Journal of Big Data

RESEARCH

Open Access

Comparison of LSM indexing techniques for storing spatial data

Qizhong Mao[1*†], Mohiuddin Abdul Qader[1†] and Vagelis Hristidis[1]

†Qizhong Mao and Mohiuddin Abdul Qader contributed equally to this work

*Correspondence: qmao002@ucr.edu

[1] Department of Computer Science and Engineering, University of California, Riverside, USA

Abstract

In the pre-big data era, many traditional databases supported spatial queries via spatial indexes. However, modern applications are seeing a rapid increase of the volume and ingestion rate of spatial data. Log-structured Merge (LSM) tree is used by many big data systems as their storage structure in order to support write-intensive large-volume workloads, which are usually only optimized for single-dimensional data. Research has studied how spatial indexes can be supported on LSM systems, but focused mainly on the local index organization, that is, how data is organized inside a single LSM component. This paper studies various aspects of LSM spatial indexing, including spatial merge policies, which determine when and how spatial components are merged. Three stack-based and one leveled merge policies have been studied, which have been implemented on a common big data system Apache AsterixDB. The write and read performance on various workloads is evaluated, and our findings and recommendations are discussed. A key finding is that Leveled policies underperform other stack-based merge policies for most types of spatial workloads.

Keywords: Spatial index, LSM, Merge policy, Stack-based, Leveled, R-tree, Partition

Introduction

Due to the increasing need of (mobile) applications such as navigation systems, location-based review systems, and geo-tagged social media, the volume and ingestion rate of spatial data are increasing rapidly. Database systems have been moving to log-structured merge (LSM) tree [1] storage architectures to facilitate high write throughput. Such systems include Apache AsterixDB [2], Cassandra [3], and HBase [4], Google Bigtable [5], and LevelDB [6], Facebook RocksDB [7], and ScyllaDB [8]. LSM systems provide superior write performance than many traditional relational databases, MySQL with InnoDB for example [7]. However, most of these systems do not have native support of spatial queries, which often rely on spatial indexes.

In most applications, a spatial index cannot live alone and must be created as a secondary index that is dependent on a primary index to query any non-spatial attributes. Most LSM systems do not have the direct support of the general secondary index, as a result, they are unable to support spatial index. In AsterixDB, LSM-fication is a generic framework to convert a class of indexes to LSM secondary indexes [9]. Using this framework, two options to index spatial data are available. The first option is a B^+-tree-based solution that indexes single-dimensional data projected from multidimensional spatial data through linearization. Note that this option also applies to systems (e.g, LevelDB and RocksDB) that use Sorted-String-Table (SSTable) and binary search methods to support range queries. The second option is a native spatial index, for example, R-tree, as a local index. To the best of our knowledge, AsterixDB is the only LSM storage engine with native support of LSM R-tree index; all other LSM-based systems only support B^+-tree index at most. Based on the results from [10, 11], the R-tree-based solution is the general preferable option for LSM spatial index in most scenarios, hence in this paper, LSM R-tree indexes are focused only.

In addition to the organization of the local index discussed above, which determines how data is organized in a single LSM component (file), another key design choice for spatial LSM indexes is the merge policy, which determines when and how components are merged. The two main merge paradigms considered are stack-based and leveled. In stack-based policies, components are organized as a stack, where the most recent components are higher in the stack. Leveled policies use (almost) fixed-size components, with newer components on higher levels; lower levels have more components per level. Stack-based LSM trees usually have better write performance and good read performance. Leveled LSM tree is the most popular paradigm in the industry with very good read performance, but higher write amplification in general [12].

The typical query for spatial indexing is a region query, where the region is typically expressed as a Minimum Bounding Rectangle (*MBR*). For each component, its MBR is maintained, so it is easy to filter components based on the query MBR. This filtering is generally not effective in stack-based policies, as most components have very large MBRs, comparable to the whole space in many applications. On the other hand, this filtering can be more effective for Leveled policy, because the components on the same level are mostly disjoint in key ranges. In the case of R-tree indexing, this means that the components at the same level have non-overlapping MBRs, or possibly limited overall, depending on the partitioning algorithm employed.

To achieve minimal spatial overlap in Leveled policies, spatial partitioning algorithms, specifically Sort-Tile-Recursive (STR) [13] and R^*-Grove [14], are employed. There are several subtle implementation decisions that significantly affect the merge performance. It is found that a critical one is the choice of *comparator*, which compares two spatial records, because different comparator performs differently in high and low selectivity queries; certain combinations of comparator and partitioning algorithm in Leveled policy can effectively create disk components of disjoint MBRs, which significantly improves filtering efficiency.

A key contribution of the paper is that several LSM spatial indexing algorithms are implemented on a common database system, AsterixDB, and compared them for write and read performance using two spatial workloads. A key conclusion is that stack-based policies generally perform better with low write and read cost. Although Leveled policy had very high write amplification, certain configurations could achieve comparable write throughput to stack-based policies. Its read performance was also very competitive in low selectivity queries.

In summary, this work makes the following contributions:

1. How an LSM architecture can be extended to support secondary spatial indexes is studied ("Spatial LSM index based on R-tree"). Several design decisions and architectures are considered.
2. A number of optimized partitioning algorithms for Leveled LSM R-tree index are examined, which minimize the overlap among MBRs while also minimizing the I/O cost ("Partitioning in leveled LSM R-tree").
3. All compared LSM spatial indexing policies on AsterixDB are implemented, and the source code is publicly available at [15].
4. All LSM spatial indexing algorithms using a real-world dataset and a synthetic dataset are experimentally compared ("Experimental evaluation").
5. Our observations and recommendations are discussed, which challenge the current popularity of Leveled policies ("Discussion").

Background

"LSM tree" discusses the fundamentals of an LSM tree and how data is maintained. "LSM architectures and merge policies" discusses two LSM architectures, stack-based and leveled, and several state-of-the-art merge policies compared and evaluated for each architecture.

LSM tree

An LSM tree [1] generally consists of two layers, one layer in memory which contains one active *memory component* (a.k.a MemTable), and one layer on disk where data is organized into one or multiple *sorted runs* [16]. Every sorted run contains records sequentially ordered by the indexed key. Depending on the LSM tree architecture (discussed in "LSM architectures and merge policies"), a sorted run can have one or multiple immutable *disk component*s. Components are typically implemented using a tree structure, such as B^+-tree. A tree structure usually partitions records into blocks or pages as nodes (some may group multiple blocks or pages into frames as nodes). This partition is usually referred to as *local partition* within a physical file. On the other hand, a sorted run may be a virtual file that is partitioned into multiple physical files, which is referred to as *global partition*. Local partition is often bound with the data structure used in physical files, where global partition is associated with the LSM tree architecture. Therefore, in this paper, only the global partition is primarily focused on.

All records inserted or updated are batched into the memory component. When the memory component reaches its capacity, it is scheduled to be *flush*ed to disk, creating a disk component, as shown in Fig. 1. A flush operation sorts the records in the memory component P_M, then bulk-writes the sorted records to a disk component P_2. A *comparator*, which compares two keys, is used to sort records in the memory component. A key

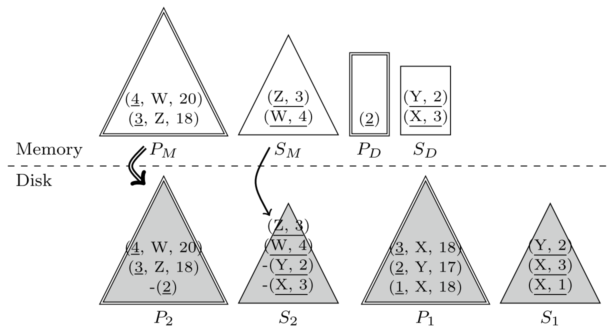


Fig. 1 LSM flush operation. Primary (double line) and secondary (single line) memory components are flushed independently to the top of disk components of the corresponding index. Delete table is flushed together with the corresponding memory component (P_M and P_D, S_M and S_D), creating antimatter (a.k.a tombstone) records (marked with -) in the flushed component. Index keys are underlined. The primary index *P*'s schema is (*CarID*, *OwnerID*, *ManufactureYear*). A secondary index *S* is built on *OwnerID*

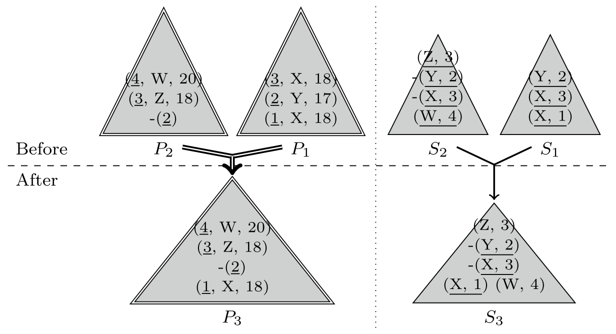


Fig. 2 LSM merge operation. Primary index and secondary index are merged independently. Anti-matter records (marked with -) can be completely deleted if the oldest disk component (e.g. P_1 or S_1) is involved in the merge

is inserted to a separate in-memory delete table P_D when a record is deleted (e.g. 2). P_D is flushed together with the P_M, adding *anti-matter* (a.k.a tomestone) records (e.g. -(2)) to P_2.

Reads become slower as the number of disk components increases. To improve the read performance, disk components are merged based on a *merge policy* (a.k.a. compaction strategy). A merge operation scans all records from all merging components and creates a sorted stream using a priority queue and the same comparator, then bulk-writes the unique records into new disk component(s), as illustrated in Fig. 2. Obsolete (old version) records are discarded during a merge, leaving only the newest version. For example, (3, X, 18) from P_1 gets removed because P_2 has a newer version (3, Z, 18). An antimatter record will overwrite any old versions of the same record (e.g record with key 2), but will be overwritten by a new version of the same valid record (from a later insertion). Anti-matter records can be deleted when the oldest sorted run is involved in a merge. *Write amplification* is a common measurement of the write cost in an LSM system. A read query first checks the metadata of all components, and adds components whose key range contains the searched key to an ordered list of *operational components*.

All operational components may contain records to answer the query, thus, the number of operational components is usually used to compute the *read amplification*, which measures the read cost in the worst case.[1]

LSM architectures and merge policies

Stack-based LSM tree

In a stack-based LSM tree, every single disk component is a sorted run (thus component and sorted run are interchangeable), where disk components are ordered by the time created from flushes or merges. Stack-based merge policies generally merge only consecutive disk components and create only one single disk component per merge.

Most stack-based merge policies make merge decisions based on certain size ratio conditions, where every single merge involves similar sized disk components. Such merge policies are often referred to as *tiering style*. The term tiering came from the SizeTiered policy in Cassandra (described later in this section), while the term stack-based came from Bigtable [17, 18]. Tiering style merge policies are a subset of stack-based merge policies. A key difference is that component sizes in tiering style merge policies are non-decreasing with respect to their time of creation, such that older components are usually no smaller than any newer components, while such restriction does not hold for the general stack-based merge policies, where there is no relation between component sizes and freshness. Certain stack-based policies choose to restrict the total number of disk components to a constant number which limits the worst case read amplification. These policies are called *bounded-depth* policies [18].

In this research, the following three stack-based merge policies are selected for evaluation:

- **Binomial** policy was originally proposed by Mathieu et al. [17], then formally defined and evaluated in [12, 18]. The name Binomial came from the fact that it uses a Binary Search Tree to make merge decisions. It is a bounded-depth policy that maintains an optimal write cost with an upper bound of worst case read amplification by only one parameter k, which restricts the maximum number of disk components. Compared to the other online merge policies, whose merge schedules are based on heuristic information such as component sizes, Binomial policy is an offline policy whose merge schedule is pre-determined only on the number of flushes. It was originally designed for append-only workload, but can be adjusted for workloads with updates or deletions as well.
- **Tiered** (a.k.a SizeTiered) policy is the default policy in Cassandra [19], and had been adopted as Universal Compaction [20] in RocksDB. It groups disk components into tiers. Every tier has B disk components. Whenever a tier has B disk components, all the B disk components are merged into a new component of B times larger into the top of the next tier. B is also called size ratio or fanout factor. The implementation of Tiered policy varies in different systems. For exam-

[1] Some operational components may be skipped by filters such as Bloom filter, reducing the actual read amplification.

ple, besides selecting similar sized components to merge, the Universal compaction in RocksDB can also select components that have more overlapping keys to reduce space amplification, or simply merge several components to enforce the total number of components to the number specified by *level0_file_num_compaction_trigger* [20], if the other two options cannot be performed. In this paper, the Tiered policy is implemented in a similar way to Cassandra, which only selects components based on size ratio, thus ignoring components' contents.

- **Concurrent** policy was recently added to AsterixDB to replace Prefix policy as its new default policy [21]. Unlike Prefix policy, which tends to merge similar sized components and excludes components whose sizes are larger than a user defined threshold, Concurrent policy is bounded-depth by a parameter k. The disk components to be merged are determined by a minimum length C, a maximum length D and a size ratio λ. Starting from the newest disk component, the policy considers any longest sequence with disk components $\{D_i, D_{i-1}, \ldots, D_1\}$ where $C + 1 \leq i \leq D$, and merges them into a single disk component if $|D_i| \leq \lambda \sum_{j=1}^{i-1} |D_j|$, where $|D_j|$ is the size of the disk component D_j.

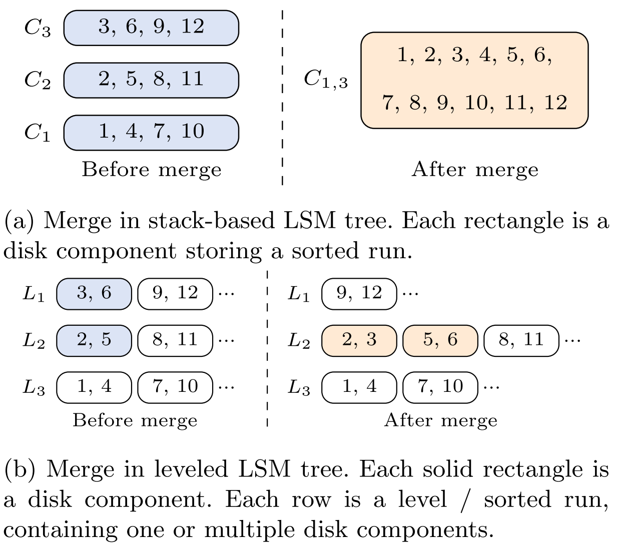
Leveled LSM tree

In a Leveled LSM tree [6, 7] every level is a sorted run which is partitioned into multiple (typically) disjoint disk components of the same size [22]. The number of disk components in level $i \geq 2$ is B times more than the number in level $i - 1$. There may also be a special level 0 which contains B_0 disk components as a buffer, which holds flushed disk components as multiple sorted runs. When a level reaches its capacity (B_0 or B^i), a disk component is selected and merged with all overlapping disk components in the next level, creating one or multiple new disk components in the next level. While the oldest disk component in level 0 must be selected, any disk component in other levels can be selected. A point query only needs to check all disk components in level 0, and at most 1 disk component in every level $i \geq 1$. A range query may just need to check a few components in each level, reducing the total size to be checked.

Comparing different merge policies

The major differences between stack-based and leveled LSM trees in terms of merge operations are illustrated in Fig. 3. All sorted runs (disk components/levels) are ordered from newer to older in top-down direction, where blue and orange rectangles represent input and output components of a merge, respectively. Three consecutive components are merged into a single component in a stack-based LSM tree, where in a leveled LSM tree, one component ([3, 6]) from one level (L_1) is merged with the only overlapping component ([2, 5]) in the next level (L_2) and the two output components are placed in the next level ($L2$).

To better illustrate the difference among the four compared merge policies, their sorted run sizes after some number of flushes are listed in Table 1. Component sizes in Tiered and Concurrent are always non-decreasing. For Binomial, it is possible that some



(a) Merge in stack-based LSM tree. Each rectangle is a disk component storing a sorted run.

(b) Merge in leveled LSM tree. Each solid rectangle is a disk component. Each row is a level / sorted run, containing one or multiple disk components.

Fig. 3 Examples of merges in stack-based and leveled LSM trees. Input and output components in a merge are marked in blue and orange, respectively. Keys in a component are represented by the numbers inside the rectangle

Table 1 Sorted run sizes of the four compared merge policies, where newer sorted runs are on the left

Merge policy	20 Flushes	40 Flushes	60 Flushes	80 Flushes	100 Flushes	120 Flushes
Binomial ($k = 4$)	1, 4, 15	2, 3, 20, 15	10, 50	10, 20, 50	15, 35, 50	1, 3, 10, 106
Tiered ($B = 4$)	4, 16	4, 4, 16, 16	4, 4, 4, 16, 16, 16	16, 64	4, 16, 16, 64	4, 4, 16, 16, 16, 64
Concurrent (default)	3, 17	1, 1, 3, 35	1, 59	3, 77	1, 1, 3, 95	1, 119
Leveled ($B_0 = 2, B = 4$)	1, 1, 4, 14	1, 1, 4, 16, 18	1, 1, 4, 16, 38	1, 1, 4, 16, 58	1, 1, 4, 16, 64, 14	1, 1, 4, 16, 64, 34

Each number is the size of a sorted run with respect to the MemTable size. Tiered, Concurrent and Leveled always have sorted runs in non-decreasing order. The first two sorted runs in Leveled policy are two disk components in level 0, the other numbers are the number of disk components of size 1 in the corresponding levels. Default parameters for Concurrent: $k = 30, C = 3, D = 10, \lambda = 1.2$.

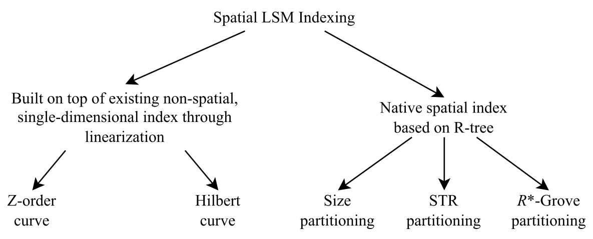

Spatial LSM Indexing
Built on top of existing non-spatial, single-dimensional index through linearization
Native spatial index based on R-tree
Z-order curve
Hilbert curve
Size partitioning
STR partitioning
R*-Grove partitioning

Fig. 4 Types of LSM-based spatial indexes

newer sorted runs are larger. For example, after 40 flushes, the third sorted run has size 20 while the fourth sorted run has size 15. Also the number of sorted runs in Binomial never exceeds $k = 4$.

LSM secondary spatial index

This section first covers how LSM secondary indexes are maintained ("LSM Secondary Index"), which affects the trade-off between write and read performance. Then it discusses two approaches to index the spatial data, which are special type of secondary data, on LSM systems: a B^{+}-tree-based solution is discussed in "Spatial LSM Index based on B+-tree", an R-tree-based solution is discussed in "Spatial LSM Index based on R-tree". How different merge policies affect the spatial index performance and present a partitioning algorithm for the Leveled policy is then discussed. Fig. 4 presents all the spatial LSM indexing approaches discussed in this paper.

LSM secondary index

Before talking about the spatial index, how LSM secondary indexes are constructed and maintained must be explained. An LSM secondary index has almost identical architecture to the primary index, except it is sorted by a composite key $\langle SK, PK \rangle$, where SK is the secondary key, and PK is the primary key. Records are first ordered by SK then by PK in disk components. When a record gets updated or deleted in the primary index, the current composite key in a secondary index may become invalid as SK is no longer valid for PK. During record insertion, an *eager* strategy uses the PK ($\underline{2}$ and $\underline{3}$ in Fig. 1) to find the old value of SK (Y for $\underline{2}$ and X for $\underline{3}$), and then inserts an antimatter record with the old SK and PK (two entries in S_D) to all secondary indexes, so any read on a secondary index will only return valid records thus the primary index does not need to be checked. A *lazy* strategy does not update secondary indexes during records insertion but needs an extra step, querying the primary index, to verify the returned records. Writes are usually slower in the eager strategy due to the checking on all secondary indexes but reads can be faster. On the other hand, the lazy strategy provides faster writes, but reads are slower due to the extra validation in the query time. Detailed discussion about these secondary indexing strategies can be found in [9, 23, 24].

An LSM secondary index can have its own memory component budget, and flushes and merges are triggered independently of the primary index. Or it can share a global budget with the primary index. In this design, the primary index and all secondary indexes are always flushed together, but they may use different merge policies. Merges may not be triggered at the same time, although certain merge policies (CorrelatedPrefix policy in AsterixDB) can enforce the merges for all indexes at the same time. Read queries on an LSM secondary index are very similar to the merge operation.

Spatial LSM index based on B^{+}-tree

Most of the works on LSM trees are optimized for single-dimensional data. Unlike single-dimensional data, there is usually no clear definition of how to order multidimensional spatial data. The most common approach is to project multidimensional data to single-dimensional data to be indexed by a B^{+}-tree. The projection is made through *linearization*. One of the most common linearization methods is the space-filling curve. The two most well-known space-filling curves are Z-order curve and Hilbert curve. The BuildIndexes function of Algorithm 1 presents the pseudocode of building a spatial index on B^{+}-tree via space-filling curve. Both GeoMesa [25] and DataStax Cassandra

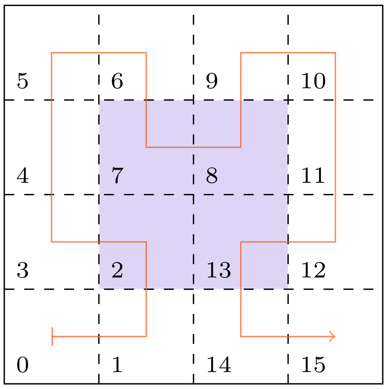


Fig. 5 Example of Hilbert curve and spatial intersection query

[26] support this type of spatial index using GeoHash [25, 27], which is based on Z-order curve. More details are discussed in "Related work".

Algorithm 1 Spatial Index based on B^+-tree

```
1:  # The space-filling curve (SFC) value of point $\vec{a}$
2:  function SFCPoint($\vec{a}$)
3:
4:  function BuildIndexes
5:      while $not$ end do
6:          $pk \leftarrow$ NextRecord.PrimaryKey            ▷ Primary key
7:          $\vec{v} \leftarrow$ NextRecord.SpatialKey            ▷ Secondary key
8:          $o \leftarrow$ NextRecord.Others           ▷ Any other attributes
9:          WritePrimaryIndex($pk$, $\vec{v}$, $o$)
10:         WriteBTreeIndex(SFCPoint($\vec{v}$), $pk$)
11:     end while
12: end function
13:
14: # All SFC values of a rectangle as $\vec{a}$ to $\vec{b}$
15: function SFCRect($\vec{a}$, $\vec{b}$)
16: # All primary keys whose spatial key's SFC value is $v$
17: function PrimaryKeys($v$)
18: # The spatial key (point) of a primary key $pk$
19: function SpatialKey($pk$)
20:
21: # $\vec{a}$ and $\vec{b}$ are the lower (left) and upper (right) points of the
    searching rectangle, respectively.
22: function SpatialSearch($\vec{a}$, $\vec{b}$)
23:     $S \leftarrow ()$              ▷ (Hash) set for unique primary keys
24:     for $v_c \in$ SFCRect($\vec{a}$, $\vec{b}$) do
25:         for $pk \in$ PrimaryKeys($v_c$) do          ▷ $B^+$-tree search
26:             $S$.Add($pk$)
27:         end for
28:     end for
29:     $R \leftarrow []$                         ▷ List of matching records
30:     for $pk \in S$ do
31:         $\vec{v} \leftarrow$ SpatialKey($pk$)              ▷ Primary index search
32:         if $\vec{v}$ WithinRect($\vec{a}, \vec{b}$) then
33:             $R$.Add(Record($pk$))
34:         end if
35:     end for
36:     return $R$
37: end function
```

A space-filling curve partitions space into cells of the same size and uses fixed-length bit strings (usually 32/64-bit numbers) to represent each cell. A toy example of a Hilbert curve with 4 bits is shown in Fig. 5. For point-type data, the value of the cell in which a point resides will be saved as the secondary key *SK*, and a B^{+}-tree is built on these cell values. Spatial queries may be handled in two ways to obtain the cells to be scanned. The first method is to find the cell values for all corners of the query MBR (the light purple region) and scans all cells between the smallest cell (2) and the largest cell (13). This method utilizes sequential disk I/O but may waste lots of resources checking records not in the contained cells (e.g., 3–6 and 9–12). It is generally preferred when the difference between the two values is small. Another method is to identify the exact cells in which the query MBR covers (2, 7, 8, and 13), then scans every covered cell. This method minimizes the disk I/O, but more random I/Os are involved. It can be used if the minimum and maximum cell values are far apart or very few cells are covered. Both methods get the records whose *SK* fall into the query cells, but every record must be further checked using its spatial attribute. As shown in the function SpatialSearch of Algorithm 1, the secondary spatial index is first searched to get all the primary keys whose spatial keys match any of the space-filling curve values from the searching MBR. This process may be implemented as a range query or multiple point queries. Next, the spatial attribute of each unique primary key must be obtained from the primary index. Then, the spatial attribute will be verified with the searching MBR to determine if the record shall be returned.

Spatial index with linearized data on B^{+}-tree can be very efficient due to the superior random and sequential read performance of B^{+}-tree. It is also relatively easy for an existing database system to support spatial index with some extended framework (GeoMesa) Despite these advantages, these methods have some common drawbacks. The major issue is that this type of index requires some prior knowledge about the space, such as the minimum and maximum values of each dimension, and object distribution, to decide the number of cells to use. Storing cell values costs extra disk space and I/O during index writes and reads. Spatial objects in some cells may be very dense, making scans in these cells relatively slow.

Spatial LSM index based on *R*-tree

Spatially close objects may not have close cell values, as shown in Fig. 5. A natural way is to place nearby records into the same groups. *R*-tree [28] and R^{*}-tree [29] are widely used as local indexes for spatial data, which partition records into disk blocks based on their spatial locations (in this paper, *R*-tree and R^{*}-tree are used interchangeably). The *R*-tree has a similar implementation to B^{+}-tree, except it partitions leaf nodes and creates internal nodes by MBRs. Spatial queries may need to traverse multiple paths to leaf nodes to find records. To bulk-write an *R*-tree, records are sorted by a comparator, then packed into multiple partitions as leaf nodes and create internal nodes accordingly in a bottom-up fashion. Common comparators used in *R*-tree include space-filling curve comparators (**Hilbert curve** or Z-order curve), and **simple** bitwise comparator (Algorithm 2). Because only the relative order of two records is needed, space filling curves values are only computed during run-time, and do not need to be stored together with the records, which saves disk space and reduces disk I/O. The simple comparator compares two points by each dimension, which is essentially the Nearest-X algorithm [13, 30]. Note that it is a generalized version of the comparator used for single-dimensional data.

Algorithm 2 Comparators

```
    # $\vec{a}$: A point type spatial object represented by a vector.
2:  # $|\vec{a}|$: The number of dimensions of $\vec{a}$

4:  function SFCCompare($\vec{a}$, $\vec{b}$)                ▷ $|\vec{a}| = |\vec{b}|$
        $v_a \leftarrow$ SFCPoint($\vec{a}$)                 ▷ Algorithm 1
6:      $v_b \leftarrow$ SFCPoint($\vec{b}$)                 ▷ Algorithm 1
        if $v_a < v_b$ then
8:          return -1                                         ▷ $\vec{a}$ is smaller
        else if $v_a > v_b$ then
10:         return 1                                          ▷ $\vec{b}$ is smaller
        else
12:         return 0                                          ▷ $\vec{a}$ and $\vec{b}$ are equal
        end if
14: end function

16: function SimpleCompare($\vec{a}$, $\vec{b}$)             ▷ $|\vec{a}| = |\vec{b}|$
        for $i \leftarrow 1, \ldots, |\vec{a}|$ do           ▷ Compare each dimension
18:         if $\vec{a}[i] < \vec{b}[i]$ then
                return -1                                     ▷ $\vec{a}$ is smaller
20:         else if $\vec{a}[i] > \vec{b}[i]$ then
                return 1                                      ▷ $\vec{b}$ is smaller
22:         else
                continue                                      ▷ Check the next dimension
24:         end if
        end for
26:     return 0                                              ▷ $\vec{a}$ and $\vec{b}$ are equal
    end function
```

In an LSM *R*-tree index, *SK* is the spatial location of every record, typically as an array of numbers. The same records are compared multiple times during flushes, merges, and queries. With a space-filling curve comparator, linearized values of records must be re-computed every time, potentially adding delays to those operations. In most cases, *R*-tree (or R^*-tree) is the preferred option for spatial index [10, 11]; hence in this paper, the LSM *R*-tree designs is only focused on.

A spatial query first determines the list of operational components by checking each component's MBR, represented by the minimum key (bottom left point) $\mathbf{K}_{min}$ and the maximum key (top right point) $\mathbf{K}_{max}$, where $\mathbf{K}$ represents an array. Given two components $C : \langle \mathbf{K}_{min}, \mathbf{K}_{max} \rangle$ and $C' : \langle \mathbf{K}'_{min}, \mathbf{K}'_{max} \rangle$ and the number of dimensions $D \geq 1$, the two components are overlapping if and only if (1) is satisfied, or disjoint otherwise.

$$\forall d \in [1, D] : \mathbf{K}_{min}[d] \leq \mathbf{K}'_{max}[d] \wedge \mathbf{K}'_{min}[d] \leq \mathbf{K}_{max}[d]. \tag{1}$$

Then, a spatial search can scan all operational components and return the results directly, as shown in Algorithm 3. Depending on the actual query, the primary index may not be involved in the spatial search.

Algorithm 3 Spatial Index based on R-tree

```
 1: function BuildIndexes
 2:     while not end do
 3:         $pk \leftarrow$ NextRecord.PrimaryKey                 ▷ Primary key
 4:         $\vec{v} \leftarrow$ NextRecord.SpatialKey               ▷ Secondary key
 5:         $o \leftarrow$ NextRecord.Others                ▷ Any other attributes
 6:         WritePrimaryIndex($pk$, $\vec{v}$, $o$)
 7:         WriteRTreeIndex($\vec{v}$, $pk$)
 8:     end while
 9: end function
10:
11: function SpatialSearch($\vec{a}$, $\vec{b}$)
12:     $R \leftarrow []$                              ▷ List of matching records
13:     for $\langle \vec{v}, pk \rangle \in$ Rect($\vec{a}$, $\vec{b}$) do                 ▷ $R$-tree search
14:         $R$.Add(Record($pk$))
15:     end for
16:     return $R$
17: end function
```

As described in "Stack-based LSM Tree", stack-based merge policies are often unaware of disk components' contents like key boundaries, which merges are scheduled in the same way regardless of the type or dimensions of the data. Disk components have a high chance to have intersected MBRs with each other, making MBR based filtering at component level less important for stack-based policies. Also, *R*-tree employs MBR-based filtering on the disk block level internally; only a small portion of disk components is read even if the component size is large. Although a spatial query usually needs to scan all disk components, the read amplification is not high, due to the low average number of disk blocks scanned per component. To the best of our knowledge, AsterixDB is the only system that uses stack-based LSM *R*-tree indexes.

Stack-based LSM *R*-tree indexes mostly rely on the local index of disk components for spatial queries, which has little room to improve in the policies themselves. However, it is very different for the Leveled LSM *R*-tree index. A Leveled LSM *R*-tree index may have thousands of disk components. A spatial query can potentially check all disk components in the worst case, which leads to very high read amplification and low locality. Two key design decisions are (a) how to partition records into components during merges and (b) what comparator to use to order records inside a component to allow faster merges. They will be discussed in the next section in detail. To the best of our knowledge, no current system is using leveled LSM *R*-tree indexes, which is surprising given the popularity of leveled merge policies. All the discussed policies on AsterixDB are implemented for the experiments.

Partitioning in leveled LSM *R*-tree

A partitioning algorithm is necessary to split records into different disk components, which affects the performance of write and read operations of a leveled LSM tree. It must be capable of distributing records into a fixed number of partitions such that the number of records in all partitions are roughly the same. This section discusses three partitioning algorithms, size, STR and R^*-Grove, along with two comparators, the Hilbert curve comparator, and the simple comparator.

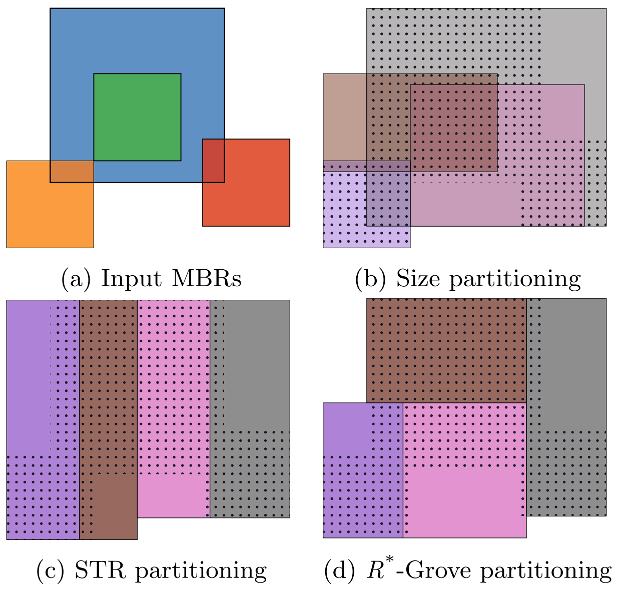


Fig. 6 Examples of three partitioning algorithms from the same input. Points are uniformly distributed in each of the four input MBRs and are marked with dots in the three partitioned sub-figures

Size partitioning Size partitioning is the default partitioning algorithm used in leveled LSM-trees. It simply distributes sorted records into multiple disk components such that all disk components have roughly the same size. A priority queue takes streams of sorted records from each merging component as inputs, and outputs a stream of sorted records from all merging components, similar to the sort-merge join algorithm. Because records are already sorted in each component, storing them in memory for sorting is not needed. Size partitioning only fetches one disk block from each merging disk component at a time, so the memory requirement is minimal. The order of the records depends on the comparator being used. By default, AsterixDB sorts spatial records by a Hilbert curve comparator for 2-D point data and Z-order curve comparator for the other types of spatial data. The two space filling curve based comparators cannot guarantee spatially disjoint disk components, as shown in Fig. 6b as the partitioning result from Fig. 6a. On the other hand, if size partitioning is coupled with the simple comparator, this combination can achieve a similar result as STR partitioning, which will be discussed in the next paragraph.

STR partitioning Sort-Tile-Recursive (STR) [13] was originally proposed to pack blocks for *R*-tree for point data. This partitioning algorithm is adopted in leveled LSM *R*-tree index. When disk components are merged, STR is applied to partition all merging records to multiple spatially disjoint groups and create a separate disk component for each group. That way, all disk components in one sorted run are disjoint, regardless of the comparator. For non-point data, STR is applied to the center points of spatial objects, but MBRs are computed from their actual MBRs. The comparator only affects the order of the records inside each component, but the components' MBRs remain the same. There are two major drawbacks of STR partitioning. The first is that STR requires storing all merging records in memory for sorting, leading to much higher CPU and memory usage, otherwise, external sorting is needed which incurs much higher disk I/O

Table 2 Run time of spatial partitioning algorithms

	Size	STR	R^*-Grove
Hilbert	$\mathcal{O}(n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log^2 n)$
Simple	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n \log^2 n)$

cost. Thus, it is generally slower than size partitioning. The second is that because STR gives higher weights on more significant dimensions, it tends to create narrow but tall rectangles (as shown in Fig. 6c from the same input), which may make read queries less efficient as a read query may need to check more disk components although only a small portion of each disk component is actually needed. This may be even more severe for higher dimensional data [14, 31].

Partitioning R^*-Grove [14, 31] partitioning is also ported, which aims to create square-like and balanced partitions for analytic frameworks like Apache Hadoop and Spark, into AsterixDB for our experiments. R^*-Grove partitions spatial records in three phases: a sampling phase which draws a random sample of the input records, a boundary computation phase which generates partition boundaries with desired level of load balance, and a final partitioning phase which puts every record into the corresponding partition. Like STR, comparator does not affect the partitioning but only affects the internal organization of disk components. As shown in Fig. 6d (from the same input), R^*-Grove tends to create more square-like MBRs so fewer disk components may be checked. However, it makes multiple passes to scan all records, and is computationally more expensive than STR, for which merges are usually slower.

Both STR and R^*-Grove face an issue of high memory usage, which limits the total size of components to be merged. A possible solution is to make two passes on all merging disk components. The first pass samples a small number of records from all merging disk components, then STR or R^*-Grove can be applied on the sampled records to obtain partition boundaries. The second pass scans all records and puts them into a corresponding partition whose MBR contains the record (or the center if it is not point type). This method only uses a small amount of memory, but significantly increases the number of disk I/Os during merge operations. The run time of placing n points into p partitions is summarized in Table 2. Size partitioning simply scans all points linearly regardless of the comparator. STR partitioning requires sorting the points by each dimension. For Hilbert curve comparator, sorting takes $\mathcal{O}(n \log n)$, thus the total run time is $\mathcal{O}(n \log n) + \mathcal{O}(n) = \mathcal{O}(n \log n)$. For Simple comparator, there is no need to sort again, thus the total run time is just $\mathcal{O}(n)$. The run time of R^*-Grove can be found in [31].

Experimental evaluation

Datasets and workloads

Two geo-location datasets of exactly 100,000,000 2-D points were used in all experiments. One is a real-world dataset randomly sampled from OpenStreetMap (*OSM* for short) [32, 33]; the other is a synthetic dataset which longitude and latitude values were uniform randomly generated. Points in the OSM dataset are highly clustered in urban areas all over the world, especially in the United States and western Europe (Fig. 7a). Points in the random dataset are uniformly distributed around the globe (Fig. 7b).

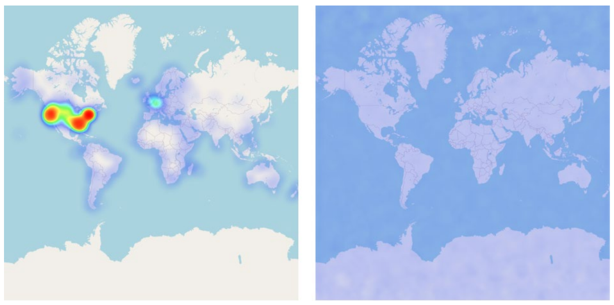


Fig. 7 Heatmap of the two datasets, coordinates range from $[-180°, -90°]$ to $[180°, 90°]$

For each dataset, a workload with interleaved reads and writes is generated as follows:

1 A *Load* phase of 50,000,000 records. Each record is associated with a unique ID in long type and a random string of 1000 bytes as a synthetic attribute (e.g., geolocation description). Points are stored as two double type numbers. Every record is exactly one kilobyte long.
2 An *Insert* phase containing 500,000 records.
3 A *Read* phase containing 10,000 spatial intersection queries. The query rectangle center is a point randomly picked from all previously inserted points. The rectangle size is determined by a random selectivity $10^{-\sigma}, \sigma \in \{3, 4, 5\}$, that the width and height are $360 \times 10^{-\sigma}$ and $180 \times 10^{-\sigma}$, respectively.

The *Load* phase was executed once in the beginning, then the *Insert* phase and *Read* phase were interleaved for 100 times that 100,000,000 total records were inserted (leading to 100 GB primary index and 2.4 GB LSM *R*-tree index), and 1,000,000 queries were executed. This interleaved workload guarantees the same data size in the corresponding insert phase and read phase in all experiments for fair comparisons.

Read queries were generated in a way that every query can return at least one record. Other selectivity values with $\sigma \in \{1, 2\}$ and $\sigma \in [6, 10]$ are also tested. It is observed the same results for $\sigma \in \{1, 2\}$ with $\sigma = 3$, and $\sigma \in [6, 10]$ with $\sigma = 5$, hence only results for $\sigma \in \{3, 5\}$ are reported ($\sigma = 4$ and $\sigma = 5$ are very similar). To avoid access to the primary index, COUNT(*) function is used so only the LSM *R*-tree index would be scanned. AsterixDB provides several built-in spatial functions, only "*spatial_intersect*" operates on the LSM *R*-tree index. Many other types of spatial query are usually based on pruning using MBR intersections, such as circle range, *k*NN and distance join, it is reasonable to focus on this type of rectangular intersection queries.

Experimental setup

Apache AsterixDB [34] is a full-function, open-source Big Data Management System (BDMS) on LSM storage. The primary index of a dataset is stored as LSM B^{+}-tree, the spatial index is stored as LSM *R*-tree. All secondary indexes and the primary index share a global memory budget; thus, they are always flushed together. AsterixDB uses the eager strategy to maintain secondary indexes. Spatial records are ordered by a Hilbert curve comparator or a Simple comparator. MBR of a disk component is computed from all records when it is created from a flush or a merge.

All experiments were performed on 5 AWS *m5.large* instances. Each instance has 2 vCPUs running on Intel Xeon Platinum 8175 M, 8 GB of memory, and 200 GB general purpose SSD (gp2). All 5 instances are located in the same zone "*us-west-2b*", connections within instances only used private IP to minimize network latency. AsterixDB was configured to use a single node in each server. Other configurations were set to the defaults. The average size of the flushed disk components in LSM *R*-tree index was around 2 MB.

Merge policy configurations

The following merge policy configurations were applied to the LSM *R*-tree index:

- Binomial: $k \in \{4, 10\}$.
- Tiered: $B \in \{4, 10\}$.
- Concurrent: Default ($k = 30, C = 3, D = 10, \lambda = 1.2$).
- Leveled: $B_0 = 2$, $B = 10$, size, STR and R^*-Grove partitioning.

Binomial policy's performance does not change much when $k > 20$. Setting a too small k will lead to merges at almost every flush. Thus $k = 4$ is chosen to test its performance with lower read amplification but higher write amplification, and $k = 10$ for higher read amplification but lower write amplification. The default size ratio (B) is 4 in Apache Cassandra's SizeTiered Compaction Strategy, and is 10 in RocksDB's Universal Compaction. These two values are chosen for Tiered. With a smaller size ratio, merges are more frequent, thus write amplification is higher, but read amplification is lower. On the other hand, a larger size ratio will lead to lower write amplification but higher read amplification. For Concurrent, its default configuration is used, as suggested in [21]. The default size ratio $B = 10$ is used in LevelDB and RocksDB in our experiments, and it is a good comparison to Tiered for the same size ratio setting. The maximum number of files in level 0 (B_0) is reduced from 36 (default in RocksDB) to 2 to further reduce its read amplification.

Both *Hilbert* curve comparator and *Simple* comparator were paired with each configuration. To avoid interference from the primary index, Binomial policy is set to use $k = 8$ for the primary index in all runs. For runs of Leveled policy, a selection algorithm to pick a disk component that overlaps with the fewest disk components in the next level, aiming at minimizing their write amplification, is used.

Write performance

Write amplification A merge policy with higher write amplification writes more data, which may reduce the write throughput, potentially slow down other operations as well. The write amplification of policies with different configurations for the two datasets are presented in Fig. 8 and Table 3. Write amplification of all stack-based policies are not affected by the dataset because the policies are all content-unaware. Comparators only affect the order of records within disk components, but not component sizes. The write amplification of a stack-based policy is the same for all its configurations, so they

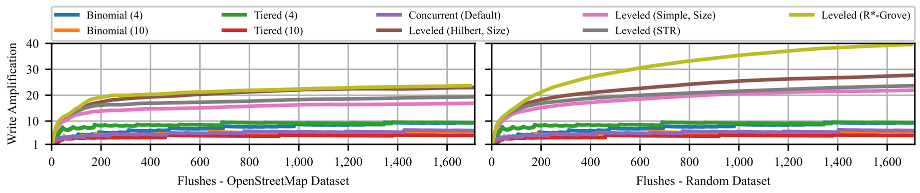


Fig. 8 Write amplification of compared policies with different configurations

Table 3 Overall write amplification of compared policies.

	Binomial		Tiered		Concurrent	Leveled			
	$k = 4$	$k = 10$	$B = 4$	$B = 10$	Default	Hilbert, Size	Simple, Size	STR	R^*-Grove
OpenStreet-Map	9.55	6.05	9.64	4.62	6.58	23.00	16.90	19.38	23.68
Random	9.55	6.05	9.64	4.62	6.63	27.74	21.97	23.65	39.90

are combined in the figure. Binomial with $k = 10$, Tiered with $B = 10$, and Concurrent had the lowest write amplification as they merge infrequently. Binomial with $k = 4$ and Tiered with $B = 4$ had slightly higher write amplification as they merged more eagerly, and Binomial must bound the number of disk components.

All Leveled policy runs had much higher write amplification than any stack-based policy. Write amplification for the random dataset is higher than the OSM dataset. For the random dataset, it has a higher chance of having more overlapping disk components involved in every merge. Runs using R^*-Grove partitioning had the highest write amplification among all and are even more significant in the random dataset. Runs using size partitioning with Hilbert curve comparator had the second highest write amplification, as this setting failed to generate disjoint disk components. Runs using STR partitioning with either comparator had slightly lower write amplification because STR partitioning guarantees disjoint disk components, so merge sizes were smaller on average. Runs using size partitioning with Simple comparator achieved the lowest among them because merging records were ordered by the longitude values; thus, they were partitioned into disjoint groups, creating almost disjoint disk components.

The write amplification of runs using R^*-Grove is much higher than the other runs of Leveled policy, especially in the random dataset. A key reason is that a partitioning algorithm that generates disjoint key ranges can only guarantee that the merged components do not overlap with any other component in the level for single dimensional data, as shown in Fig. 9a, where component 1 from level i has overlapping key range with component 3 and 4 from level $i + 1$ (each rectangle represents the component's key range in the whole key space). However, as shown in Fig. 9b, creating disjoint merged components may fail to guarantee disjoint components in the level. Having overlapping components in a level does not only increase the read amplification, but also increases the write amplification as the probability of merging with more components becomes higher. STR partitioning also has this issue, but it is not so obvious. MBRs created from STR partitioning tend to be tall and thin, which will look like a vertically stretched version of Fig. 9a, in that there will be only a few overlapping components in every level.

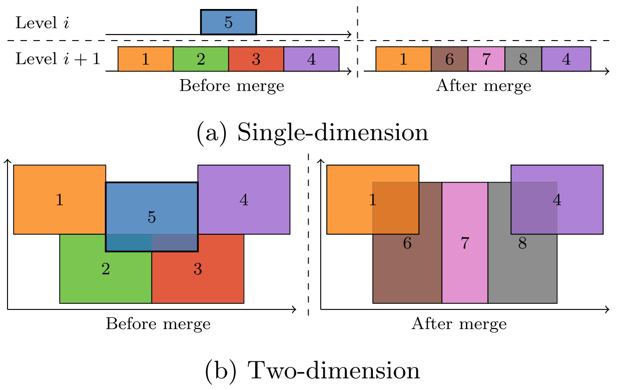


Fig. 9 Components' key boundaries before and after a merge, where components 2, 3 and 5 are merged and replaced by 6, 7 and 8 (illustration purpose only, not from real data)

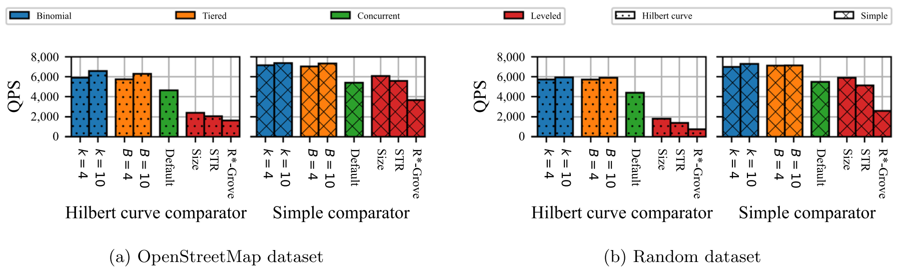


Fig. 10 Average write throughput (requests per second) for all policies with different configurations

But for R^*-Grove, overlapping components can frequently occur, leading to much higher write amplification.

Write throughput The write throughput is further measured and listed in Fig. 10. All stack-based policies showed a very high write throughput. Binomial and Tiered runs had the highest write throughput. Concurrent has much lower write throughput though its write amplification is low. For runs of Leveled policy, write throughput of runs with Simple comparator was very close to Binomial and Tiered despite having high write amplification, runs with Hilbert curve comparator still got the lowest throughput as expected. The write throughput of runs using R^*-Grove partitioning were much lower than the others.

All indexes shared a global memory budget in AsterixDB; any secondary index was always flushed together with the primary index. The write throughput of an LSM secondary index can be dominated by the throughput of the primary index, as the primary index is much larger (2.4 GB vs 100 GB). For this reason, write throughput of Binomial and Tiered runs were slowed down. Write stalls or spikes are not observed in write throughput in the R-tree index either, which should be common in stack-based policies [35, 36].

Hilbert curve comparator is generally slower in computation than Simple comparator as it needs multiple internal iterations to compare two values, significant overheads could be added to write throughput. To verify this hypothesis, a set of small experiments

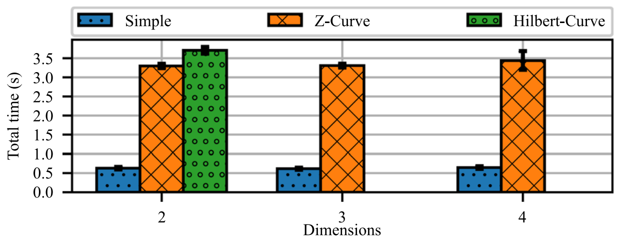


Fig. 11 Total time to sort arrays of 1,000,000 random points. AsterixDB's Hilbert curve comparator only supports two dimensional points due to its slow computation for higher dimensions

have been done, using the same source codes of both comparators plus a Z-order curve comparator from AsterixDB to sort arrays of 1,000,000 random points of 2, 3 and 4 dimensions, respectively. Results in Fig. 11 showed that Simple comparator is about six times faster than the other two.

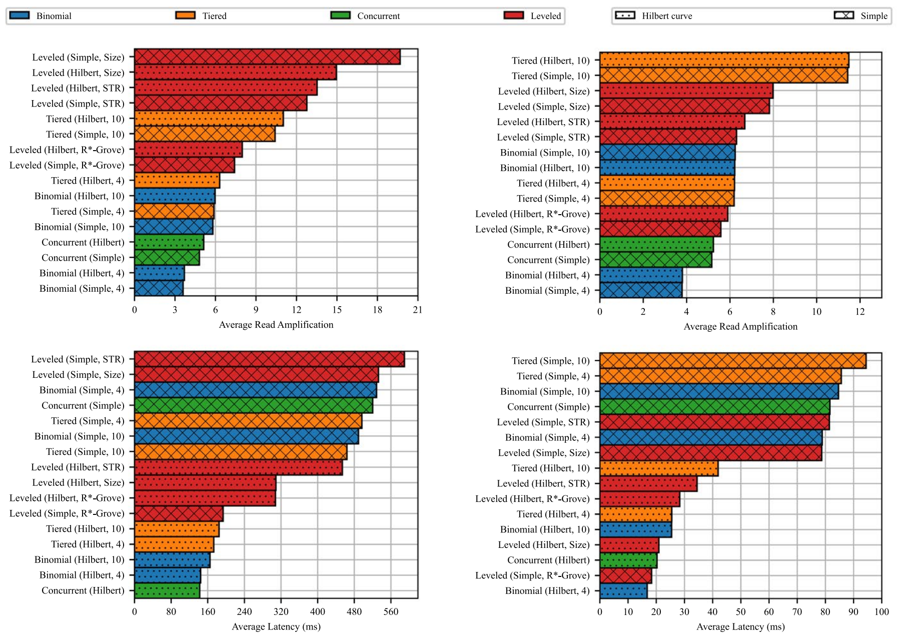
Read performance

The read performance is measured by the following two metrics: (a) average (mean) *read amplification*, i.e., the number of operational disk components of each spatial query, and (b) average (mean) *read latency*, i.e., the total time spent to scan all operational disk components. Latency here is different from query response time in that it measures the time accessing every operational disk component and excludes the time of query compilation and network latency.

High selectivity (10^{-3})

A spatial query with higher selectivity covers a more substantial area, which returns more results on average. The average number of returned records is measured to be about 28,000 for the OSM dataset and 75 for the random dataset. The difference between these two numbers signified from the clustering of the OSM dataset, where a large selectivity query hit more points in highly clustered areas than unclustered areas in the random dataset.

The average read amplification and latency for the OSM dataset are shown in Fig. 12a. In general, the read amplification of a stack-based LSM index is the same as the total number of disk components, because all disk components must be scanned. For leveled LSM index, only 10 to 20 disk components were scanned, even though over 1000 disk components were available; MBR based filtering was very efficient. Two runs using size partitioning had the highest two read amplification. Looking into latency, all policies with Hilbert curve comparator had lower latency than those with Simple comparator, except for the runs using R^*-Grove partitioning. With Hilbert curve comparator, stack-based policies still had the lowest latency numbers, the latency of Leveled policy using size partitioning and R^*-Grove partitioning were not bad. Surprisingly, Leveled policy using R^*-Grove partitioning with Simple comparator achieved very competitive results to the faster five runs of the stack-based policies with Hilbert curve comparator, while most of the other Leveled policy runs were slower. Overall, Hilbert curve comparator would be preferred for large selectivity queries, read latency is almost linearly correlated to the read amplification; thus stack-based policies might be better, but Leveled policy using R^*-Grove partitioning with Simple comparator is also a good option.



(a) OpenStreetMap dataset (b) Random dataset

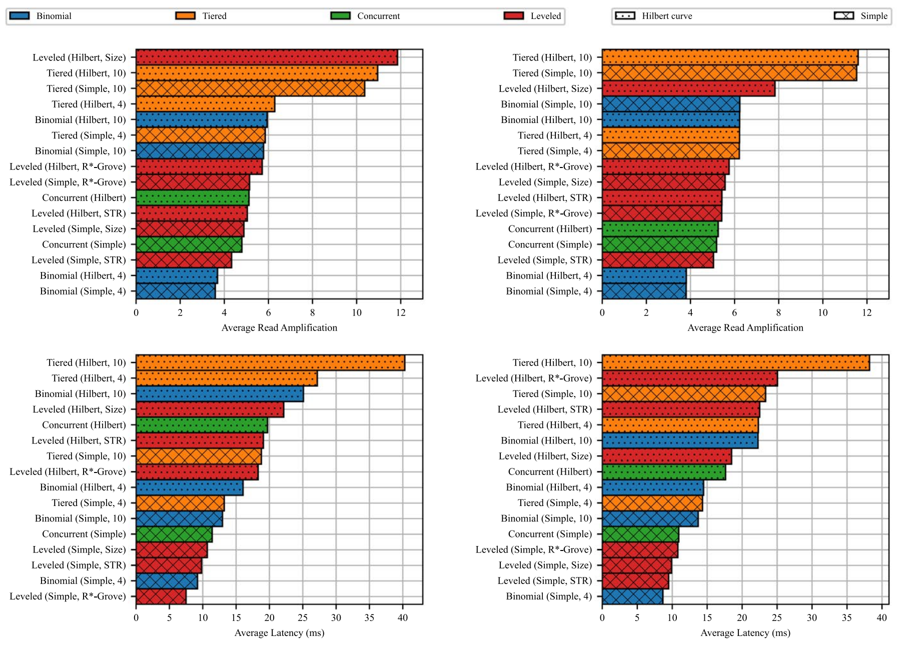
Fig. 12 Average read amplification and latency for selectivity 10^{-3}

For the random dataset, read amplification was about the same as the OSM dataset for all stack-based policies, as shown in Fig. 12b. However, read amplification of Leveled policy runs dropped to below 8 with significant improvements, meaning that MBR based filtering had been even more efficient for this type of dataset. Runs with Hilbert curve comparator still outperformed runs with Simple comparator for read latency, except for the runs using R^*-Grove partitioning. Still, runs of Leveled policy ranked very well among them, especially the run using R^*-Grove partitioning with Simple comparator which ranked second, thanks to their lower read amplification. Latency numbers in the random dataset were 6 to 8 times shorter than the numbers in the OSM dataset. Here, reads tend to be slower in the highly clustered dataset for high selectivity queries.

Low selectivity (10^{-5})

Common spatial queries usually return less than 100 results, which cover a relatively small area. In our experiments, an average of 12 results is measured from the OSM dataset and 1 from the random dataset. However, the number could be 0 most of the time for the random dataset if query rectangles were not generated from existing points.

Similar to high selectivity queries, write amplification of all the stack-based policies remained the same for both datasets, as almost all disk components were scanned, as shown in Fig. 13a, b. Except for one run of Leveled policy using size partitioning with Hilbert curve comparator, the other five runs of Leveled policy became very competitive in both datasets, which even had lower read amplification than some stack-based policies. The two runs using STR partitioning, and the runs using size partitioning and R^*-Grove partitioning with Simple comparator, had much better MBR based filtering for low selectivity queries.



(a) OpenStreetMap dataset (b) Random dataset

Fig. 13 Average read amplification and latency for selectivity 10^{-5}

Because of the lower read amplification and fewer returned records, read latency numbers were all smaller than those in the high selectivity queries. Runs with the Hilbert curve comparator were slower than those with the Simple comparator. The slower computation of the Hilbert curve comparator became a significant bottleneck for low selectivity queries, while it showed superior efficiency for high selectivity queries. The three Leveled runs with Simple comparator were all in the top four among all. Queries could finally take advantage of their better MBR based filtering capabilities to provide much faster index access time. From comparing the two figures of read latency for the two datasets, it can be seen that the numbers are very close to the same policy with the same configuration. The impact of data clustering was not evident on read performance for small selectivity queries.

Discussion

Among all the compared policies, Binomial was the winner in almost all settings, showing the best read amplification and latency numbers, while maintaining the highest write throughput and near-top write amplification. Concurrent was only second to Binomial in terms of read performance in most settings, but it had relatively low write throughput, although its write amplification was low. Its multithreaded merge was a bottleneck in write throughput. There could be some optimizations to multi-threaded merges, but most need hardware or operating system support [7, 37]. Tiered had the lowest write amplification and very high write throughput, but the read performance was sacrificed. The Leveled policy had the highest write amplification, but writes could still be fast with proper configurations. In our experiments, the Leveled policy showed good read performance mostly in low selectivity queries, although the combination of R^{*}-Grove partitioning and Simple comparator achieved outstanding read performance at the cost of the lowest write performance. Therefore it may not be a good option for high selectivity queries in general. There could be cases where it may be better suited. Leveled architecture is a perfect fit for object stores (Amazon S3, Microsoft Azure, etc.) which tend to have many relatively small files (or so-called blobs). Compared to stack-based policies, it can manage records more efficiently via file (disk component) based filtering, rather than relying on local indexes.

In terms of policy configuration, Hilbert curve comparator performed better than Simple comparator in high selectivity queries but was worse in low selectivity queries due to its slow computation. If Leveled policy must be chosen, size partitioning is generally a good option for high selectivity queries, while STR partitioning and R^*-Grove partitioning are still very competitive, especially in low selectivity queries. With a larger index size, STR and R^*-Grove might be better options because they guarantee to create disjoint disk components to have better MBR based filtering capability. However, the higher CPU and memory requirement during merges as well as the low write performance of R^*-Grove must be considered.

How LSM secondary indexes are maintained may have a major impact on the write and read performance of a secondary index. With the eager strategy, write throughput may be determined by the primary index, while with the lazy strategy, read latency may be dominated by the time to verify returned records against the primary index.

Limitations and future work This paper focuses on the write and read performance of R-tree based LSM spatial indexes. Based on the results from [11], comparisons against indexes based on B^+-tree are not included, which may be a more common approach on existing LSM database systems. It may be worthwhile to revisit these designs on different LSM architectures, since B^+-tree usually has better write and read performance than R-tree for certain types of non-intersection spatial queries. The lack of optimizations on hardware and operating system limited the MBR based filtering efficiency for Leveled policy. Some better results would be expected for Leveled policy if some optimizations could be done, such as hardware support for MBR based filtering (e.g. FPGA based filtering) to utilize STR or R^*-Grove partitioning.

Related work

Supporting spatial index

Most LSM systems only support single-dimension indexes such as B^+-tree. To support spatial index, they must rely on some linearization method to project multidimensional data into a single dimension to be loaded in B^+-tree. GeoMesa [25] is a spatial-temporal index that supports so-called Bigtable-style databases including Google Bigtable [5], Apache Accumulo [38], Apache HBase [4]. It uses a customized GeoHash [25, 27] implementation based on the Z-order curve to encode spatial and temporal data into bit strings. STEHIX [39] and Brahimet et al. [26] took a similar approach but only limited to HBase and DataStax Cassandra, respectively. Kim et al. [10, 11] studied five LSM spatial indexes, four of them fall into this category: DHB-tree, DHVB-tree, and SHB-tree all map point data with space-filling curves (Hilbert curve); SIF builds an inverted index (based on B^+-tree) but the main idea is similar to

Table 4 Summary of spatial index support in various systems

	Linearized B^+-tree	Internal R-tree	Exteral R-tree	Inverted index	Separate framework (Spark)
GeoMesa [25]	✓				
STEHIX [39]	✓				
Brahim et al. [26]	✓				
DHB-tree, DHVB-tree, and SHB-tree [10, 11]	✓				
SIF [10, 11]				✓	
Qader et al. [24]				✓	
R-HBase [40]			✓		
BGRP-tree [41]			✓		
Nanjappan [42]			✓		
LevelGIS [43]			✓		
AsterixDB [2]		✓			
LSM RUM-Tree [44]		✓			
Galvizo [45]		✓			
DataStax [46]					✓

SHB-tree. Like SIF, a posting list based LSM inverted index design described in [24] can also be extended to support spatial index.

Another common approach is to build LSM spatial indexes on R-tree. One implementation is to use an external R-tree index to manage multiple LSM trees. R-HBase [40] and BGRP-tree [41] partition the data space into grid cells or regions and use an in-memory R-tree to index the partitions, although the local indexes are still built on B^+-trees. Nanjappan implemented a separate R^*-tree index outside of Cassandra [42]. LevelGIS [43] uses a three-layer hierarchical structure of R-tree index on LevelDB to support spatial queries. Like AsterixDB [2], some choose to use R-tree as the internal index for each component in the LSM tree. LSM RUM-Tree [44] utilizes Update Memo on AsterixDB's R-tree index to support update-intensive spatial workloads, which is orthogonal to our work. [45] described how multi-valued fields are indexed in AsterixDB, which could also be extended to improve the R-tree index performance as a future work. Many open-source projects add spatial index support to LevelDB, RocksDB, and some other LSM systems, most still use the first approach which is B^+-tree with linearized spatial data, only a few of them adopt the LSM R-tree approach. To the best of our knowledge, none of these projects have been deployed in practice. RocksDB used to provide a utility called SpatialDB, but it got abandoned and removed from GitHub since January 2019.

Some systems choose to use an LSM database only for storage and use some other structures for spatial queries. For example, DataStax stores geospatial data in Cassandra, but builds geospatial indexes and handles geospatial queries via Solr [46]. Compared to native LSM secondary spatial index, the key drawback of such systems is that insertions are slower as they need to be written to two or more systems.

All the above spatial index support techniques are summarized in Table 4.

Spatial partitioning algorithms

Partitioning algorithms can highly affect the write and read performance of a leveled LSM *R*-tree index. Some *R*-tree packing algorithms can be directly used for partitioning in merges. The combinations of size partitioning with Hilbert curve comparator and Simple comparator have the same effect as Hilbert Sort [47] and Nearest-X [30], respectively, which are both outperformed by STR partitioning [13]. OMT [48] is a top-down *R*-tree bulk-loading algorithm which might be portable as well. Other partitioning algorithms include sampling-based methods like SpatialHadoop [49] and R^*-Grove [14, 31], and quad-tree-based method like [50] for Hadoop. Most of these algorithms designed for *R*-tree bulk-loading or Hadoop can efficiently handle static data, where the data is written only once. However, they are usually not designed for dynamic data, where the data frequently changes in some write-heavy workloads. A better partitioning algorithm that takes both heavy writes and reads is much desired, that LSM spatial index can benefit a lot from it.

Conclusions

This paper compared and evaluated secondary spatial index performance of both stack-based and leveled LSM architectures with four representative merge policies, on a common platform (AsterixDB). The results from both the OpenStreetMap dataset and the synthetic random dataset have shown that Binomial policy is probably the best candidate for LSM *R*-tree-based spatial index, although it is not specifically optimized for multidimensional spatial data. While having higher write amplification and generally lower write throughput, with proper configuration, Leveled policy can achieve close or even better read performance to some of the better stack-based policies. Although most stack-based policies do not benefit from MBR based filtering at the disk component level, MBR based leveled partitioning can provide much better filtering efficiency to improve spatial query performance in proper settings. Compared to analytic frameworks, a key challenge of MBR based leveled partitioning in LSM tree is to maintain more disjoint square-like MBRs while keeping the write cost low. The selectivity of spatial queries should be considered when choosing a comparator and partitioning algorithm for a Leveled policy.

Abbreviations

LSM	Log-structured merge
MBR	Minimum bounding rectangle
OSM	OpenStreetMap
SSTable	Sorted string table
STR	Sort-tile-recursive

Acknowledgements
Not applicable.

Author contributions
QM led the implementation and the experiments. QM and MAQ worked on the algorithms and the related work. VH came up with the problem definition and overall research direction. All authors contributed to the writing. All authors read and approved the final manuscript.

Funding
This work was supported by NSF Grants IIS-1838222, IIS-1901379 and IIS-2227669.

Availability of data and materials

The datasets generated and/or analyzed during the current study are available in the UCR STAR repository, https://star.cs.ucr.edu/ [33].

Declarations

Ethics approval and consent to participate

Not applicable.

Consent for publication

Not applicable.

Competing interests

The authors declare that they have no competing interests.

Received: 22 March 2022 Accepted: 11 April 2023
Published online: 23 April 2023

References

1. O'Neil P, Cheng E, Gawlick D, O'Neil E. The log-structured merge-tree (LSM-tree). Acta Inf. 1996;33(4):351–85. https://doi.org/10.1007/s002360050048.
2. Alsubaiee S, Altowim Y, Altwaijry H, Behm A, Borkar V, Bu Y, Carey M, Cetindil I, Cheelangi M, Faraaz K. et al. AsterixDB: A scalable, open source BDMS. arXiv preprint 2014;arXiv:1407.0454
3. Lakshman A, Malik P. Cassandra: a decentralized structured storage system. SIGOPS Oper Syst Rev. 2010;44(2):35–40. https://doi.org/10.1145/1773912.1773922.
4. George L. HBase: the definitive guide: random access to your planet-size data. Sebastopol, CA, USA: O'Reilly Media Inc; 2011.
5. Chang F, Dean J, Ghemawat S, Hsieh WC, Wallach DA, Burrows M, Chandra T, Fikes A, Gruber RE. Bigtable: a distributed storage system for structured data. ACM Trans Comput Syst. 2008;26(2):4–1426. https://doi.org/10.1145/1365815.1365816.
6. Dent A. Getting started with LevelDB. Birmingham, UK: Packt Publishing Ltd; 2013.
7. Dong S, Callaghan M, Galanis L, Borthakur D, Savor T, Strum M. Optimizing space amplification in RocksDB. CIDR. 2017;3:3.
8. ScyllaDB Inc. ScyllaDB 2021;https://www.scylladb.com/
9. Alsubaiee S, Behm A, Borkar V, Heilbron Z, Kim Y, Carey MJ, Dreseler M, Li C. Storage management in AsterixDB. Proc VLDB Endowment. 2014;7(10):841–52.
10. Kim Y. Transactional and spatial query processing in the big data era. PhD thesis, University of California, Irvine 2016.
11. Kim Y, Kim T, Carey MJ, Li C. A comparative study of log-structured merge-tree-based spatial indexes for big data. In: 2017 IEEE 33rd International Conference on Data Engineering (ICDE), 2017; pp. 147–150 . IEEE.
12. Mao Q, Jacobs S, Amjad W, Hristidis V, Tsotras VJ, Young NE. Comparison and evaluation of state-of-the-art LSM merge policies. VLDB J. 2021;30(3):361–78. https://doi.org/10.1007/s00778-020-00638-1.
13. Leutenegger ST, Lopez MA, Edgington J. STR: a simple and efficient algorithm for *R*-tree packing. In: Proceedings 13th International Conference on Data Engineering, 1997; pp. 497–506. IEEE.
14. Vu T, Eldawy A. *R*-Grove: growing a family of *R*-trees in the big-data forest. In: Proceedings of the 26th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, 2018; pp. 532–535.
15. Mao Q. Spatial Index on AsterixDB 2020. https://git.io/JUkaj.
16. Ahn JS, Qader MA, Kang WH, Nguyen H, Zhang G, Ben-Romdhane S. Jungle: towards dynamically adjustable key-value store by combining LSM-tree and copy-on-write B^+-tree. In: 11th USENIX Workshop on Hot Topics in Storage and File Systems 2019.
17. Mathieu C, Staelin C, Young NE, Yousefi A. Bigtable merge compaction. arXiv preprint 2014;.arXiv:1407.3008 **abs/1407.3008**.
18. Mao Q, Jacobs S, Amjad W, Hristidis V, Tsotras VJ, Young NE. Experimental evaluation of bounded-depth LSM merge policies. In: 2019 IEEE International Conference on Big Data (Big Data), 2019; pp. 523–532. IEEE.
19. Cassandra: How is data maintained? 2019. https://docs.datastax.com/en/cassandra-oss/3.0/cassandra/dml/dmlHowDataMaintain.html.
20. Facebook Inc. RocksDB Wiki: Universal Compaction 2020. https://github.com/facebook/rocksdb/wiki/Universal-Compaction.
21. Luo C. Merge Policies and Schedulers in AsterixDB 2019. https://cwiki.apache.org/confluence/x/iQ3jBw.
22. Luo C, Carey MJ. LSM-based storage techniques: a survey. VLDB J. 2020;29(1):393–418.
23. Luo C, Carey MJ. Efficient data ingestion and query processing for LSM-based storage systems. 2018; arXiv preprint arXiv:1808.08896.
24. Qader MA, Cheng S, Hristidis V. A comparative study of secondary indexing techniques in LSM-based NoSQL databases. In: Proceedings of the 2018 International Conference on Management of Data, 2018; pp. 551–566.
25. Hughes JN, Annex A, Eichelberger CN, Fox A, Hulbert A, Ronquest M. GeoMesa: a distributed architecture for spatio-temporal fusion. In: Geospatial Informatics, Fusion, and Motion Video Analytics V, 2015;vol. 9473, p. 94730. International Society for Optics and Photonics.
26. Brahim MB, Drira W, Filali F, Hamdi N. Spatial data extension for Cassandra NoSQL database. J Big Data. 2016;3(1):11.
27. Niemeyer G. GeoHash 2008. http://geohash.org.

28. Guttman A. *r*-trees: a dynamic index structure for spatial searching. In: Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data. 1984; pp. 47–57.
29. Beckmann N, Kriegel H-P, Schneider R, Seeger B. The *R**-tree: an efficient and robust access method for points and rectangles. In: Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, 1990; pp. 322–331.
30. Roussopoulos N, Leifker D. Direct spatial search on pictorial databases using packed *R*-trees. In: Proceedings of the 1985 ACM SIGMOD International Conference on Management of Data, 1985; pp. 17–31.
31. Vu T, Eldawy A. *R**-Grove: Balanced spatial partitioning for large-scale datasets. 2020; arXiv preprint arXiv:2007.11651.
32. Haklay M, Weber P. OpenStreetMap: user-generated street maps. IEEE Pervasive Comput. 2008;7(4):12–8.
33. Ghosh S, Vu T, Eskandari MA, Eldawy A. UCR-STAR: the UCR spatio-temporal active repository. SIGSPATIAL Special. 2019;11(2):34–40. https://doi.org/10.1145/3377000.3377005.
34. Apache Software Foundation: Apache AsterixDB 2020. https://asterixdb.apache.org.
35. Luo C, Carey MJ. On performance stability in LSM-based storage systems. Proc VLDB Endow. 2019;13(4): 449–462 . https://doi.org/10.14778/3372716.3372719
36. Yao T, Zhang Y, Wan J, Cui Q, Tang L, Jiang H, Xie C, He X. MatrixKV: Reducing write stalls and write amplification in LSM-tree based KV stores with matrix container in NVM. In: 2020 USENIX Annual Technical Conference (USENIX ATC 20), 2020; pp. 17–31. USENIX Association, Online.
37. Wang P, Sun G, Jiang S, Ouyang J, Lin S, Zhang C, Cong J. An efficient design and implementation of LSM-tree based key-value store on open-channel SSD. In: Proceedings of the Ninth European Conference on Computer Systems, 2014; pp. 1–14.
38. Kepner J, Arcand W, Bestor D, Bergeron B, Byun C, Gadepally V, Hubbell M, Michaleas P, Mullen J, Prout A. Achieving 100,000,000 database inserts per second using Accumulo and D4M. In: 2014 IEEE High Performance Extreme Computing Conference (HPEC), 2014; pp. 1–6. IEEE, Waltham, MA, USA . IEEE.
39. Chen X, Zhang C, Ge B, Xiao W. Spatio-temporal queries in HBase. In: 2015 IEEE International Conference on Big Data (Big Data), 2015; pp. 1929–1937. IEEE.
40. Huang S, Wang B, Zhu J, Wang G, Yu G. R-HBase: a multi-dimensional indexing framework for cloud computing environment. In: 2014 IEEE International Conference on Data Mining Workshop, 2014; pp. 569–574. IEEE.
41. Takasu A, An efficient distributed index for geospatial databases. In: Database and Expert Systems Applications, 2015; pp. 28–42 . Springer.
42. Nanjappan A. R*-Tree index in Cassandra for Geospatial Processing 2019.
43. Xu R, Liu Z, Hu H, Qian W, Zhou A. An efficient secondary index for spatial data based on LevelDB. In: International Conference on Database Systems for Advanced Applications, 2020; pp. 750–754. Springer.
44. Shin J, Wang J, Aref WG. The LSM RUM-Tree: a log structured merge *R*-Tree for update-intensive spatial workloads. In: 2021 IEEE 37th International Conference on Data Engineering (ICDE), 2021. pp. 2285–2290. IEEE.
45. Galvizo G. On indexing multi-valued fields in AsterixDB. Master's thesis, University of California, Irvine 2021.
46. DataStax: Geospatial queries for Point and LineString 2021; https://docs.datastax.com/en/dse/6.0/cql/cql/cql_using/search_index/queriesGeoSpatial.html
47. Kamel I, Faloutsos C. On packing *R*-trees. In: Proceedings of the Second International Conference on Information and Knowledge Management, 1993; pp. 490–499.
48. Lee T, Lee S. OMT: Overlap minimizing top-down bulk loading algorithm for *R*-tree. In: CAISE Short Paper Proceedings, 2003; vol. 74, pp. 69–72.
49. Eldawy A, Mokbel MF. SpatialHadoop: A MapReduce framework for spatial data. In: 2015 IEEE 31st International Conference on Data Engineering, 2015; pp. 1352–1363. IEEE.
50. Whitman RT, Park MB, Ambrose SM, Hoel EG. Spatial indexing and analytics on Hadoop. In: Proceedings of the 22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, 2014; pp. 73–82.

Publisher's Note