexity. We have observed that indexing with a buffer is the most powerful model for existing temporal databases. Implementing a retroactive B-tree, following the tree-of-lists architecture, is observed to be an I/O-efficient data structure for temporal indexing of large databases.

In the future, we plan to validate the proposed model with a two-dimensional range query and analyze the effect of chronon size in multiversion retroactive data structures.

Conflict of interest

The authors declare no potential conflict of interest regarding the publication of this work. In addition, the ethical issues including plagiarism, informed consent, misconduct, data fabrication and, or falsification, double publication and, or submission, and redundancy have been completely witnessed by the authors.

Funding

The author(s) received no financial support for the research, authorship, and/or publication of this article.

References

Achakeev, D., & Seeger, B. (2013). Efficient bulk updates on multiversion B-trees. *Proceedings of the VLDB Endowment*, *6*(14), 1834–1845. https://dl.acm.org/doi/pdf/10.14778/2556549.2556566

Andersen, M. P., & Culler, D. E. (2016). BTrDB: Optimizing storage system design for timeseries processing. In *14th USENIX Conference on File and Storage Technologies (FAST 16)* (pp. 39–52). https://dl.acm.org/doi/10.5555/2930583.2930587

Arge, L. (2003). The buffer tree: A technique for designing batched external data structures. *Algorithmica, 37*, 1–24. https://doi.org/10.1007/s00453-003-1021-x

Arge, L., Danner, A., & Teh, S. M. (2003). I/O-efficient point location using persistent B-trees. *Journal of Experimental Algorithmics (JEA), 8*, 1–2. https://dl.acm.org/doi/pdf/10.1145/996546.996549

Arge, L., Hinrichs, K. H., Vahrenhold, J., & Vitter, J. S. (2002). Efficient bulk operations on dynamic R-trees. *Algorithmica, 33*, 104–128. https://doi.org/10.1007/s00453-001-0107-6

Bayer, R., & Schkolnick, M. (1977). Concurrency of operations on B-trees. *Acta Informatica*, *9*, 1–21. https://link.springer.com/article/10.1007/BF00263762

Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E., & O'Neil, P. (2007). A critique of ANSI SQL isolation levels. *arXiv preprint cs/0701157*. https://arxiv.org/pdf/cs/0701157

Bertino, E., Ooi, B. C., Sacks-Davis, R., Tan, K. L., Zobel, J., Shidlovsky, B., & Andronico, D. (2012). Indexing techniques for advanced database systems (Vol. 8). *Springer Science & Business Media*.

Comer, D. (1979). Ubiquitous B-tree. *ACM Computing Surveys (CSUR), 11*(2), 121–137. https://dl.acm.org/doi/pdf/10.1145/356770.356776

Copeland, G. (1980, March). What if mass storage were free? In *Proceedings of the Fifth Workshop on Computer Architecture for Non-Numeric Processing* (pp. 1–7). https://dl.acm.org/doi/pdf/10.1145/800083.802685

Dignös, A., Böhlen, M. H., Gamper, J., & Jensen, C. S. (2016). Extending the kernel of relational DBMS with comprehensive support for sequenced temporal queries. *ACM Transactions on Database Systems (TODS), 41*(4), 1–46. https://dl.acm.org/doi/pdf/10.1145/2967608

Fekete, A., Liarokapis, D., O'Neil, E., O'Neil, P., & Shasha, D. (2005). Making snapshot isolation serializable. *ACM Transactions on Database Systems (TODS), 30*(2), 492–528. https://dl.acm.org/doi/pdf/10.1145/1071610.1071615

Goodrich, M. T., Tsay, J. J., Vengroff, D. E., & Vitter, J. S. (1993, November). External-memory computational geometry. In *Proceedings of 1993 IEEE 34th Annual Foundations of Computer Science* (pp. 714–723). IEEE. https://doi.org/10.1109/SFCS.1993.366816

Haapasalo, T. K., Jaluta, I. M., Sippu, S. S., & Soisalon-Soininen, E. O. (2008, October). Concurrency control and recovery for multiversion database structures. In *Proceedings of the 2nd PhD Workshop on Information and Knowledge Management* (pp. 73–80). https://dl.acm.org/doi/pdf/10.1145/1458550.1458563

Haapasalo, T., Jaluta, I., Seeger, B., Sippu, S., & Soisalon-Soininen, E. (2009, March). Transactions on the multiversion B+-tree. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology* (pp. 1064–1075). https://dl.acm.org/doi/pdf/10.1145/1516360.1516482

Hadjieleftheriou, M., Kollios, G., Tsotras, V. J., & Gunopulos, D. (2006). Indexing spatiotemporal archives. *The VLDB Journal*, *15*, 143–164. https://doi.org/10.1007/s00778-004-0151-3

Harshit, S., & Santosh, V. (2021). B-tree versus buffer tree: A review of I/O efficient algorithms. In *Intelligent Systems: Proceedings of SCIS 2021* (pp. 417–425). https://link.springer.com/chapter/10.1007/978-981-16-2248-9_40

HBase. (2017). http://hbase.apache.org

Jaluta, I., Sippu, S., & Soisalon-Soininen, E. (2005). Concurrency control and recovery for balanced B-link trees. *The VLDB Journal, 14*, 257–277. https://doi.org/10.1007/s00778-004-0140-6

John, A., Sugumaran, M., & Rajesh, R. S. (2016). Indexing and query processing techniques in spatio-temporal data. *ICTACT Journal on Soft Computing, 6*(3), 1198–1217. https://doi.org/10.21917/ijsc.2016.0167

Kulkarni, K., & Michels, J. E. (2012). Temporal features in SQL: 2011. *ACM SIGMOD Record, 41*(3), 34–43. https://dl.acm.org/doi/pdf/10.1145/2380776.2380786

Lehman, P. L., & Yao, S. B. (1981). Efficient locking for concurrent operations on B-trees. *ACM Transactions on Database Systems (TODS), 6*(4), 650–670. https://dl.acm.org/doi/pdf/10.1145/319628.319663

Leveldb. (2011). https://github.com/google/leveldb

Lomet, D., & Salzberg, B. (1989). Access methods for multiversion data. *ACM SIGMOD Record, 18*(2), 315–324. https://dl.acm.org/doi/pdf/10.1145/66926.66956

Lomet, D., Barga, R., Mokbel, M. F., Shegalov, G., Wang, R., & Zhu, Y. (2005, June). Immortal DB: Transaction time support for SQL Server. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data* (pp. 939–941). https://dl.acm.org/doi/pdf/10.1145/1066157.1066295

Lomet, D., Hong, M., Nehme, R., & Zhang, R. (2008). Transaction time indexing with version compression. *Proceedings of the VLDB Endowment, 1*(1), 870–881. https://dl.acm.org/doi/pdf/10.14778/1453856.1453951

Ozsoyoglu, G., & Snodgrass, R. T. (1995). Temporal and real-time databases: A survey. *IEEE Transactions on Knowledge and Data Engineering, 7*(4), 513–532. https://doi.org/10.1109/69.404027

Pelkonen, T., Franklin, S., Teller, J., Cavallaro, P., Huang, Q., Meza, J., & Veeraraghavan, K. (2015). Gorilla: A fast, scalable, in-memory time series database. *Proceedings of the VLDB Endowment, 8*(12), 1816–1827. https://dl.acm.org/doi/pdf/10.14778/2824032.2824078

Rajamani, R. (2007). Oracle total recall/flashback data archive. *Technical report, Oracle*. https://forseurope.wordpress.com/wp-content/uploads/2010/01/flashback-data-archive-whitepaper.pdf

Salzberg, B., Jiang, L., Lomet, D., Barrena, M., Shan, J., & Kanoulas, E. (2004). A framework for access methods for versioned data. In *Advances in Database Technology—EDBT 2004: 9th International Conference on Extending Database Technology* (pp. 730–747). Springer. https://doi.org/10.1007/978-3-540-24741-8_42

Saracco, C. M., Nicola, M., & Gandhi, L. (2010). A matter of time: Temporal data management in DB2 for z. *IBM Corporation*, 7. https://cs.ulb.ac.be/public/_media/teaching/infoh415/a_matter_of_time.pdf

Spdx. (2020). GNU general public license v2.0: B+tree basics. https://github.com/torvalds/linux/blob/master/include/linux/btree.h

Taghva, M. R., Mansouri, T., Feizi, K., & Akhgar, B. (2016). Fraud detection in credit card transactions; using parallel processing of anomalies in big data. *Journal of Information Technology Management, 8*(3), 477–498. https://doi.org/10.22059/jitm.2016.57818

Tao, Y., & Papadias, D. (2001). The MV3R-tree: A spatio-temporal access method for timestamp and interval queries. In *Proceedings of Very Large Data Bases Conference (VLDB), 11–14 September, Rome*. https://hdl.handle.net/1783.1/168

Tian, R., Zhai, H., Zhang, W., Wang, F., & Guan, Y. (2022). A survey of spatio-temporal big data indexing methods in distributed environment. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing, 15*, 4132–4155. https://doi.org/10.1109/JSTARS.2022.3175657

Wang, L., Cai, R., Fu, T. Z., He, J., Lu, Z., Winslett, M., & Zhang, Z. (2018, April). Waterwheel: Realtime indexing and temporal range query processing over massive data streams. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)* (pp. 269–280). IEEE. https://doi.org/10.1109/ICDE.2018.00033

Widmayer, P., Becker, B., Gschwind, D. I. S., Ohler, D. W. I. T., & Seeger, B. (1996). An asymptotically optimal multiversion B-tree. *Very Large Data Bases Journal*. Retrieved from https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=6f4cf2dd0d8af70af29ae857353badad12426183

Yang, F., Tschetter, E., Léauté, X., Ray, N., Merlino, G., & Ganguli, D. (2014, June). Druid: A real-time analytical data store. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data* (pp. 157–168). https://dl.acm.org/doi/pdf/10.1145/2588555.2595631

Zhang, R., & Stradling, M. (2010). The HV-tree: A memory hierarchy aware version index. *Proceedings of the VLDB Endowment, 3*(1-2), 397–408. https://dl.acm.org/doi/pdf/10.14778/1920841.1920894

Zhao, X., Lam, K. Y., Zhu, C., Chow, C. Y., & Kuo, T. W. (2021). MVLevelDB: Using log-structured tree to support temporal queries in IoT. *IEEE Internet of Things Journal, 9*(10), 7815–7825. https://doi.org/10.1109/JIOT.2021.3113994

Bibliographic information of this paper for citing:

Verma, Santosh Kumar; Saha, Suman & Goel, Sanjay (2025). Incorporating Retroactive Operations in Large temporal Databases Using Retroactive B-Tree. *Journal of Information Technology Management*, 17 (Special Issue), 87-107.
https://doi.org/10.22059/jitm.2025.102923