Comprehensive Comparison of LSM Architectures for Spatial Data

Qizhong Mao[1§], Mohiuddin Abdul Qader[2§], Vagelis Hristidis[3]
Computer Science and Engineering, University of California, Riverside, USA
Email: { [1]qmao002, [2]mabdu002, [3]evangelo }@ucr.edu

***Abstract*—Spatial indexes in traditional relational databases supported spatial queries in the pre-big data era. However, the volume and ingestion rate of spatial data is increasing rapidly in modern applications. Many big data systems use LSM tree as their storage structure in order to support write-intensive large-volume workloads, which are usually optimized for single-dimensional data. Research has studied how to support spatial indexes on LSM systems, but have mainly focused on the local index organization, that is, how data is organized inside a single LSM component. In this paper, we study various aspects of spatial LSM indexing, including spatial merge policies, which determine when and how spatial components are merged. We consider both stack-based and leveled merge policies, which we have implemented on the same big data system. We evaluate the write and read performance on various workloads and discuss our findings and recommendations. A key finding is that Leveled policies are underperforming other merge policies for most types of spatial workloads.**

***Index Terms*—spatial index, LSM, merge policy, stack-based, Binomial, leveled, R-tree**

I. Introduction

The volume and ingestion rate of spatial data are increasing rapidly due to applications such as navigation systems, location-based review systems, and geo-tagged social media. Database systems have been moving to log-structured merge (LSM) tree [1] storage architectures to facilitate high write throughput. Such systems include Bigtable [2], HBase [3], Cassandra [4], LevelDB [5], RocksDB [6] and AsterixDB [7]. LSM systems provide superior write performance than most relational databases. However, little attention had been paid to the support of LSM 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, not to mention the support of spatial index. LSM-fication is a generic framework to convert a class of indexes to LSM secondary indexes [8]. Using this framework, we have two options to index spatial data. The first option is a B^+-tree-based solution that indexes single-dimensional data projected from multidimensional spatial data through linearization. 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 database system 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 [9], the R-tree-based solution is the preferable option for LSM spatial index, hence in this paper, we focus on LSM R-tree indexes.

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 we consider 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 fixed-size components, with newer components on higher levels; lower levels have more components per level. Stack-based LSM tree usually has 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.

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, we maintain its MBR, 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 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, we employ spatial partitioning algorithms, specifically Sort-Tile-Recursive (STR) [10]. There are several subtle implementation decisions that significantly affect the merge performance. We 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 improves filtering efficiency.

A key contribution of the paper is that we implemented several LSM spatial indexing algorithms on a common database system, AsterixDB, and compared them for read and write performance using several spatial workloads. A key conclusion is that stack-based policies generally perform better with

§Both authors contributed equally.

978-1-7281-6251-5/20/$31.00

generally 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) We study how an LSM architecture can be extended to support secondary spatial indexes (Section II-A). We consider several design decisions and architectures.
2) We propose an optimized partitioning algorithm for Leveled LSM R-tree index, which minimizes the overlap among MBRs while also minimizing the I/O cost (Section II-C).
3) We implemented all compared LSM spatial indexing policies on AsterixDB. Source code is available at [11].
4) We experimentally compared several LSM spatial indexing algorithms using a real-world dataset (Section III).
5) We discuss our observations and recommendations, which challenge the current popularity of Leveled policies (Section IV).

II. LSM Secondary Spatial Index

In this section, we discuss R-tree-based spatial LSM indexes. In Section II-A we explain how data is organized into R-trees and how filtering works. Section II-B discusses details about stack-based policies, while II-C focuses on leveled policies and studies different ways to partition data across components during merges. More details on how secondary indexes are organized, relative to the primary index can be found at [8, 12, 13].

A. Spatial LSM Index based on R-tree

A natural way of organizing spatial records is to place nearby records into the same groups. R-tree [14] and R^*-tree [15] are widely used as local indexes for spatial data, which partition records into disk blocks based on their spatial locations. R-tree and R^*-tree have similar implementation to B^+-tree, except they partition leaf nodes and creates internal nodes by MBRs. Spatial queries may need to traverse multiple paths to leaf nodes to find records. In most cases, R-tree (or R^*-tree) is the preferred option for spatial index [9]; hence in this paper, we only focus on the LSM R-tree designs.

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 include space-filling curve comparators (**Hilbert curve** or Z-order curve), and **Simple** comparator. A space-filling curve comparator sorts records based on the relative order in a space-filling curve. Since space-filling curve comparators are usually slow in computation (Section III-D), we also consider the Simple comparator, which compares two points by the value of each dimension, which is essentially the Nearest-X algorithm [10, 16].

To determine the *operational components* (components whose key ranges overlap with the query key or range) for a spatial query, the key range of every disk component must be checked. The key range of a disk component can be represented by the minimum key $\overline{K}_{min}$ and the maximum key $\overline{K}_{max}$, where $\overline{K}$ represents an array of size 1 for single-dimensional data, or multidimensional data otherwise. For spatial data, they are the low and high points of the MBR. Given two components $C : \langle \overline{K}_{min}, \overline{K}_{max} \rangle$ and $C' : \langle \overline{K'}_{min}, \overline{K'}_{max} \rangle$ and the number of dimensions $D \geq 1$, the two components are overlapping if and only if (1) is satisfied, or disjoint if and only if (2) is satisfied.

$$\forall d \in [1, D] : \overline{K}_{min}[d] \leq \overline{K'}_{max}[d] \wedge \overline{K'}_{min}[d] \leq \overline{K}_{max}[d] \quad (1)$$

$$\exists d \in [1, D] : \overline{K}_{min}[d] > \overline{K'}_{max}[d] \vee \overline{K'}_{min}[d] > \overline{K}_{max}[d] \quad (2)$$

B. Stack-based LSM R-tree Index

Stack-based merge policies merge consecutive disk components and generate a single disk component. These policies are unaware of disk components' contents, that is, merges are scheduled in the same way regardless of the type or the dimensions of the data. Since it is unlikely that inserted records are sorted, disk components have a high chance to have overlapping MBRs, and hence MBR-based filtering of on-disk components are less important. Also, R-tree employs MBR-based filtering on the disk page level; only a small portion of disk components is read even if the component size is large. Hence, even though a spatial query usually needs to scan all disk components, the read amplification is not high. To the best of our knowledge, AsterixDB is the only system that uses stack-based LSM R-tree indexes.

We select to use the **Binomial** stack-based merge policy [17] in our experiments, which has been shown to outperform other policies [18]. The Binomial policy maintains an optimal write cost with very low read amplification,using only one parameter k, which bounds the maximum number of disk components.

C. Leveled LSM R-tree Index

In a **Leveled** LSM tree [5, 6], every level is a sorted run which contains multiple disjoint disk components of the same size. 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 buffer level 0 which contains B_0 potentially overlapping disk components, which holds flushed disk components as multiple sorted runs. 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. We have implemented the Leveled policy on AsterixDB for our experiments.

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.

A first approach is to use Hilbert or Z-order curve comparator and size-based partitioning, where we sort records based on their Hilbert or Z-order values and split to components when the maximum size of a component is reached. This approach

may lead to components overlapping with each other. A visual example is available on the left of Figure 1. Merges may create many overlapping disk components at the same level, making MBR based filtering very inefficient and increasing the read amplification significantly. To improve MBR based filtering efficiency, a better partitioning algorithm that can create spatially disjoint disk components can be very helpful.

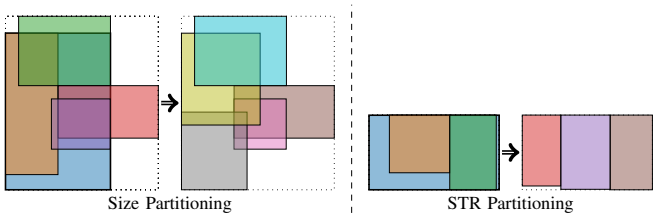

Fig. 1: MBRs of disk components before and after a merge by Size and STR partitioning, with Hilbert curve comparator. Every rectangle corresponds to a disk component. The two sub-graphs are in different scales.

A second approach is to partition using Sort-Tile-Recursive (STR) [10], which was originally proposed to pack pages in R-tree for point data. We adopt this partitioning algorithm in leveled LSM R-tree index. When disk components are merged, we apply STR 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, we apply STR to the center points of spatial objects, but a component's MBR is computed from the points' actual MBRs. Size partitioning only fetches one disk page from each merging disk component at a time, so the memory requirement is minimal. STR partitioning, however, requires storing all records in memory. Each record requires two copies, one from the input stream and one from the partitioned groups. For the smallest case which a record is 24 bytes (2 double and 1 long), and a pointer of 8 bytes, partitioning 1 million records would require at least 3 GB of memory. In practice, the partitioning process may also require disk I/O due to swapping. Moreover, the partitioned records may need to be sorted again using the comparator to be bulk-written into R-trees. Overall, STR partitioning demands much higher CPU and memory usage than size partitioning and is generally slower during construction. An example of STR partitioning in a merge is illustrated in Figure 1 on the right, which inputs are some overlapping disk components, outputs are disjoint disk components.

A third approach is to create disjoint disk components through a combination of Simple comparator and size partitioning. With this configuration, records are strictly ordered and split by the first dimension value, such that (2) $\overline{K}_{min}[1]' > \overline{K}_{max}[1]$ always holds, if C' is created later than C in the same merge. Both STR partitioning and this combination have a common problem as the way of organizing records are very similar: they both tend to create very thin rectangles with very small width but large height. Even though they can effectively create disjoint disk components, a spatial query with relatively wide MBR may cover a small portion of MBRs of many disk components, potentially increasing the read amplification of certain types of spatial queries.

III. Experimental Evaluation

A. Dataset and Workloads

We used a geo-location dataset of exactly 100 million 2D points in all experiments. It is a real-world dataset randomly sampled from OpenStreetMap (OSM) [19]. Points in the OSM dataset are highly clustered in urban areas all over the world, especially in the United States and western Europe (Figure 2).

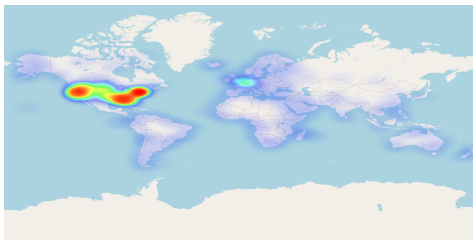

Fig. 2: Heatmap of the OSM dataset, from $[-180°, -90°]$ to $[180°, 90°]$.

For the OSM dataset, we generated a workload with interleaved reads and writes 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 1,000 bytes as a synthetic attribute (e.g., geo-location description). Points are stored as two double 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. We also tested other selectivity values with $\sigma \in \{1, 2\}$ and $\sigma \in [6, 10]$. We observed the same results for $\sigma \in \{1, 2\}$ with $\sigma = 3$, and $\sigma \in [6, 10]$ with $\sigma = 5$, hence we only reported results for $\sigma \in \{3, 5\}$ ($\sigma = 4$ and $\sigma = 5$ are very similar). AsterixDB provides several builtin 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, kNN and distance join, it is reasonable to focus on this type of rectangular intersection queries.

B. Experimental Setup

Apache AsterixDB [20] 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^+-trees, the spatial index is stored as LSM R-trees. All secondary indexes and the primary index share a global memory budget; thus they are always flushed together. AsterixDB uses an 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 Amazon *m5.large* instances in the same region. Each instance has 2 vCPUs running on Intel Xeon Platinum 8175M, 8 GB of memory, and 200 GB general purpose SSD (gp2). AsterixDB was configured to use a single node. Other configurations were set to the defaults. The average size of the flushed disk components in LSM R-tree index was around 2 MB.

C. Merge Policies Compared

We applied Binomial policy [17, 18] with $k \in \{4, 10\}$, Leveled policy [5, 6] with $B_0 = 2$, $B = 10$, Size and STR partition to the LSM R-tree index. Both *Hilbert* curve comparator and *Simple* comparator were paired with each configuration. Binomial policy with $k = 8$ was set for the primary index in all runs. For runs of Leveled policy, we used 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.

D. 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. We present the write amplification of policies with different configurations in Figure 3. Write amplification of Binomial is not affected by the dataset because it is content-unaware. Comparators only affect the order of records in disk components, but not component sizes. The write amplification of Binomial are the same for each configuration, so they are combined in the figure. Binomial with $k = 10$ had the lowest write amplification as they merged infrequently. Binomial with $k = 4$ had slightly higher write amplification as they merged more eagerly to bound the number of disk components.

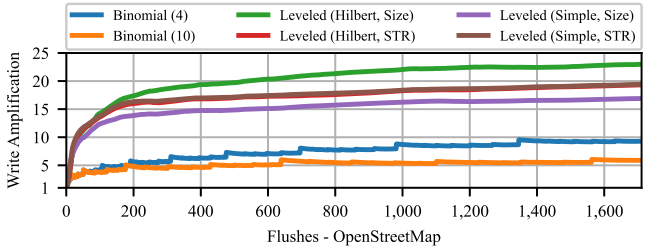

Fig. 3: Write amplification of compared policies with different configurations.

All Leveled policy runs had much higher write amplification than Binomial. Runs using size partitioning with Hilbert curve comparator had the 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. 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.

Write Throughput: We have further measured the write throughput and listed the numbers in Table I. Binomial policy showed a very high write throughput. For runs of Leveled policy, write throughput of runs with Simple comparator was very close to Binomial despite they had high write amplification, runs with Hilbert curve comparator still got the lowest throughput as expected.

Policy	**Binomial**				**Leveled**			
Comparator	Hilbert		Simple		Hilbert		Simple	
Configuration	4	10	4	10	Size	STR	Size	STR
Avg. records/sec	5,903	6,576	7,155	7,370	2,409	2,062	6,070	5,581

TABLE I: Average write throughput (number of records written per second) for both policies with different configurations.

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 v.s 100 GB). For this reason, write throughput of Binomial runs were slowed down. We did not observe write stalls or spikes in write throughput in the R-tree index either, which should be common in stack-based policies like Binomial [21, 22].

Hilbert curve comparator is generally slower in computation than Simple comparator as it needs multiple internal iterations to compare two records, significant overheads could be added to write throughput. To verify this hypothesis, we ran a set of small experiments 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. Results in Figure 4 showed that Simple comparator is about six times faster than the other two.

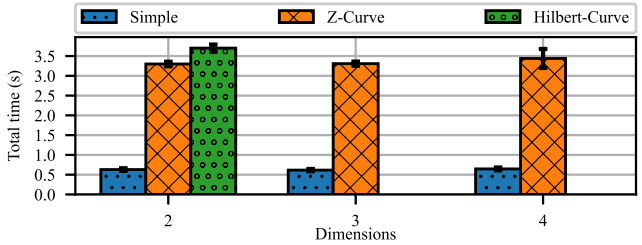

Fig. 4: Total time to sort arrays of 1,000,000 random points. AsterixDB's Hilbert curve comparator only supports 2 dimensional points.

E. Read Performance

We measured the read performance 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 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. We measured the average number of returned records of about 28,000 for the OSM dataset.

The average read amplification and latency for the OSM dataset are shown in Figure 5. In general, the read amplification of a stack-based LSM index is the same as the number of disk components, because all disk component must be scanned. For leveled LSM index, only 10 to 20 disk components were scanned though over 1,000 disk components were available; MBR based filtering was very efficient. Two runs using size partitioning had the highest two read amplification. Looking into latency, both policies with Hilbert curve comparator had lower latency than those with Simple comparator. With Hilbert curve comparator, Binomial still had the lowest latency numbers, the latency of Leveled policy using size partitioning was not bad. Overall, for large selectivity queries, Hilbert curve comparator would be preferred, read latency is almost linearly correlated to the read amplification; thus Binomial might be better.

Low Selectivity (10^{-5}): Common spatial queries usually return less than 100 results, which cover a relatively small area. In our experiments, we measured an average of 12 results from the OSM dataset.

Similar to high selectivity queries, write amplification of Binomial remained the same, as almost all disk components were scanned, as shown in Figure 6. Except for one run of Leveled policy using size partitioning with Hilbert curve comparator, the other three runs of Leveled policy became competitive. Runs using STR partitioning, and the run using size partitioning with Simple comparator, had much better MBR based filtering for low selectivity queries.

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 Simple comparator were faster than those with Hilbert curve comparator. The slower computation of Hilbert curve comparator became a significant bottleneck for small selectivity queries, while it showed superior efficiency for high selectivity queries. The two Leveled runs using STR partitioning ranked second and third among all. Queries could finally take advantage of their better MBR based filtering capabilities to provide much faster index access time.

IV. Discussion

Between the two compared policies, Binomial was the winner in almost all settings, showing the best read amplification and latency, while maintaining the highest write throughput and near-top write amplification. The Leveled policy had the highest write amplification, but writes could still be fast with proper configurations. In our experiments, the Leveled policy only showed good read performance in low selectivity queries, and therefore it may not be a good option for high selectivity queries. There could be cases where it may suit better. Leveled architecture is a perfect fit for object stores (Amazon S3, Microsoft Azure, etc.). Comparing 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 is still very competitive, especially in low selectivity queries. With a larger index size, STR might be a better option because it guarantees to create disjoint disk components to have better MBR based filtering capability. However, its higher CPU and memory requirement during merges must be considered.

LSM secondary index maintenance strategy 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: In this paper we focused on the write and read performance of R-tree based LSM spatial indexes. Based on the results from [9], we did not include comparisons against indexes based on B^+-tree, 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. We would expect some better results 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 partitioning.

V. Related Work

Several LSM systems have included support for spatial data, but these approaches are mostly an after-thought, that is, they are built on top of a standard key-value store. In contrast, our work studies how spatial indexing can be a native indexing approach. R-HBase [23] and BGRP-tree [24] 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 [25]. LevelGIS [26] uses a three-layer hierarchical structure of R-tree index on LevelDB to support spatial queries. Various open-source projects add native spatial index support to LevelDB, RocksDB, and other LSM systems, by creating a B^+-tree with linearized spatial data. 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 was abandoned and removed from GitHub in January 2019. Most works still focus on B^+-tree-based spatial index as an extension framework of existing

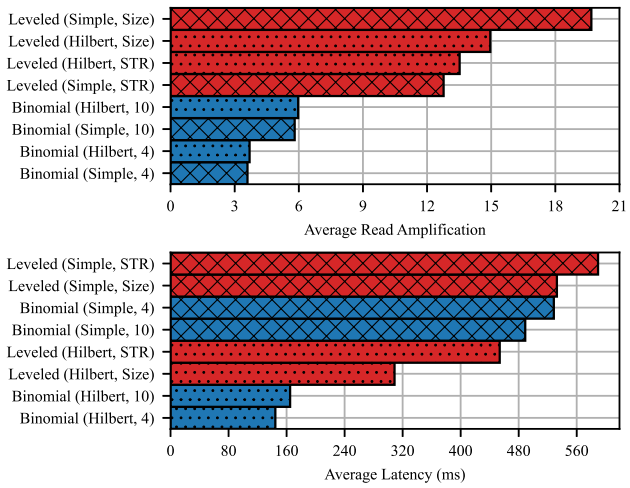

Fig. 5: OSM dataset, selectivity 10^{-3}.

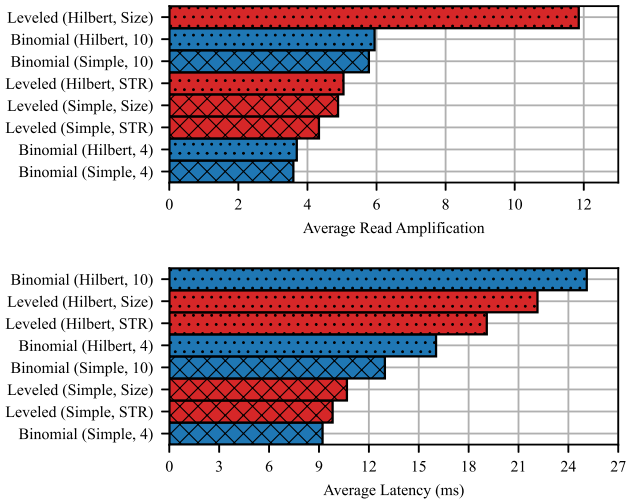

Fig. 6: OSM dataset, selectivity 10^{-5}.

systems [27, 28]. AsterixDB is currently the only BDMS that adopts the native LSM R-tree approach.

VI. Conclusions

In this paper, we compared and evaluated secondary spatial index performance of stack-based and leveled LSM architectures with two representative merge policies, on a common platform (AsterixDB). The results have shown that Binomial policy is probably the best candidate for LSM R-tree-based spatial index, even though it is not specifically optimized for spatial data. With proper configuration, Leveled policy can achieve close performance. MBR based leveled partitioning can provide better filtering efficiency at the disk component level to improve spatial query performance in proper settings. We also showed that the choice of different comparators and partitioning algorithms for a Leveled policy depends on spatial queries' selectivity.

Acknowledgment

This work was supported by NSF grants IIS-1619463, IIS-1838222 and IIS-1901379.

References

[1] P. O'Neil *et al.*, "The log-structured merge-tree (LSM-tree)," *Acta Inf.*, vol. 33, no. 4, pp. 351–385, Jun. 1996.
[2] F. Chang *et al.*, "Bigtable: A distributed storage system for structured data," *ACM Trans. Comput. Syst.*, vol. 26, no. 2, pp. 4:1–4:26, Jun. 2008.
[3] L. George, *HBase: the definitive guide: random access to your planet-size data*. O'Reilly Media, Inc., 2011.
[4] A. Lakshman *et al.*, "Cassandra: A decentralized structured storage system," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, pp. 35–40, Apr. 2010.
[5] A. Dent, *Getting started with LevelDB*. Packt Publishing Ltd, 2013.
[6] S. Dong *et al.*, "Optimizing space amplification in RocksDB." in *CIDR*, vol. 3, 2017, p. 3.
[7] S. Alsubaiee *et al.*, "AsterixDB: A scalable, open source BDMS," *arXiv preprint arXiv:1407.0454*, 2014.
[8] S. Alsubaiee *et al.*, "Storage management in AsterixDB," *Proceedings of the VLDB Endowment*, vol. 7, no. 10, pp. 841–852, 2014.
[9] Y.-S. Kim *et al.*, "A comparative study of log-structured merge-tree-based spatial indexes for big data," in *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. IEEE, 2017, pp. 147–150.
[10] S. T. Leutenegger *et al.*, "STR: A simple and efficient algorithm for R-tree packing," in *Proceedings 13th International Conference on Data Engineering*. IEEE, 1997, pp. 497–506.
[11] Q. Mao, "Spatial index on AsterixDB," 2020. [Online]. Available: https://git.io/JUkaj
[12] C. Luo *et al.*, "Efficient data ingestion and query processing for LSM-based storage systems," *arXiv preprint arXiv:1808.08896*, 2018.
[13] M. A. Qader *et al.*, "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.
[14] A. Guttman, "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.
[15] N. Beckmann *et al.*, "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.
[16] N. Roussopoulos *et al.*, "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.
[17] C. Mathieu *et al.*, "Bigtable merge compaction," *arXiv preprint arXiv:1407.3008*, vol. abs/1407.3008, 2014.
[18] Q. Mao *et al.*, "Experimental evaluation of bounded-depth LSM merge policies," in *2019 IEEE International Conference on Big Data (Big Data)*. IEEE, 2019, pp. 523–532.
[19] M. Haklay *et al.*, "OpenStreetMap: User-generated street maps," *IEEE Pervasive Computing*, vol. 7, no. 4, pp. 12–18, 2008.
[20] Apache Software Foundation, "Apache AsterixDB," 2020. [Online]. Available: https://asterixdb.apache.org
[21] C. Luo *et al.*, "On performance stability in LSM-based storage systems," *Proc. VLDB Endow.*, vol. 13, no. 4, p. 449–462, Dec. 2019.
[22] T. Yao *et al.*, "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)*. USENIX Association, Jul. 2020, pp. 17–31.
[23] S. Huang *et al.*, "R-HBase: A multi-dimensional indexing framework for cloud computing environment," in *2014 IEEE International Conference on Data Mining Workshop*. IEEE, 2014, pp. 569–574.
[24] A. Takasu *et al.*, "An efficient distributed index for geospatial databases," in *Database and Expert Systems Applications*. Springer, 2015, pp. 28–42.
[25] A. Nanjappan, "R*-Tree index in Cassandra for geospatial processing," 2019.
[26] R. Xu *et al.*, "An efficient secondary index for spatial data based on LevelDB," in *International Conference on Database Systems for Advanced Applications*. Springer, 2020, pp. 750–754.
[27] J. N. Hughes *et al.*, "GeoMesa: a distributed architecture for spatio-temporal fusion," in *Geospatial Informatics, Fusion, and Motion Video Analytics V*, vol. 9473. International Society for Optics and Photonics, 2015, p. 94730F.
[28] M. B. Brahim *et al.*, "Spatial data extension for Cassandra NoSQL database," *Journal of Big Data*, vol. 3, no. 1, p. 11, 2016.