SILVA: Spatial Index Library with Versioned Access (technical report)

Bo Huang
UC Riverside
bhuan102@ucr.edu

Yan Gu
UC Riverside
ygu@cs.ucr.edu

Ahmed Eldawy
UC Riverside
eldawy@ucr.edu

Yihan Sun
UC Riverside
yihans@cs.ucr.edu

Abstract

The continuous collection and sharing of geospatial data have become central to modern data-driven research and applications. While this data enables advanced analytics, it also introduced new challenges in efficient data collection, auditing, and querying. Most importantly, with the continuous data change, researchers need to be able to access past versions of the data not just the most recent version for reproducibility of results and consistency. This paper presents SILVA, a library of main-memory spatial indexes designed for versioned access to highly dynamic data. We formally define the problem of multiversion spatial data management and propose multiple indexes in SILVA to address this problem. SILVA builds on state-of-the-art functional data structures to support immutable versions with atomic batch updates. SILVA is highly parallel and is crafted for spatial data with minimal memory footprint. Experiments on synthetic and real data show that SILVA achieves up-to two orders of magnitude faster performance than state-of-the-art indexes, peaking at 112 million updates per second, while reducing memory usage by 70%. Existing spatial algorithms can be seamlessly applied on any version on the proposed index with no noticeable overhead on performance.

PVLDB Reference Format:
Bo Huang, Yan Gu, Ahmed Eldawy, and Yihan Sun. SILVA: Spatial Index Library with Versioned Access (technical report). PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/ItakEjgo/mvzd.

1 Introduction

Data generation and collection have become increasingly efficient and versatile, greatly accelerating the frequency at which datasets are updated and shared. Open datasets, such as daily crime reports [20], traffic accident records [40], or demographic changes [21], illustrate the continuous ingress of data. However, this rapid growth presents substantial challenges in data collection and processing, requiring robust mechanisms to ensure accuracy, consistency, and integrity during ongoing updates.

In data collection, handling high-frequency updates from diverse sources adds complexity to storage and indexing operations.

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

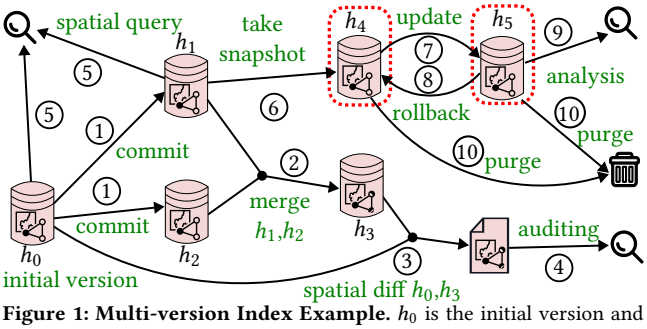

Figure 1: Multi-version Index Example. h_0 is the initial version and there are totally six versions. All versions are indexed and can be accessed and *modified*. Multiple queries and updates can be processed simultaneously.

These data streams involve a combination of insertions, updates, and deletions, requiring the efficient management of such continuous modifications. Additionally, detecting and signaling potential conflicts among these sources is crucial for maintaining data accuracy, which often requires examining historical states of the data. Data auditors, for instance, must frequently track changes in specific geographical regions, underscoring the importance of effective data tracking across versions. Due to this complexity, most data providers will make only the latest dataset available for download and provide no access to older versions, limiting reproducibility and data integrity.

On the analysis side, integrating multiple datasets, such as demographics, population statistics, and crime data, presents significant challenges. Analysts often need to focus on data specific to particular geographical regions to ensure meaningful insights. Moreover, for consistent and reproducible analyses, users need access historical versions of the data rather than being limited to the most recent copy. For example, a social scientist studying the impact of a natural disaster will want to access that data before and after the event.

To overcome these challenges, an ideal system should efficiently index not only the current data but also historical versions. It should minimize redundancy by preserving unchanged portions efficiently, thereby optimizing storage and retrieval. Furthermore, the system should support treating multiple updates as a single atomic operation to maintain consistency. It must also enable seamless historical data navigation and conflict detection, allowing users to focus on specific geographical regions and temporal contexts without handling irrelevant data.

Most existing spatial and spatio-temporal indexes emphasize in-place updates, while ignoring history tracking [7, 9, 28, 32]. Many recent database systems, including main-memory databases [42, 50, 53], use multiversioning as a concurrency control mechanism to avoid locking, but queries typically only access the most recent data, lacking essential data auditing features such as version comparisons

and merges [24, 36, 38, 46, 57, 59].

Recently, functional (immutable) data structures (FDSs) have been used as an effective tool for supporting versions [22, 23, 54–56]. These data structures operate much like their mutable counterparts but guarantee immutability by generating a new version on each update through copy-on-write, rather than modifying the structure in place. With a unique handle for each version, queries on any version are as efficient as queries on a standard index. Modern designs can carefully provide high parallelism and efficient space usage despite the cost of versioning [22, 23]. While functional data structures offer some level of multiversion tracking, existing studies are not efficient enough for spatial datasets [22, 23, 54–56].

This paper introduces SILVA[1], a spatial index library with versioned access. First, we define the problem of multiversion spatial data management which includes data collection, change tracking, and efficient query processing. Then, we formalize the set of operations that an index structure should support to handle this problem. After that, we propose two *in-memory* indexes, PaC-Z tree and MVZD-tree, that are designed on the concept of *functional data structures* to support these operations. In particular, PaC-Z tree introduced in Sec. 4 extends one-dimensional indexes with a space-filling curve to support 2D and 3D data but it suffers from scalability issues due to spatial misalignment. The MVZD-tree redesigns the index with spatial awareness and employs the *spatial invariance* property to improve spatial alignment across multiple versions of the index. We build on this idea to provide highly-parallel algorithms for batch insertion, deletion, and update which efficiently create a new version of the index where all the modifications appear as one atomic update. We continue by adding efficient algorithms for spatial diff operation between versions, branch and merge operations, spatial range count and report, k nearest neighbors, and spatial join. Users can run these operations on any version of the index without incurring any additional computational overhead. Finally, to control memory usage, users can delete any old versions that are no longer needed and the system will automatically reclaim the non-accessible parts of the index. We ran extensive large-scale scalability experiments against state-of-the-art multi-version spatial indexes and showed that our solutions outperform existing indexes by up to *two orders* of magnitude in processing time and use only *one third* memory space.

The contributions of this paper are summarized below:

(1) We introduce the problem of spatial multi-version data processing and define the set of index operations for this problem.
(2) We develop SILVA, a spatial index library with main-memory indexes for spatial multi-version data processing based on stable spatial partitioning and functional data structures.
(3) We provide highly-parallel algorithms for all the fundamental operations on the proposed index structure.
(4) We run an extensive experimental evaluation against state-of-the-art index structures using synthetic and real data to verify the scalability of the proposed solution.

The rest of this paper is organized as follows. Sec. 2 highlights the background and preliminaries for our work, and Sec. 3 presents the problem definition of versioned spatial data management and operations it needs to support. The approaches in SILVA are discussed in Sec. 4 and Sec. 5. Sec. 6 analyze the version management operation costs, and Sec. 7 presents our experimental results. We discuss related works in Sec. 8, and conclude this paper in Sec. 9.

[1]Silva in Latin means "forest" representing a collection of trees.

2 Background and Preliminaries

2.1 Parallel Computation Model

This paper models parallel algorithms using the *binary fork-join* paradigm [5, 16, 17]. This model extends the random access machine (RAM) model with a *fork* instruction that starts two parallel child threads; The parent thread waits until both are done. A *parallel-for* can be easily implemented in a logarithmic number of fork steps. Under this model, the computation can be viewed as a directed acyclic graph (DAG) with vertices representing computation and edges representing parent-child relationships.

We analyze the parallel algorithms using *work-span* analysis. In the DAG model, the *w*ork (W) represents the total number of operations in all vertices, i.e., the computational cost assuming serial execution. The *depth* or *span* (D) is the length of the longest path in the DAG, representing the critical path of dependencies. Under a randomized work-stealing scheduler, the parallel execution completes in $O(D + W/\rho)$ time with probability at least $1 - W^{-c}$, for any constant $c > 0$ using ρ processors [16, 17, 31].

2.2 Preliminaries for Spatial Data

The discussion of this paper focuses on 2D points in Euclidean space but can be easily extended to low-dimensional points. Each point is identified by a unique ID and a location (x, y). To improve access to spatial data, several indexes have been developed, including *quadtrees* [28], k-d trees [9], and *R-trees* [8, 32, 37, 45, 51, 57, 58]. They generally work by grouping nearby points in a hierarchical structure to allow a quick top-down search.

One common indexing method is to use a *space filling curve* (SFC), e.g., Z-Curve [44], with a traditional one-dimensional index. Z-Curve works by traversing the space with a Z-shaped curve that linearizes all the points to produce a total ordering of all locations. Computationally, this is equivalent to interleaving the bits of the point coordinates to produce a single location on the Z-curve.

2.3 Functional Data Structures (FDSs)

Functional Data Structures (FDSs) are a class of *immutable* data structures in which data cannot be modified in place. Instead, updates such as insertions and deletions produce a new version of the structure, preserving the original one. To avoid duplicating the entire structure, tree-based FDSs typically use *path-copying* (or *copy-on-write*), where only modified nodes along the update path are copied and unmodified nodes are shared between versions.

FDSs naturally support versioned access by maintaining previous states as immutable snapshots, enabling git-style branching. In tree-based FDSs, any modification propagates from the point of change to the root, resulting in a new root node that uniquely identifies the updated version. This design ensures both *atomicity* and *consistency*, as updates are visible only after the root is replaced in a single atomic step. Additionally, since each version maintains a valid tree structure, traditional read-only operations such as range queries can be executed on any version without requiring any changes to the algorithm and without incurring extra overhead.

Several database systems have adopted path-copying to build

one-dimensional indexes [1, 3, 11, 29, 55]. For example, P-tree [55] was proposed to support hybrid transactional and analytical processing (HTAP), and PaC-tree [22] improved this by employing leaf compression to reduce storage with minimal performance impact.

Despite their advantages, FDSs can incur substantial storage overhead due to repeated path-copying. Many systems mitigate this by restricting queries and updates to the latest version and discarding older versions over time. However, managing storage becomes more complex with spatial and multidimensional data, where preserving structural similarity across versions is more difficult. To the best of our knowledge, our work is the first to generalize functional data structures to spatial indexes, while also supporting cross-version operations such as *spatial-diff* and *merge*.

3 Problem Definition

This section introduces the problem of multiversion spatial data management and defines the set of fundamental operations that should be supported. Let P be the spatial dataset (i.e., a set of points), which may change dynamically over time. We consider the modifications come in *batches* (single modification can be viewed as a special case). A *version* refers to a certain state of index on P. The goal of maintaining a *multi-version spatial index* for P is to organize all versions of P in a data structure that enables fast access, modification and querying of any historical version (not necessary the latest one). A modification needs to preserve the previous version and create a new version in history.

In this paper, we assume each version is represented and accessed by a *handle*. In SILVA, where index is a path-copying tree structure, the handle is just the root pointer of the tree.

Interface of a Multi-version Spatial Index. We now define the interface for a multi-version spatial index. A multi-version spatial index should provide the *same* set of operations that a regular spatial index provides, such as construction, updates, and spatial queries. It should also support multi-version control operations, which are useful in data collection, auditing, and analytics, such as acquiring or releasing a history version. For a multi-version spatial index, a query on any version should be *as easy as* query a regular spatial index.

We start with presenting the ***standard spatial queries*** that should be supported on any single version of the spatial index.

RangeCount(h, r): Given a version handle h and an *axis-aligned rectangle region* r, a range count query returns the *number of points* in r (i.e., a number).

RangeReport(h, r): Given a version handle h and a *axis-aligned rectangle region* r, a range report query returns *all points* in r (i.e., a set of spatial points).

Knn(h, p, k): Given a version handle h, a query point p, and an integer k, a k nearest neighbor query returns k closest points to p in version h_i.

In addition, a versioned spatial index should also be able to handle operations related to multi-versioning, which we refer to as ***version operations***. Such operations include constructing the initial version, modifying a version to generate a new version, merging/joining multiple versions, and finding the difference between versions. Below are detailed definitions for these operations:

SpatialJoin(h_1, h_2, r, w): Given two version handles, h_1 and h_2, an *axis-aligned rectangle region* r, and a distance upper-bound w, a spatial join (report) query returns all pairs of points (one from h_1 and one from h_2) in query region r with distance less than w.

Build(P): Given a spatial dataset P, build initializes a new index and returns the initial version handle.

BatchInsert(h, I): Given a version handle h and point set I, insert points in I to h and return the new version handle.

BatchDelete(h, D): Given a version handle h and point set D, delete points in D from h and return the new version handle.

Commit(h, I, D): Given a version handle h, an insert point set I and delete point set D, delete points in D from h. Commit returns the new version handle after deleting points in D from h and inserting points in I to h.

SpatialDiff(h_1, h_2, r): Given two version handles h_1, h_2, and an *axis-aligned rectangle region* r, return a pair of point sets, which stands for the minimal change of points (insert or delete) within region r that transform from h_1 to h_2.

Merge(h_1, h_2): Merge two existing versions h_1 and h_2, return the new version handle where the points in the new version is the *union* of points in h_1 and h_2.

Purge(h): Delete and *free* the memory space for the unused version h, without influencing other existing versions.

Notice that it is straight-forward to implement these operations by making a full copy of the index with each operation. However, this would be extremely inefficient due to the redundant storage of non-modified parts. The challenge is to implement these operations while reducing redundancy between index versions as further shown in the next sections.

4 The Design of the PaC-Z Tree

In this section, we illustrate the design of a functional spatial tree by extending the existing functional tree used for 1D data. Here we consider the PaC-tree, introduced in 2022 [22], which is widely recognized as the state-of-the-art approach.

The core idea is to apply a space-filling curve—in this case, the Z-curve—to map high-dimensional data into one dimension. This idea enables the PaC-tree to maintain the order of all points efficiently. Hence, we refer to this design as the PaC-Z tree. We note that while space-filling curves offer an effective linearization of high-dimensional data, they do not naturally support spatial queries, including both classic operations such as range queries and version-based queries, like spatial difference. Therefore, more sophisticated designs are required here.

We address this challenge through two approaches. The first one is via the coding of the curves—we can also map the query ranges based on the Z-curve, which allows some pruning during queries. We refer to this approach as PaC-Z-Code tree. However, this mapping for the queries does not fully preserve spatial locality, which means that query efficiency is generally lower than that of classic spatial indexes such as kd-trees. The second approach is to dynamically augment the bounding box of all points within each subtree and store this metadata at each node. This design is reminiscent of structures like binary R-trees or bounding volume hierarchies (BVHs). However, despite the similarities, the tree remains organized and updated according to the 1D ordering induced by the Z-curve, not the logic of an R-tree. We refer to this variant

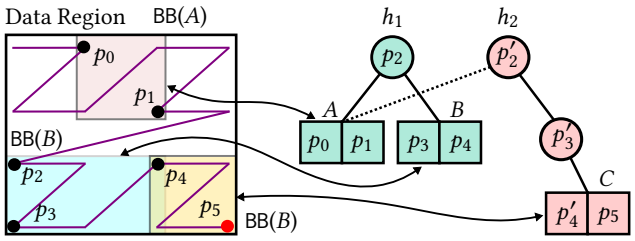

Figure 2: PaC-Z-BB tree Example. The left side shows the point locations and bounding boxes (BB). The right side shows the structure of two tree versions h_1 and h_2 before and after inserting p_5, respectively.

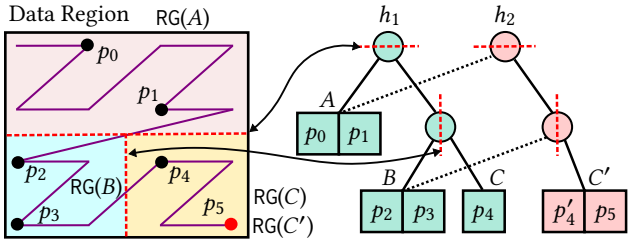

Figure 3: MVZD-tree Example. The data region is used to define the Z-curve and the partition lines. The left side shows the point locations and regions (RG). The right side shows two versions h_1 and h_2 before and after inserting p_5, respectively.

as the PaC-Z-BB tree, as bounding boxes are included for all nodes. Between these two approaches, PaC-Z-BB tree uses more space due to the additional bounding boxes stored, but it generally delivers much better query performance.

The PaC-tree. The design of the PaC-tree is based on the P-tree [12, 15, 56], which is considered a standard parallel binary search tree (BST). P-trees support a full interface for construction, single and batch updates, set operations, and many 1D queries, using either the AVL tree, the red-black tree, the treap, or the weight-balance tree as the underlying structure. All tree updates use a so-called "concat-based framework" in a divide-and-conquer manner, which supports high parallelism. Despite all the benefits, a major issue of using P-trees in version operations is the space usage. PaC-trees [22] are built based on P-trees but with leaf wrapping—any subtree with size no more than a given threshold (usually 32) is flattened into a leaf node with all nodes stored in a flat array. Thus, the auxiliary space to maintain the search structure is negligible compared to the data size. Although the change seems small, the entire concat-based framework needs to be redesigned to adapt to this change for achieving both theoretical and practical efficiency.

We now discuss how to adapt the design of PaC-trees to support spatial indexes. As mentioned before, the high-level idea here is simple—we can use a space-filling curve to order spatial data in a 1D total order, and maintain the data using the same approach as in the PaC-tree. However, we will not be able to query on this tree since no spatial information is kept. To deal with it, we show two ways to incorporate the spatial information into the tree structure.

Our PaC-Z-Code tree. Our first attempt to facilitate spatial queries on PaC-Z trees is to utilize the spatial information from the Z-curve. For each tree node, we in addition store the Z-value for each point in this node. With this information, when traversing the tree, we know the range of the Z-curve of each subtree by subdividing the range recursively. The benefit of this approach is that for a range report query, we can compute its range on the Z-curve by testing its four corners. The traversal of a subtree is pruned when it has no overlap with the query range, which significantly accelerates query speed. We refer to this approach as the PaC-Z-Code tree. Unfortunately, we do not see an equivalent pruning strategy for other spatial queries.

Our PaC-Z-BB tree. Our second approach, the PaC-Z-BB tree, is inspired by the idea of R-trees, which also have relaxed structural information in the tree partitioning. In an R-tree, a bounding box is augmented in each tree node, indicating the total region for all points in this subtree, which can be used in pruning the searches. For instance, in a range query, if the query box has no overlap with the bounding box of a tree node, then the entire subtree can be skipped. While the PaC-Z-BB tree uses a very different idea from the R-tree in maintaining the tree structure, the same idea for spatial queries can be used.

To support bounding boxes efficiently, we use the augmentation in PaC-tree, which stores the bounding box coordinates within each tree node as the *augmented value*. In the algorithm, when a tree node is created, or involved in a rotation, the new bounding box will be automatically recalculated in constant time.

The spatial queries on PaC-Z-BB trees are almost identical to those on R-trees. Range reports and k-nn searches can be pruned when the points in a subtree cannot be candidates. Range counts can be further accelerated when the entire subtree is within the query range. Spatial diff queries are more subtle. Technically we can use the same approach as MVZD-tree which will be discussed later in Sec. 5 , but due to frequent rotations, the performance is generally less ideal.

5 The Design of the MVZD-Tree

The main idea of the MVZD-tree is based on using the quadtree as its skeleton, which is a strict space-partition tree that subdivides the space efficiently. First, due to the natural of quadtree, MVZD-tree does not need to explicitly maintain bounding boxes in the tree nodes, making MVZD-tree space-efficient (thus faster). Another major benefit is that, the structure of the MVZD-Tree is history independent, i.e., the state of the tree is fully determined by the set of elements in the tree. Note that the new algorithms for the versioned spatial queries introduced in this paper rely on the *space invariance* property, so fewer structural changes of the tree facilitates such queries significantly. However, we are not aware of any existing design of functional spatial-partition data structures in the literature with parallel updates and version operations like *spatial diff* and *merge*. We will show our design for MVZD-tree in this section to facilitate efficient updates and queries.

MVZD-Tree Nodes. As a binary spatial-partition tree, the MVZD-tree contains *interior nodes* that store: 1) two pointers to the subtrees, and 2) the subtree sizes (i.e., augmented values) for faster query performance, and *leaf nodes* that store a list of no more than B (practically 32) points sorted by Z-values. Note that saving space

Algorithm 1: BUILD

Input: A point set P
Output: The initial version handle of P
Parameter: c: leaf capacity, b: Z-value binary representation length.

```
1  Function Build(P)
2      if P = ∅ then return nullptr
3      calculate Z-values for points in P and sort them by their Z-values
4      h ← BuildNode(P, 0, |P|, b)                    // h is the version handle
5      return h
6  Function BuildNode(P, l, r, b)
7      if b = 0 or r − l ≤ c then
8          x ← new leaf node stores points {P_l, ..., P_r}.
9          return x
10     mid ← the minimum i ∈ [l, r) with the b-th bit in P_i as 1
11     In Parallel:
12         L ← BuildNode(P, l, mid, b − 1)             // Build left subtree
13         R ← BuildNode(P, mid, r, b − 1)             // Build right subtree
14     Let x be a new interior node with L and R as children
15     Update augment values for x
16     return x
```

Algorithm 2: BATCHINSERT

Input: A version handler h, a point set I to be inserted
Output: New version handle after insertion
Parameter: c: leaf capacity, b: Z-value binary representation length.

```
1  Function BatchInsert(h, I)                          // BatchInsert entry
2      if I = ∅ then return h
3      calculate Z-values for points in I and sort them by Z-values
4      return BatchInsertNode(h, I, 0, |I|, b)
5  Function BatchInsertNode(h, I, l, r, b)
6      if h = nullptr then return Build (I, l, r, b)
7      x ← new tree node by copying h
8      if x is leaf then
9          if b = 0 or |x.points| + r ≤ c + l then
10             x.points ← x.points after inserting {I_l, ..., Ir}
11             return x                                // merge points to x.points
12         else
13             I′ ← x.points ∪ {I_l, ..., I_r}
14             return Build (I′, 0, |I′|, b)
15     mid ← the minimum i ∈ [l, r) with the b-th bit in I_i as 1
16     In Parallel:
17         if l < mid then x.lc ← BatchInsertNode(h.lc, I, l, mid, b − 1)
18         if mid < r then x.rc ← BatchInsertNode(h.rc, I, mid, r, b − 1)
19     Update augmented values for x
20     return x
```

usage is crucial for better performance of spatial indexes in general, and for multi-version indexes in particular. Therefore we do not store bounding boxes or partition hyperplanes, since they can be computed on-the-fly in query processing.

Tree Structure. As mentioned above, the MVZD-tree has a binary structure which can be adapted to any dimensions. Following the idea of the *quadtree*, MVZD-tree uses the spatial median to partition the space alternately for x- and y-axes. Hence, MVZD-tree is history independent since the partition hyperplanes are always the spatial median, and partitions in all levels are deterministic. This is a desirable property for multi-version access since it improves spatial alignment across versions resulting in reduced storage and better query performance.

Operations. The rest of this section presents our design for all index operations for MVZD-tree based on the structure above.

1. BUILD. Alg. 1 shows the pseudocode of BUILD function. If there are no input points, a `nullptr` will be returned in Line 2. Otherwise, we first calculate Z-values for all points, and then sort them using a parallel sorting algorithm. After that, we calculate the bit length of the datasets and use it for space partition as described in the partition strategy. Then, line 4 calls BUILDNODE function to build the initial version and return the handle.

BUILDNODE function creates a tree node for a given point set. line 9 checks whether the input points can fit in one leaf node. If yes, it returns a leaf node with all input points. Otherwise, line 10 splits the points according to the b-th bit of Z-values by searching for the first value with that bit set to 1 using binary search. After that, the process continues by recursively building the left and right subtrees *in parallel* for the two sublists, with corresponding subregions (indicated by bit $b-1$). Note that there is no data movement since the two subranges of subtrees can be obtained by $[l, mid)$ and $[mid, r)$, respectively. Then, we create an interior node with the subtrees in line 14 and update its augmented value in line 14.

2. BATCHINSERT and BATCHDELETE. MVZD-tree processes modifications in *batches* as described in Sec. 3. Alg. 2 shows the procedure of BATCH_INSERTION. h is a version handle of the version to be modified, and I is the point set to be inserted. In line 7, we first make a *copy* of h to make changes without affecting the original version. Similar to BUILD, we calculate Z-values for I and sort them by Z-values. Then, BATCHINSERTNODE is called to update the copied handle h' for I with Z-value bit length b.

BATCHINSERTNODE will make a copy of the given handle and return the copied handle after dealing with the insertion. As shown in line 6, if the given handle is `nullptr`, we call Alg. 1 to build a subtree and return the subtree root as the new handle. Then, we will copy the given handle h since it needs to be modified. Note that the copied handle x will share unmodified child pointers with h. line 8 checks whether copied tree node x is a *leaf node*. If so, we should either insert the points directly into that node or split it if an overflow happens. If the inserted points list can fit with the existing points in copied leaf node x, then line 11 simply merges them since they are both sorted, and returns a new leaf node. Otherwise, it is an *overflow* case, and line 12 handles it by merging all the points into a new sorted list I' and building a new subtree using Alg. 1. The rest of the algorithm handles the case where we insert into a non-leaf node. line 15 partitions the inserted point list at the bit of position b. Then, the insertion procedure of child pointers will be run in parallel. Note that a child pointer will be copied only if insertions happen in its corresponding region. Finally, in line 19 the augmented values of current interior node will be updated according to the two (possibly copied) child pointers.

Alg. 3 shows the batch *deletion* algorithm, which follows a similar idea to batch insertion, with some differences for the leaf case and after-delete *underflow* handling. Similar to batch insert, the entry function BATCHDELETE sorts the set D and initiates the recursive call to BATCHDELETENODE. To deal with deletion, BATCHDELETENODE will copy the given handle and return the copied handle after deletion processing. Lines 8-9 delete points from a leaf node by running the set difference operation for sorted lists (similar to merge). If it becomes empty, it frees the node and returns a `nullptr`. Notice that removing points from a leaf node might cause an underflow to its parent node which will be checked by the calling function as shown below. If h points to an internal node, lines 11-16 split

Algorithm 3: BatchDelete

Input: A version handle h, a point set D to be deleted.
Output: New version handle after deletion
Parameter: c: leaf capacity, b: Z-value binary representation length.

```
1  Function BatchDelete(h, D)                                   // BatchDelete entry
2      if D = ∅ then return h
3      calculate Z-values for points in D and sort them by Z-values
4      return BatchDeleteNode(h, D, 0, |D|, b)
5  Function BatchDeleteNode(h, D, l, r, b)
6      x ← new tree node by copying h
7      if x is leaf then
8          x.points ← x.points after removing {D_l, ..., D_r}
9          if x.points = ∅ then free node x
10         return x
11     mid ← the minimum i ∈ [l, r) with the b-th bit in D_i as 1
12     In Parallel:
13         if l < mid then
14             x.lc ← BatchDeleteNode(h.lc, D, l, mid, b − 1)
15         if mid < r then
16             x.rc ← BatchDeleteNode(h.rc, D, mid, r, b − 1)
17     if lc = nullptr and rc = nullptr then free node x
18     else if x.lc = nullptr then x ← x.rc                     // lift the right child
19     else if x.rc = nullptr then x ← x.lc                     // lift the left child
20     else if |x.lc.points| + |x.rc.points| ≤ c then           // form a new leaf
21         x ← new leaf node stores points in x.lc.points ∪ x.rc.points
22     update augmented values for x
23     return x
```

Algorithm 4: SpatialDiff

Input: Two version handlers h_1, h_2, and a query rectangle r
Output: The insertion, deletion point sets can transform h_1 to h_2
Parameter: b: Z-value binary representation length

```
1  Function SpatialDiff(h1, h2, r)
2      ⟨I, D⟩ ← SpatialDiffNode(h1, h2, r, b)
3      return ⟨I, D⟩
4  Function SpatialDiffNode(h1, h2, r, b)
5      box ← current tree node bounding box
6      I ← ∅, D ← ∅
7      if h1 = h2 or box ∩ r = ∅ then return ⟨I, D⟩   // Case I: identical nodes
8      else if h1 = nullptr and h2 ≠ nullptr then   // Case II: one nullptr
9          I ← range report results of h2 in query region r
10     else if h1 ≠ nullptr and h2 = nullptr then
11         D ← range report results of h1 in query region r
12     else if h1 is leaf and h2 is leaf then              // Case III: two leaves
13         ⟨I, D⟩ ← point set difference for h1.points and h2.points in r.
14     else if h1 is leaf and h2 is not leaf then // Case IV: one leaf, one interior
15         ⟨I, D⟩ ← point set difference for h1.points and h2 in r
16     else if h1 is not leaf and h2 is leaf then
17         ⟨I, D⟩ ← point set difference for h1 and h2.points in r
18     else
19         In Parallel:                                    // Case V: two interiors
20             ⟨I_L, D_L⟩ ← SpatialDiff(x.lc, y.lc, r, b − 1)
21             ⟨I_R, D_R⟩ ← SpatialDiff(x.rc, y.rc, r, b − 1)
22         ⟨I, D⟩ ← ⟨I_L ∪ I_R, D_L ∪ D_R⟩
23     return ⟨I, D⟩
```

the set deleted point list and recursively delete data from both subtrees. Underflow handling starts at line 17 after the recursive calls. In line 17, if both child pointers are `nullptr`, h will be freed and replaced with `nullptr`. When only one child pointer exists, we will lift that pointer in line 18 or line 19. Otherwise, we will check whether two child nodes can be merged into a leaf node in line 21. Finally, if no underflow is detected, line 22 updates the interior node augmented values based on the new, possibly copied, child pointers.

The efficient underflow handling helps keep MVZD-tree compact and space efficient. It can also help with reducing the depth, which speeds up spatial queries.

3. Purge. MVZD-tree supports purging an existing version without influencing other stored versions. Recall that the reference counter represents how many times a tree node is used throughout the entire history. To remove a version h, Purge recursively decrements the reference counters of tree nodes in h starting from the root. When a reference counter becomes zero, we will safely purge the tree node since it is no longer needed in any history.

4. Commit. Given a version handler h, insertion point set I, and deletion point set D, the Commit function returns the new version handle after modification. We first call BatchDelete to delete D from h and get an intermediate version h^t. Then, we use BatchInsert to insert I into h^t, generating the final version h'. Before returning h', we garbage collect the intermediate version h^t to ensure no extra space is used except for the new version.

5. SpatialDiff. Given two version handlers, h_1 and h_2, and a query rectangle r, SpatialDiff computes the set differences of points within r between the two versions. It first gets the bit length of Z-values, and then calls SpatialDiffNode to compute the differences.

As shown in Fig. 4, the algorithm processes the spatial diff by advancing two pointers simultaneously and recursively comparing the subtrees they reference. line 5 computes the bounding box based on the partitioning strategy. Due to the spatial invariance property, both pointers traverse identical paths, ensuring the bounding boxes are always aligned. This property significantly improves efficiency by enabling early pruning, as further detailed below. If the bounding box does not intersect the query region, r, no changes are reported. The remaining five cases are as follows:

Case I: identical nodes: If the two pointers reference the same node (line 7), the region is unchanged and no differences are reported. Fig. 4 shows this case by moving from root along path ①.

Case II: one `nullptr`: If one pointer is null, this indicates either a newly inserted or a completed deleted region. The non-null node's points are reported as insertions or deletions according to which side is null line 8 and line 10. This is trivial case since it can only happen when insert to empty version or deleting all points from a version.

Case III: two leaves: If the two pointers reference difference leaf nodes, line 12 computes the inserted/deleted points for the points in two leaves by a linear scan (all points are sorted in Z-values). Fig. 4 demonstrates this case by following the path ② → ④. In this case, since all points are sorted, we can report the changes by linear scanning the point sets once.

Case IV: one leaf, one interior: If two pointers refer to a leaf node and an interior node respectively, we continue to traverse down to the leaf nodes on interior node side and comparing them to the list of points in the leaf node. As we traverse down the non-leaf node, we partition the set of points in the leaf node to match. Note that the leaf node partition does not make structural changes, and the partitioning does not involve any data movement since points are sorted. In example Fig. 4, when the two pointers traverse along tree path ② → ③.

Case V: two interiors: line 19 handles the two interior nodes case by

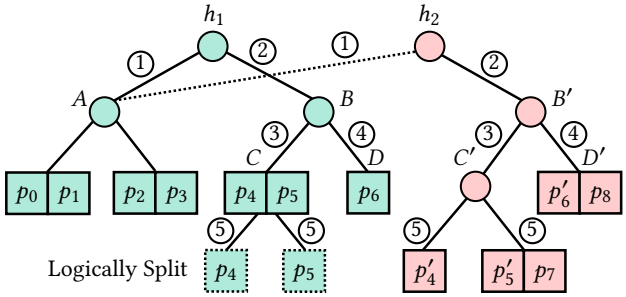

Figure 4: MVZD-tree Spatial Diff Example. h_1 and h_2 are two version handles. The number on the tree path indicates spatial diff traversal order.

Algorithm 5: MERGE

Input: Two version handles h_1, h_2, and conflict resolving strategy S.
Output: The handler of new merged version.
Parameter: r_g: The global region (entire space).

```
1  Function Merge(h1, h2, S)                                  // Merge entry
2      h_b ← common base version handle of h1 and h2
3      In Parallel:
4          ⟨I1, D1⟩ ← SpatialDiff(h_b, h1, r_g)   // get changes from h_b to h1
5          ⟨I2, D2⟩ ← SpatialDiff(h_b, h2, r_g)   // get changes from h_b to h2
6      ⟨I, D⟩ ← no conflict updates by linear scan I1, I2, D1, D2
7      ⟨I′, D′⟩ ← conflict updates by linear scan I1, I2, D1, D2
8      h_t ← h_b
9      if strategy S is provided then                          // solve conflict
10         h_t ← Commit I′, D′ to h_t according to S
11     h ← Commit(h_t, I, D)                                   // commit not conflict points
12     GarbageCollect (h_t)
13     return h
```

recursively calling SpatialDiffNode for the child pointers of the two interior nodes. Due to spatial invariance, the two subtrees are aligned; hence, we only need to compare the children in the same direction, i.e., left-to-left and right-to-right. Fig. 4 shows this case by following path ②.

Finally, line 22 merges the point set changes and returns the complete spatial diff result.

6. Merge. Given two version handlers, h_1 and h_2, and a conflict-solving strategy S, the Merge function returns a new version handle containing all points in h_1 and h_2. The spatial merge operation is implemented based on SpatialDiff and Commit. Firstly, we calculate the common base version h_b for h_1 and h_2 in line 2. The base version can be achieved by tracking back along the version history. e.g., when versions are managed in tree-like style, we can get h_b by their *least common ancestor*.

Then in line 4 and line 5, we call SpatialDiff to compute the diff of the base version to both h_1 and h_2, respectively. After that, we merge the modification sets together in line 6 to obtain the non-conflict and conflict modifications, respectively. Since all points are sorted by Z-values, this procedure can be achieved by linear scanning the point sets. In line 9, if a conflict-solving strategy is provided, we will commit the conflict updates according to the given strategy and obtain an intermediate version. Finally, we call Commit to commit the non-conflict changes in line 11, and return the new merged version handle. Last but not least, the intermediate version will be purged, and there is no extra space usage except for the merged version.

6 Theoretical Analysis

This section presents the work and span analysis for version operations in SILVA, i.e., *build*, *batch update*, and *commit*. We omitted spatial queries and merge since they are highly data sensitive and the analysis on the worst case becomes trivial. Since all three indexes use *leaf wrapping* technique, we use B (usually 32) to represent the leaf node capacity and assume it is a constant.

6.1 PaC-Z tree

Theorem 6.1 (Build Cost). *Given a set of points P with size n, PaC-Z tree can be built in $O(n \log n)$ work with $O(\log n)$ span.*

Proof. The build procedure first compute Z-values for all points in $O(n)$ work with $O(\log n)$ span, then sort them in $O(n \log n)$ work with $O(\log n)$ span (using optimal parallel comparison-based sorting), and finally build a perfect balanced binary search tree use divide-and-conquer manner in $O(n)$ work with $O(\log n)$ span. Therefore, build work is $O(n + n \log n + n) = O(n \log n)$, and build span is $O(\log n + \log n + \log n) = O(\log n)$. □

Lemma 6.2 (Batch Insert/Delete Cost). *Given a PaC-Z tree T built upon point set P with size n, and a point set P' with size m. Batch insert/delete P' to/from T can be processed in $O(m \log n)$ work with $O(\log m \log n)$ span.*

Proof. W.l.o.g., let's consider batch insert P' to T. The analysis of batch delete is similar. The batch insert procedure contains three parts: 1) partition P' into subtrees until meet the base cases; 2) process base cases (append inserted points to existing leaf; dealing with leaf overflow); and 3) rebalance after insertion. Firstly, node copy for a tree node has $O(1)$ cost. Since both interior and leaf nodes have constant number of fields. For the partition part, T has $O(\log n)$ levels, and in each level the partition procedure cost $O(m)$ work with $O(\log m)$ span. Therefore, the partition work in total is $O(m \log n)$ and the partition span is $O(\log m \log n)$. For the base case, we need to touch all updated points in the worst case. Therefore the work is $O(B + m) = O(m)$ and span is $O(\log(B + m)) = O(\log m)$. For the rebalance part, PaC-Z tree uses concat primitive [56] for rebalancing. Each concat can combine two trees with size x and y in $O(\log \frac{x}{y})$ work and span $(x > y)$. There are $O(m)$ leaf nodes need to be rebalanced, and thus $O(m)$ concat will be called. Each concat costs at most $O(\log \frac{n}{m})$ work and span (w.l.o.g. assume $n > m$). Therefore the total work for rebalance is $O(m \log \frac{n}{m})$, and the span is $O(\log \frac{n}{m})$.

By adding the three parts together, we obtain that insertion work is: $O(m \log n + m + m \log \frac{n}{m}) = O(m \log n)$, and insertion span is $O(\log m \log n + \log m + \log \frac{n}{m}) = O(\log m \log n)$, bounded by partition. □

Theorem 6.3 (Commit Cost). *Given a PaC-Z tree T build upon point set P with size n, delete point set D with size m_1, and insert point set I with size m_2. A new version T' can be committed in $O((m_1 + m_2) \log n)$ work with $O(\log(m_1 m_2) \log n)$ span.*

Proof. Commit consists of three steps. Batch deletion, batch insertion, and purge the intermediate version. First we use Lem. 6.2 to get the costs for batch insertion and batch deletion. For batch deletion, we create a temporary version T_t by deleting D from T, which costs $O(m_1 \log n)$ work and $O(\log m_1 \log n)$ span. Then we batch insertion I into T_t. Since m_1 points are deleted from P, the size of T_t is $n - m_1$. Therefore the batch insertion costs work $O(m_2 \log(n - m_1))$

and span $O(\log m_2 \log(n - m_1))$. Finally, we purge T_t, which contains $n - m_1$ points. The work and span for purge are the same as traverse all tree nodes in parallel, i.e., $O(n - m_1)$ work and $O(\log(n - m_1))$ span.

Combine all three parts together, we obtain that the commit procedure costs with $O(m_1 \log n + m_2 \log(n - m_1) + (n - m_1)) = O((m_1 + m_2)\log(n))$ work and $O(\log m_1 \log n + \log m_2 \log(n - m_1)) = O(\log(m_1 m_2) \log n)$ span. □

6.2 MVZD-tree

Given a spatial dataset P with n points, let $d_{\max}$ ($d_{\min}$) be the furthest (closest) pairwise distance in P, respectively. The aspect ratio of P, denoted by Δ_P, is defined to be $\Delta_P = \frac{d_{\max}}{d_{\min}}$.

Theorem 6.4 (Build Cost). *Given a dataset P consists of n points, a MVZD-tree can be built on P in $O(n \log n)$ work with $O(\log n \log \Delta_P)$ span.*

Proof. The first step in the build algorithm is to compute and sort by Z-values which requires $O(n \log n)$ work and $O(\log n)$ span when using optimal parallel comparison-based sorting.

To compute the cost of the tree construction step, we observe that MVZD-tree is a binary tree with $O(n)$ leaf nodes (each leaf contains at least 1 point), and the total number of tree nodes is $O(n)$. For each tree node, a binary search with cost $O(\log n)$ is performed to partition the points (see line 10). Therefore, the total *work* is $O(n \log n)$. For the *span*, the height of MVZD-tree is determined by the aspect ratio Δ_P. Since the space is halved for each level, the height of MVZD-tree is $O(\log \Delta_P)$. For each level of recursion, a binary search is performed to partition the points into two parts, therefore the span is $O(\log \Delta_P \log n)$. □

Lemma 6.5 (Batch Insert/Delete Cost). *Given a MVZD-tree T maintaining dataset P with size n, and a point set P' with size m. Batch inserted/deleted P' to/from T can be processed with $O(m \log \Delta_P)$ work and $O(\log \Delta_P \log m)$ span.*

Proof. We show the proof of batch insertion here, and the proof for deletion is similar. Firstly, the tree after inserting P' into P has the same shape as directly constructing a MVZD-tree on top of $P \cup P_I$ since MVZD-tree is *history independent*. Given the fixed node size, the node copy procedure costs $O(1)$ operations. Therefore the tree height after batch insertion is $O(\log \Delta_{P \cup P'})$.

We analyze the work first. The tree height of T is $O(\log \Delta_P)$. For each level of the tree, we need to partition entire P' once. The partition procedure has $O(m)$ work and $O(\log m)$ span. Therefore, the total work of batch insertion is $O(m \log \Delta_P)$.

For the span part, consider the longest path during the insertion. It touches $O(\log \Delta_P)$ tree nodes (a chain from root to leaf), and for each touch it requires a $O(\log m)$ binary search to partition (as shown in line 15). Therefore, the span is $O(\log \Delta_P \log m)$. □

Theorem 6.6 (Commit Cost). *Given a MVZD-tree T built for point set P with size n, delete point set D with size m_1, and insert point set I with size m_2. A new version T' can be committed in $O((m_1 + m_2) \log \Delta_P)$ work with $O(\log m_1 m_2 \log \Delta_P)$ span.*

Proof. We consider the three steps for commit: 1) Batch delete D; 2) Batch insert I; and 3) Purge the intermediate version.

For the first step, we create a temporary version T_t by batch deleting D, which costs $O(m_1 \log \Delta_P)$ work with $O(\log \Delta_P \log m_1)$ span according to Lem. 6.5. T_t is the same shape as directly built on point set $P \setminus D$. Therefore, the second step costs $O(m_2 \log \Delta_{P \setminus D})$ work and $O(\log \Delta_{P \setminus D} \log m_2)$ span. To purge T_t, we just need to traverse T_t in parallel with $O(n - m_1)$ work and $O(\log \Delta_{P \setminus D})$ span. Since $|P \setminus D| < |P|$, then $O(\log \Delta_{P \setminus D} \log m_2) = O(\log \Delta_P \log m_2)$.

By combining the three parts, the commit costs $O(m_1 \log \Delta_P + m_2 \log \Delta_P + n - m_1) = O((m_1 + m_2) \log \Delta_P)$ work and $O(\log \Delta_P \log m_1 + \log \Delta_P \log m_2 + \log \Delta_P) = O(\log m_1 m_2 \log \Delta_P)$ span. □

7 Experiment

7.1 Summary of Experimental Results

We conduct extensive experimental evaluation on both *real* [47] and *synthetic* [30] datasets to illustrate the efficiency of SILVA. Compared with MVR-tree from [39], SILVA achieves up to **82**× (**84**×) speedup in build time with only **25.6%** (**30.5%**) memory space on synthetic (real) datasets, respectively. For batch updates, SILVA can achieve up to **145**× speedup running sequentially. For single-versioned spatial queries, MVZD-tree achieves the best or comparable to the best performance for all tested queries among all tested methods, which includes well-recognized single-versioned R-tree in *boost* [18]. PaC-Z tree show slightly worse performance than MVZD-tree due to its spatial misalignment. For spatial diff query, MVZD-tree employs our novel spatial diff algorithm, which gains up to **17**× speedup compared with other baselines for small amount of modifications.

In a word, SILVA outperforms existing multi-version spatial index significantly in terms of operation time, query time, and memory footprint. Meanwhile, it achieves the best or comparable to the best performance as traditional single-version index.

7.2 Experimental Setup

Setup. All experiments were conducted on a 96-core machine with four-way Intel Xeon Gold 6252 CPUs and 1.5 TB memory. All indexes are implemented in `C++` and compiled by `g++` 14.2.1 with `-O3` flag with parallel primitives from *ParlayLib* [13]. We use *numactl* for parallel experiments, which interleaves memory across CPUs on NUMA architectures. Results are reported by the *average of three test rounds* after a warm-up round.

Baselines. We test SILVA against existing approaches: boost-R, MVR-tree and MV3R-tree. boost-R is the R-tree from *Boost* [18] with quadratic split, which does not support versions. Leaf capacities of all approaches are set to 32. MV3R-tree is a multi-version R-tree proposed in [57]. The original paper does not provide public code, and thus we provide our own implementation for comparison. MVR-tree is a variant of MV3R-tree implemented in the *libspatialindex* [39] library. Both MVR-tree and MV3R-tree support versions via version chains, which only allows for modifying the latest version to obtain new versions, and thus does not support more complicated versioned operations such as merging, branching, or spatial diff. All baselines are sequential.

Dataset. We run experiments on both real-world and synthetic data. For real data, we use *Open Street Map* (OSM) [47]. For synthetic data, we generate them using `Uniform` and skewed `Varden` [30] distributions with coordinate range $[0, 10^7]$. Used dataset statistics are discussed at the beginning of each evaluation section. Detailed

Table 1: Dataset Statistics

Dataset Type	Name	# of Points
Real World [47]	Uzbekistan	10.4M
	Malaysia-Singapore-Brunei	27.2M
	Kazakhstan	29.3M
	Thailand	37.8M
	Nepal	67.7M
	Japan	95.1M
Synthetic	Uniform	0.1M to 1B
	Varden [30]	0.1M to 1B

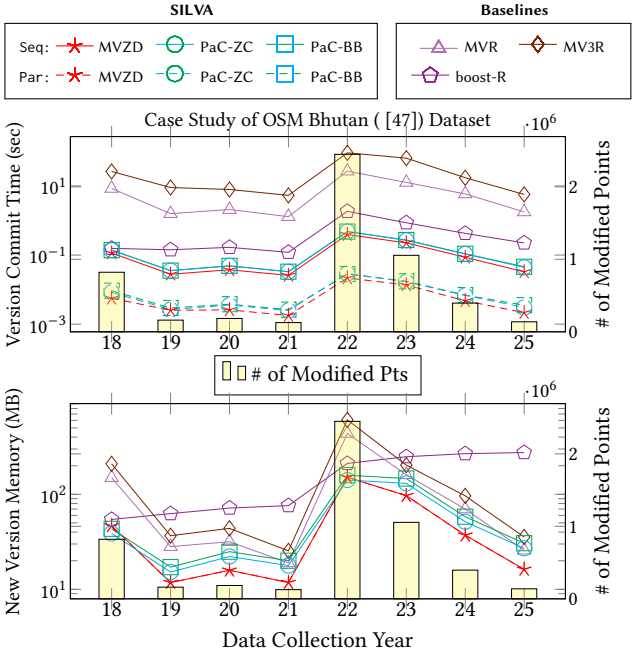

Figure 5: Real-world case study on *Version Commit Time* and *Memory Usage* (OSM Bhutan from 2018-2025). Lower the better. *Solid* lines are measured in *sequential*. Bars show the number of modified points in each batch. *Dashed* lines are measured in *parallel* (96-cores).

Table 2: Statistics for Case Study Data (osm-bhutan, 2018–2025). Highlighted lines indicate years with burst updates.

Year	Point Insertions	Point Deletions	Point Updates
2018	821,488	-	-
2019	133,978	8,800	18,104
2020	138,602	7,785	37,460
2021	91,946	27,093	7,758
2022	2,116,010	115,180	217,249
2023	755,301	242,215	57,398
2024	312,513	35,170	48,188
2025	115,273	5,395	17,292

dataset statistics are shown in Tab. 1.

7.3 End-to-end Performance with a Real Case Study

To evaluate the end-to-end performance with a real workload, we used the yearly commits from *OSM Bhutan* [47] dataset. The real update statistics are shown in Tab. 2. Yellow lines show the year with burst of modifications. We started with the initial dataset from 2018 and simulated the history of the yearly changes until 2025. All changes in one year are applied as one batch commit with a mix of insertions, deletions, and updates. We measured both the commit time and the memory usage after each commit. Results are shown in Fig. 5. The yellow bars show the total number of records in each commit. We also show the sequential running time of SILVA since the baselines are all sequential.

Commit Time. The upper part of Fig. 5 shows the commit time in seconds in log scale. For all tested systems, the commit time is proportional to the batch size, as expected.

Even the sequential version of SILVA consistently outperforms all baselines by up to **68**×. boost-R is slightly faster than MV3R-tree and MVR-tree, but it does not support versions. Using parallelism, MVZD-tree, PaC-Z tree, PaC-Z-BB tree achieves up to **1235**×,**916**×, **950**× speedup compared with MVR-tree and MV3R-tree, and up to **89**×,**75**×, **63**× speedup compared with boost R-tree.

In SILVA, MVZD-tree has the best overall performance since it does not need rebalance as mentioned in Sec. 5.

Memory Usage. The lower part of Fig. 5 shows the extra memory used for the new version after each update. R-tree has to fully copy the old index to preserve the old version, and thus performs the worst, with memory growing quadratically. Such high memory overhead makes it less competitive than others in supporting versions. For this reason, we exclude boost-R in future experiments.

All other techniques are version-aware, with memory increasing proportionally to the batch size. MVR-tree and MV3R-tree consume more memory due to large node sizes [39]. Beyond the initial version (2018), MVZD-tree consistently achieves the smallest memory footprint. This is attributed to the spatial invariance property in MVZD-tree that reduces tree structural changes between versions, which is only observed on the second and subsequent versions. Interestingly, while unbalanced trees are typically associated with increased memory due to deeper structures and more nodes, in the context of multi-version spatial indexing, we observe the opposite trend with reduced memory usage over time with real-world data.

7.4 Build (Initial Version) Performance

Fig. 6 shows *initial version* build performance on up to one billion records. Since the baselines are sequential, we also report our sequential time for comparison.

Build Time. According to Fig. 6 (a) and (b), SILVA outperforms the SOTA multi-version spatial index MVR-tree and MV3R-tree by up to two orders of magnitude in both `Uniform` and `Varden` distributions due to the efficient parallel build algorithm Alg. 1. MVR-tree and MV3R-tree failed with one billion points as they ran out of time (more than three hours). Among SILVA, MVZD-tree benefits from its small memory footprint by keeping tree nodes small. For `Uniform` data, MVZD-tree has the best performance since `Uniform` is the best-case scenario where MVZD-tree is balanced. For `Varden` data, all SILVA indexes have similar performance.

Memory Usage. As shown in Fig. 6 (c) and (d), the compact node representation in SILVA results in a 70% reduction in memory usage. In SILVA, MVZD-tree has the best memory usage with `Uniform` data since the tree becomes naturally balanced. The skewed `Varden` distribution results in a slightly increased memory usage due to tree unbalance, but as shown earlier, MVZD-tree can save memory in subsequent commits.

7.5 Batch Updates, Commit, and Merge

Batch Insert/Delete. We start with a fixed index of 100M points and perform a batch insert/delete with the batch size varying from

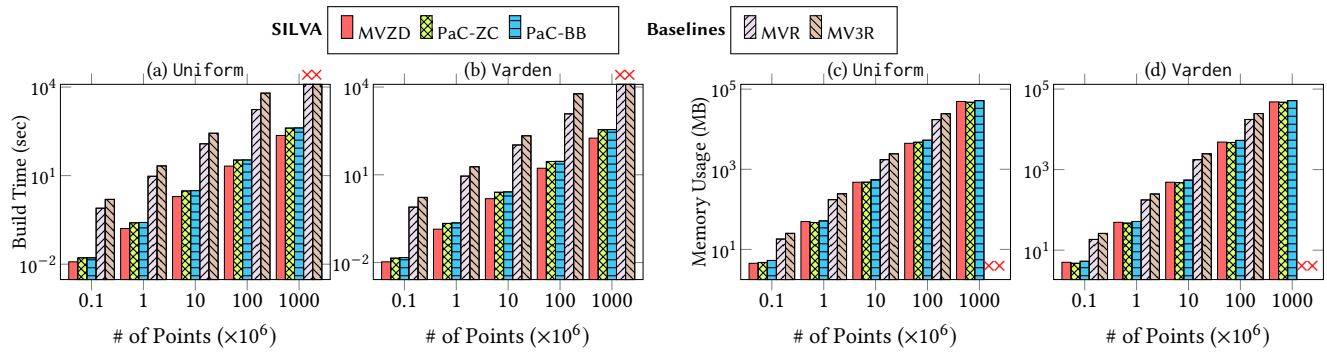

Figure 6: Build Time (Sequential) and Build Memory evaluation on synthetic dataset, lower the better. (a), (b) show the *build time* in *seconds*. (c), (d) show the *Memory Usage* in *Megabytes*. All axes are in *log* scale. *Note*: ✕ means *timeout* or *not applicable*.

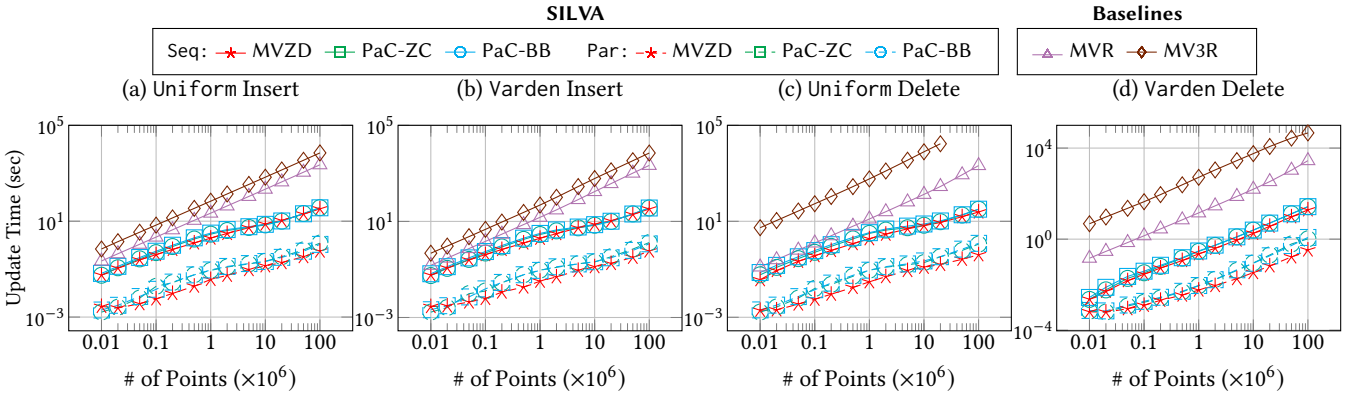

Figure 7: Insert/Delete evaluation on synthetic initial version with 100M points, lower the better. (a), (b) show **insert** time in **seconds**. (c), (d) show **delete** time in **seconds**. All axes are in **log** scale. **Solid** lines run in **sequential**, and **Dashed** lines run in **parallel** (96-cores).

10^4 to 10^8 on both `Uniform` and `Varden` datasets. The results on both synthetic distributes are similar.

Based on results in Fig. 7, we observed that, for small batch sizes, all indexes work well sequentially as the batch update optimizations are not yet observable. When the batch size gets larger, SILVA becomes orders of magnitude faster than MVR-tree and MV3R-tree since SILVA creates a single version for the entire batch rather than record-to-record updates. MVZD-tree is consistently faster among all SILVA indexes since it does not require tree balancing and contains more shared nodes due to the spatial invariance property. When running in parallel on 96-cores, MVZD-tree has up to 67 × self-speedup, which is the fastest in SILVA.

Commit and Merge. To test the performance of the commit operation, we used a mixed workload of 50-50 insertions and deletions as shown on Fig. 9. We observe similar behavior with MVZD-tree being the fastest among other SILVA indexes.

For merge, we use the same modification point set as commit. We generate two new versions to avoid version conflicts, one takes all insertions and the other takes all deletions. Fig. 10 shows the sequential merge time for SILVA. One interesting observation is that MVZD-tree is two orders of magnitude faster than PaC-Z tree for small amount of changes. This is due to the efficient spatial-diff algorithm that utilizes spatial invariance to quickly prune large portions of unchanged regions. Practically speaking, users tend to use the merge operation when there are only a few changes which is what the algorithm in MVZD-tree is optimized for. As the amount of changes increase to eventually reach 100% of all points, there becomes less room for improvement and the performance of all methods converges, with MVZD-tree taking slightly longer due to the overhead of the spatial-diff algorithm as detailed shortly.

7.6 Spatial Diff

To evaluate spatial diff, we first commit a new version generated similar to the commit experiment in Sec. 7.5. That is, we start with an index containing 100M points. After that, we apply an update operation with varying sizes, from 10^4 to 10^6, with a 50-50 mix of insert and delete operations. Then, we generate random query regions and classify them based on the number of points they contain prior to the update: $[0, 100)$ for *small* regions and $[100, 10{,}000)$ for *medium* regions. The `Uniform` results are similar to `Varden` shown in Fig. 8.

We compare SILVA to three baselines. Two of them, namely ZD-RQ and ZD-DT are variants we implemented to show the effectiveness of proposed techniques. Two baselines run a traditional range query search on the two versions, sort the results, and compute the difference; ZD-RQ runs on Zd-tree and the other runs on R-tree. The third baseline, ZD-DT, implements the same spatial join algorithm as MVZD-tree, but runs on non-versioned Zd-tree, i.e., without node sharing.

Based on Fig. 8, we observed that: MVZD-tree is the clear winner for small amounts of changes, i.e., less than 10%, and is close to the

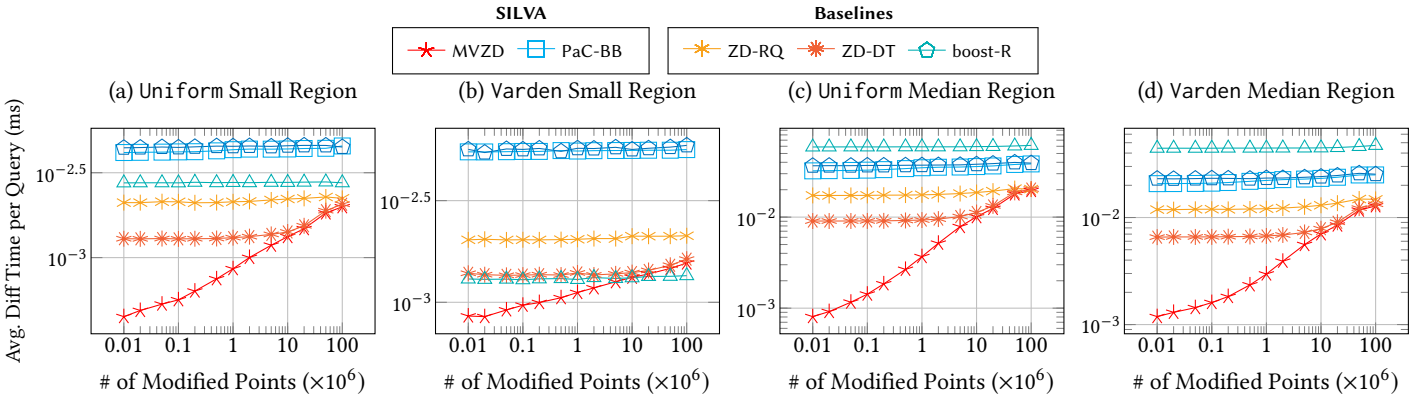

Figure 8: Spatial diff (sequential) on synthetic datasets with 100M points for small/median regions, lower the better. Modified points consists of 50% insertion and 50% deletion. Each small/median region contains $[0, 100)$/$[100, 10000)$ points before modification, respectively. All axes are in **log** scale.

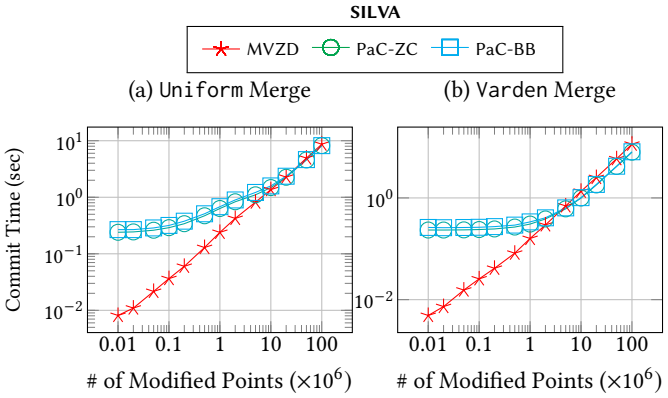

Figure 9: Merge time (sequential) on synthetic datasets with 100M points, lower the better. Modified points consists of 50% insertion and 50% deletion. All axes are in **log** scale.

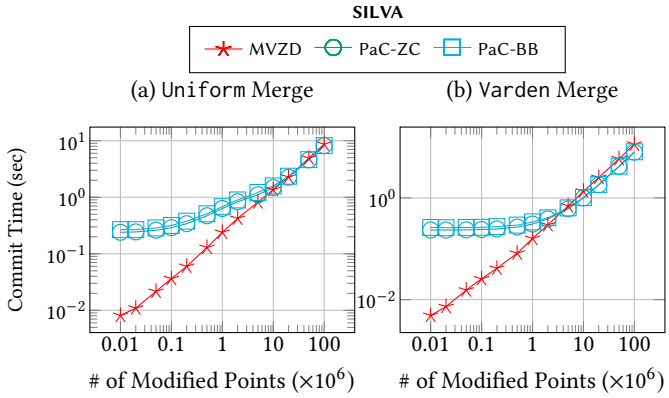

Figure 10: Merge time (sequential) on synthetic datasets with 100M points, lower the better. Modified points consists of 50% insertion and 50% deletion. All axes are in **log** scale.

best with larger amounts of changes. This shows the effectiveness of the proposed spatial diff algorithm (Alg. 4). Comparing ZD-DT to ZD-RQ, we can observe the effectiveness of the dual tree traversal algorithm even without node sharing. Interestingly, boost-R performs well for small regions where the range search runs very quickly and produces a small result that can be compared quickly. However, it becomes the slowest for medium regions where the results become very large. PaC-BB does not perform well in spatial diff, due to overlapping bounding boxes introduced by the *Z-curve* layout, which limits the effectiveness of pruning during range reporting.

To conclude this part, MVZD-tree has overall the best performance of spatial diff queries, especially for detecting differences for small number of modifications.

7.7 Parallel Speedup

We use the single-core execution time of MVZD-tree as our reference value, and report the relative speedups of SILVA across different number of cores. For build, we tested on 10^8 points. For merge, we start with an initial version of 10^8 points and apply 10% modifications as earlier described in Sec. 7.5. The results are shown in Fig. 11. Due to limited space, we report the results for *build* and *merge* but omit the results for *insert/delete* which are similar to build.

As shown in Fig. 11 (a), all indexes in SILVA achieve linear speedup in terms of building time as the number of cores grows to 48. MVZD-tree continues to achieve high speedup due to its minimal tree node size design. For merge, Fig. 11 (b) shows that MVZD-tree has a slightly less speedup with a large number of threads due to a bottleneck with parallel memory allocation. Since MVZD-tree traverses two trees simultaneously, there will be frequent memory allocation across the traversal procedure when modified points are sparsely distributed. This is a common bottleneck in main-memory databases [24] that require a specialized memory allocator to overcome but solving it is outside the scope of this paper.

7.8 Standard Spatial Queries

One of the advantages of SILVA is that every version can be treated as a traditional spatial index. Therefore, we can port any existing spatial queries to SILVA without introducing extra query overhead, e.g., filter records that do not belong this version. In this paper, we implemented *range report*, *range count*, *kNN*, and *spatial join*, on SILVA to demonstrate its extensibility on single-versioned queries.

Fig. 12 (a) shows a scatter plot of the running time of *range report* by the query output size. MVZD-tree is largely as efficient as the highly optimized boost-R even though it provides access to all version of the index. This shows that MVZD-tree does not

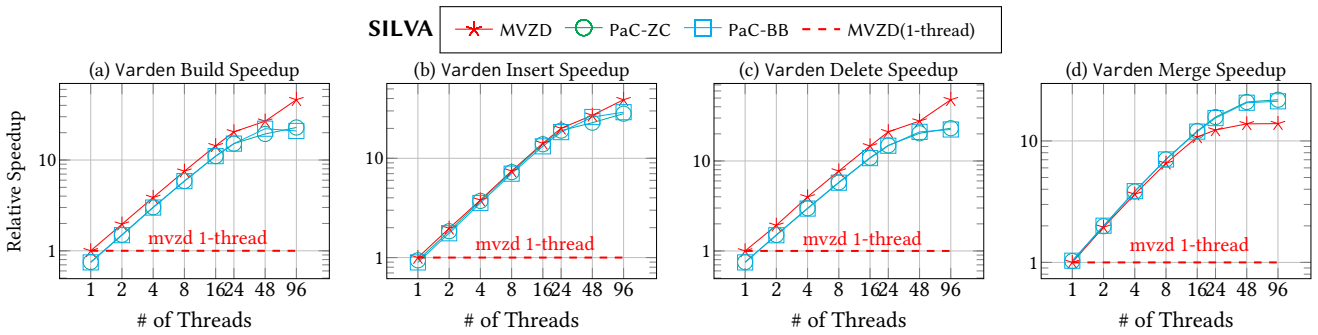

Figure 11: Bo: TODO: remove two of them, keep merge. **Parallel speedup w.r.t.# of threads on synthetic Varden datasets with 100M points, higher the better.** The relative speedup is calculated w.r.t. MVZD tree 1-thread (shown in the dashed horizontal line) performance.

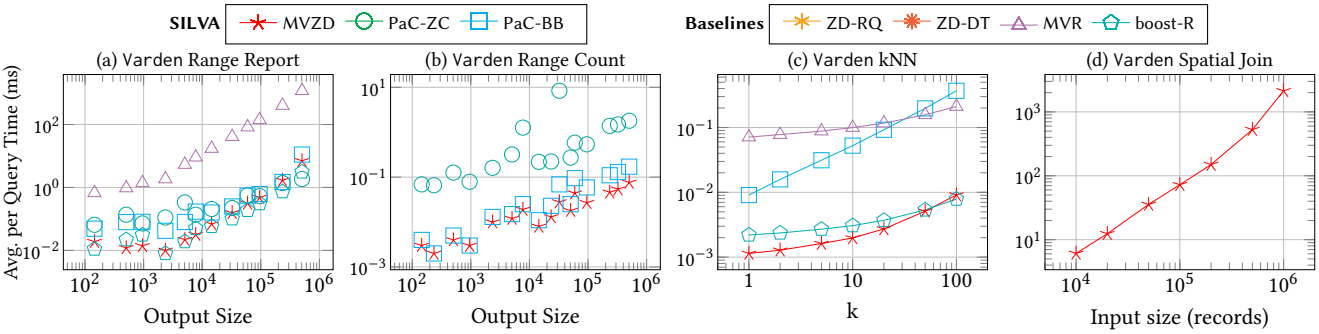

Figure 12: Single-version query (sequential) evaluation on synthetic datasets with 100M points, lower the better. *x*-axis is the parameter studied (i.e., # of points, output size, or *k*). All axes are in **log** scale. For (a), # of points is chosen before modifications. All queries are read-only and can be parallelized.

have to trade off performance to provide all its unique features. Other indexes in SILVA provide a comparable performance while MVR-tree is two orders of magnitude slower.

Fig. 12 (b) shows similar results for the *range count* query. MVR-tree and R-tree do not have a specialized range count algorithm and behave similar to range report so we excluded them. MVZD-tree and PaC-Z-BB tree achieves **92**× and **10**× self-speedup respectively compared with range report due to subtree size augmentation.

Fig. 12 (c) shows the results of the *kNN* queries while varying *k* from 1 to 100. MVZD-tree provides faster or comparable performance to the optimized boost-R. Both achieve at least an order of magnitude speedup over other techniques.

We also implemented spatial join (report) operation based on *synchronized traversal* [33] to find all pairs of points within a distance upper bound in two indexes for MVZD-tree. Fig. 12 (d) shows the spatial join time as the input size increases on Varden, with spatial predicate distance set to 50,000. MVZD-tree can provide spatial join report results in 2 seconds on two indexes with 10^6 points, which shows its good extensibility to support existing spatial queries.

7.9 Real Data Evaluation

To verify the generalization of the results with real data distributions, we ran all the experiments with real datasets from Open Street Map (OSM) with up to 100M records. Due to limited space, Fig. 13 shows the results of index build, commit, and merge.

Fig. 13 (a) shows that SILVA keeps its efficient speedup over MVR-tree with up to two orders of magnitude speed up in index construction. Fig. 13 (b) shows the index size after construction where the compact design of SILVA helps in significantly reduce the memory usage of the index as compared to MVR-tree. While MVZD-tree takes up slightly more initial space than other indexes in SILVA, it saves space on subsequent updates.

Fig. 13 (c) and (d) show the performance of commit and merge with OSM Japan dataset. This experiment is setup in the same way described earlier where we start with an index that contains the full dataset of nearly 100M points. Then, we select subsets of increasing sizes and split each one into 50% insertions and 50% deletions. The results confirm our earlier findings with MVZD-tree being the faster for commit. For merge, MVZD-tree is super fast for small amounts of changes and becomes similar to other techniques when we have 10% or more changes.

To summarize, SILVA works as efficient with real data as it does with synthetic data.

8 Related Work

This section covers the related work on traditional spatial indexes, the use of multi-version for concurrency control, multi-version spatial indexes, and functional data structures.

Spatial indexes have been playing an important role in data systems in both academia [2, 26, 27] and industry [42, 43, 48, 49, 52] for many decades. Traditional **single-version spatial indexes** are designed to speed up some spatial queries for one datasets, e.g., *quadtrees* [28], *k-d trees* [9], *segment trees* [10], and *R-trees* [8, 32]. To utilize multi-core architecture, some work introduced parallelism on spatial indexes, such as *bulk loading operations* in *R-trees* [4], *parallel R-trees* [34], parallel construction, batch updates in *Zd-trees* [14] and *k-d trees* [41]. All that work were limited to single-version operations where there is only one single view of the index where

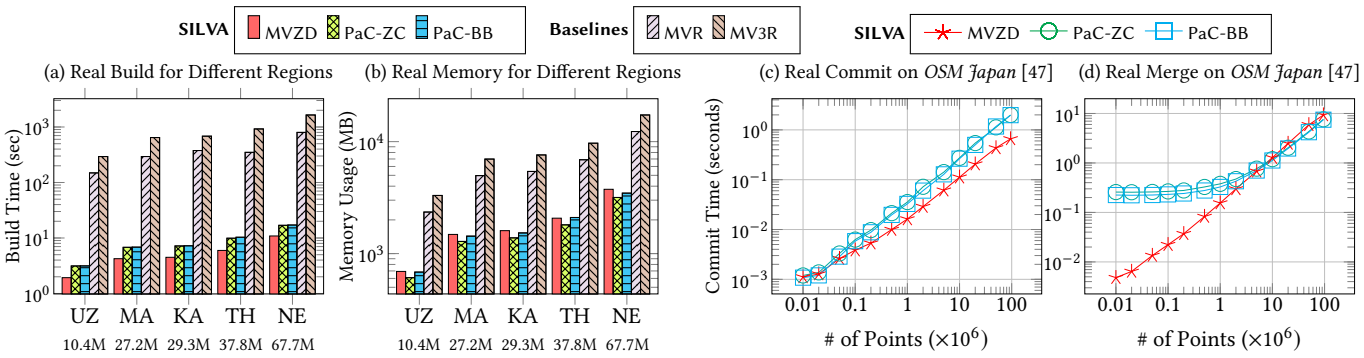

Figure 13: Real-world OSM data evaluation (in different regions), all measures are lower the better.

all updates and queries operate on. They utilize in-place updates as a mean to achieve efficiency which defies the idea of versioned access that our work focuses on.

With the rise of main-memory databases, there has been recent work on **multi-version concurrency control (MVCC) indexes**. These indexes achieve higher level of parallelism by replacing traditional lock-based concurrency control mechanism with modern latch-free techniques. The key idea is to utilize *path-copying* which makes a copy of modified portions of the index instead of overwriting existing data [25]. By eliminating overwriting, concurrency control can be simplified at the cost of occasionally having to repeat some operations.

There exists some **multi-versioned spatial index** exploration in the literature. In [35], the versions are managed by a combination of *segment tree* and *R-tree*. The *historical R-trees* [45] extended the **path-copying** technique to support *R-trees* to support multi-versioned spatial data. *MV3R-tree* [57] builds a combination of an *MVR-tree* (an *R-tree* variant for *MVB-tree* [6]), for historical single-version access, and a 3DR-tree, to access multiple versions simultaneously. While the methods above provide some multi-version access, all updated were limited to the latest version with no support for advanced queries, such as *what-if* analysis [19] or *history correction* (e.g., rollback).

A more structured way to support multiversioning is through **Functional data structures**. Parallel Augmented Map (PAM) [56] maintains a multi-versioned map using a functional binary search tree with join-based algorithms [15]. Parallel Compressed tree (PaC-tree) [22] further improves on this idea by wrapping tree leaf nodes into compressed blocks to save space without significantly sacrificing update and query performance. PAM was also used to build a multicore in-memory HTAP DBMS [55] that achieves snapshot isolation. Parallel k-d tree [41] utilizes functional data structures to build a highly-parallelized multi-dimensional index but it is limited to only one version. None of the work above achieves full versioned access, i.e., query and update, on spatial data.

This work utilizes functional data structures to build a spatial index with full versioned access. Unlike existing methods, users can update and query any version of the index with the options to commit and merge the changes or rollback and purge them. The proposed work focuses on batch updates and high parallelism allowing it to outperform all existing methods.

9 Conclusion

In this paper, we proposed the versioned spatial data access problem and SILVA, a spatial index library with version access using the idea of spatial invariance and functional data structures. SILVA supports both single- and multi-versioned spatial data management, with high parallelism. On the one hand, SILVA keeps all versions indexed. Accessing any one of them is the same as accessing a traditional single-versioned spatial index, without introducing the overhead of touching records in other versions. On the other hand, SILVA provides fundamental interfaces like spatial diff, commit, and merge. These interfaces are crucial building blocks in managing versions. Update operations in SILVA can be highly parallelized. We implemented three indexes in SILVA: PaC-Z-Code tree, PaC-Z tree, and MVZD-tree, and conducted extensive experiments to evaluation their effectiveness. According to the experimental results, SILVA consistently achieves the best or comparable to the best performance with significant improvement on memory footprint.

References

[1] 2015. Lightning memory-mapped database manager (LMDB). https://dbdb.io/db/lmdb.

[2] Sattam Alsubaiee, Yasser Altowim, Hotham Altwaijry, Alexander Behm, Vinayak R. Borkar, Yingyi Bu, Michael J. Carey, Inci Cetindil, Madhusudan Cheelangi, Khurram Faraaz, Eugenia Gabrielova, Raman Grover, Zachary Heilbron, Young-Seok Kim, Chen Li, Guangqiang Li, Ji Mahn Ok, Nicola Onose, Pouria Pirzadeh, Vassilis J. Tsotras, Rares Vernica, Jian Wen, and Till Westmann. 2014. AsterixDB: A Scalable, Open Source BDMS. *Proceedings of the VLDB Endowment (PVLDB)* (2014).

[3] J Chris Anderson, Jan Lehnardt, and Noah Slater. 2010. *CouchDB: the definitive guide: time to relax.* " O'Reilly Media, Inc.".

[4] Lars Arge, Klaus H Hinrichs, Jan Vahrenhold, and Jeffrey Scott Vitter. 2002. Efficient bulk operations on dynamic R-trees. *Algorithmica* 33 (2002), 104–128.

[5] Nimar S Arora, Robert D Blumofe, and C Greg Plaxton. 2001. Thread scheduling for multiprogrammed multiprocessors. *Theory of Computing Systems (TOCS)* 34, 2 (2001), 115–144.

[6] Bruno Becker, Stephan Gschwind, Thomas Ohler, Bernhard Seeger, and Peter Widmayer. 1996. An Asymptotically Optimal Multiversion B-Tree. *The VLDB Journal* 5, 4 (1996), 264–275.

[7] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. 1990. The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 322–331.

[8] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. 1990. The R*-tree: An efficient and robust access method for points and rectangles. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 322–331.

[9] Jon Louis Bentley. 1975. Multidimensional binary search trees used for associative searching. *Commun. ACM* 18, 9 (1975), 509–517.

[10] Jon Louis Bentley. 1977. *Algorithms for Klee's rectangle problems.* Technical Report. Carnegie-mellon University.

[11] Philip A Bernstein, Colin W Reid, and Sudipto Das. 2011. Hyder-A Transactional Record Manager for Shared Flash.. In *Conference on Innovative Data Systems*

Research (CIDR), Vol. 11. 9–20.
[12] Guy Blelloch, Daniel Ferizovic, and Yihan Sun. 2022. Joinable Parallel Balanced Binary Trees. *ACM Transactions on Parallel Computing (TOPC)* 9, 2 (2022), 1–41.
[13] Guy E. Blelloch, Daniel Anderson, and Laxman Dhulipala. 2020. ParlayLib — a toolkit for parallel algorithms on shared-memory multicore machines. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 507–509.
[14] Guy E Blelloch and Magdalen Dobson. 2022. Parallel Nearest Neighbors in Low Dimensions with Batch Updates. In *Algorithm Engineering and Experiments (ALENEX)*. SIAM, 195–208.
[15] Guy E. Blelloch, Daniel Ferizovic, and Yihan Sun. 2016. Just Join for Parallel Ordered Sets. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.
[16] Guy E. Blelloch, Jeremy T. Fineman, Yan Gu, and Yihan Sun. 2020. Optimal parallel algorithms in the binary-forking model. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 89–102.
[17] Robert D. Blumofe and Charles E. Leiserson. 1993. Space-efficient scheduling of multithreaded computations. In *ACM Symposium on Theory of Computing (STOC)*. ACM, 362–371.
[18] Boost geometry 2024. Boost Geometry Library. https://www.boost.org/library/latest/geometry/.
[19] Surajit Chaudhuri and Vivek R. Narasayya. 1998. AutoAdmin 'What-if' Index Analysis Utility. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 367–378.
[20] Chicago Data Portal 2024. Chicago Crimes: 2001 to Present. https://data.cityofchicago.org/Public-Safety/Crimes-2001-to-Present/ijzp-q8t2/about_data.
[21] Demographic and Economic Data 2022. TIGER with Selected Demographic and Economic Data. https://www.census.gov/geographies/mapping-files/time-series/geo/tiger-data.html.
[22] Laxman Dhulipala, Guy E. Blelloch, Yan Gu, and Yihan Sun. 2022. PaC-trees: Supporting Parallel and Compressed Purely-Functional Collections. In *ACM Conference on Programming Language Design and Implementation (PLDI)*.
[23] Laxman Dhulipala, Guy E Blelloch, and Julian Shun. 2019. Low-latency graph streaming using compressed purely-functional trees. In *ACM Conference on Programming Language Design and Implementation (PLDI)*. 918–934.
[24] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL server's memory-optimized OLTP engine. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 1243–1254.
[25] James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. 1989. Making data structures persistent. *J. Computer and System Sciences* 38, 1 (1989), 86–124.
[26] Ahmed Eldawy, Vagelis Hristidis, Saheli Ghosh, Majid Saeedan, Akil Sevim, A. B. Siddique, Samriddhi Singla, Ganesh Sivaram, Tin Vu, and Yaming Zhang. 2021. Beast: Scalable Exploratory Analytics on Spatio-temporal Data. In *ACM International Conference on Information and Knowledge Management*. 3796–3807.
[27] Ahmed Eldawy and Mohamed F. Mokbel. 2015. SpatialHadoop: A MapReduce Framework for Spatial Data. In *IEEE International Conference on Data Engineering (ICDE)*. 1352–1363.
[28] Raphael A Finkel and Jon Louis Bentley. 1974. Quad trees a data structure for retrieval on composite keys. *Acta informatica* 4 (1974), 1–9.
[29] Peter Frühwirt, Marcus Huber, Martin Mulazzani, and Edgar R Weippl. 2010. InnoDB database forensics. In *2010 24th IEEE International Conference on Advanced Information Networking and Applications*. IEEE, 1028–1036.
[30] Junhao Gan and Yufei Tao. 2017. On the hardness and approximation of Euclidean DBSCAN. *ACM Transactions on Database Systems (TODS)* 42, 3 (2017), 1–45.
[31] Yan Gu, Zachary Napier, and Yihan Sun. 2022. Analysis of Work-Stealing and Parallel Cache Complexity. In *SIAM Symposium on Algorithmic Principles of Computer Systems (APOCS)*. SIAM, 46–60.
[32] Antonin Guttman. 1984. R-trees: A dynamic index structure for spatial searching. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 47–57.
[33] Edwin H. Jacox and Hanan Samet. 2007. Spatial join techniques. *ACM Trans. Database Syst.* (2007).
[34] Ibrahim Kamel and Christos Faloutsos. 1992. Parallel R-trees. *ACM SIGMOD International Conference on Management of Data (SIGMOD)* 21, 2 (1992), 195–204.
[35] Curtis P. Kolovson and Michael Stonebraker. 1991. Segment Indexes: Dynamic Indexing Techniques for Multi-Dimensional Interval Data. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*. ACM Press, 138–147.
[36] Per-Åke Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M Patel, and Mike Zwilling. 2011. High-performance concurrency control mechanisms for main-memory databases. *Proceedings of the VLDB Endowment (PVLDB)* 5, 4 (2011), 298–309.
[37] Scott T. Leutenegger, Jeffrey Edgington, and Mario Alberto López. 1997. STR: A Simple and Efficient Algorithm for R-Tree Packing. In *IEEE International Conference on Data Engineering (ICDE)*. 497–506.
[38] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-tree for New Hardware Platforms. In *IEEE International Conference on Data Engineering (ICDE)*. IEEE Computer Society, 302–313.
[39] libspatialindex 2024. libspatialindex. https://libspatialindex.org/en/latest/.
[40] Los Angeles Open Data 2024. Traffic Collision Data from 2010 to Present. https://data.lacity.org/Public-Safety/Traffic-Collision-Data-from-2010-to-Present/d5tf-ez2w/about_data.
[41] Ziyang Men, Zheqi Shen, Yan Gu, and Yihan Sun. 2025. Parallel kd-tree with Batch Updates. *ACM SIGMOD International Conference on Management of Data (SIGMOD)* 3, 1 (2025), 62:1–62:26.
[42] MicrosoftSQL 2025. Microsoft SQL server. https://www.microsoft.com/en-us/sql-server/.
[43] mongodb 2025. MongoDB. https://www.mongodb.com/.
[44] Guy M Morton. 1966. A computer oriented geodetic data base and a new technique in file sequencing. (1966).
[45] Mario A. Nascimento and Jefferson R. O. Silva. 1998. Towards Historical R-trees. In *ACM symposium on Applied Computing (SAC)*. ACM, 235–240.
[46] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Inf.* 33, 4 (1996).
[47] Open Street Map Data 2024. Source of OSM data. https://download.geofabrik.de/.
[48] oracle 2025. Oracle Database. https://www.oracle.com/database/.
[49] PostGIS 2025. PostGIS. https://postgis.net/.
[50] Redis 2024. Redis: The Real-Time Data Platform. https://redis.io/.
[51] Simonas Saltenis, Christian S. Jensen, Scott T. Leutenegger, and Mario Alberto López. 2000. Indexing the Positions of Continuously Moving Objects. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*. ACM, 331–342.
[52] spatialhadoop 2025. SpatialHadoop. https://spatialhadoop.cs.umn.edu/.
[53] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. 2007. The End of an Architectural Era (It's Time for a Complete Rewrite). In *Proceedings of the VLDB Endowment (PVLDB)*. ACM, 1150–1160.
[54] Yihan Sun and Guy Blelloch. 2019. Implementing Parallel and Concurrent Tree Structures. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*. 447–450.
[55] Yihan Sun, Guy E Blelloch, Wan Shen Lim, and Andrew Pavlo. 2019. On supporting efficient snapshot isolation for hybrid workloads with multi-versioned indexes. *Proceedings of the VLDB Endowment (PVLDB)* 13, 2 (2019), 211–225.
[56] Yihan Sun, Daniel Ferizovic, and Guy E Blelloch. 2018. PAM: Parallel Augmented Maps. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*.
[57] Yufei Tao and Dimitris Papadias. 2001. MV3R-Tree: A Spatio-Temporal Access Method for Timestamp and Interval Queries. In *Proceedings of the VLDB Endowment (PVLDB)*. Morgan Kaufmann, 431–440.
[58] Yufei Tao, Dimitris Papadias, and Jimeng Sun. 2003. The TPR*-Tree: An Optimized Spatio-Temporal Access Method for Predictive Queries. In *Proceedings of the VLDB Endowment (PVLDB)*. 790–801.
[59] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. 2017. An Empirical Evaluation of In-Memory Multi-Version Concurrency Control. *Proceedings of the VLDB Endowment (PVLDB)* 10 (2017), 781–792. Issue 7.