2017 IEEE 33rd International Conference on Data Engineering

A Comparative Study of Log-Structured Merge-Tree-Based Spatial Indexes for Big Data

Young-Seok Kim†
Samsung Advanced Institute of Technology (SAIT),
Samsung Electronics. Co., Ltd.
ys24.kim@samsung.com

Taewoo Kim, Michael J. Carey, Chen Li
Department of Computer Science,
University of California, Irvine
{taewok2, mjcarey, chenli}@ics.uci.edu

***Abstract*—The proliferation of GPS-enabled mobile devices has generated geo-tagged data at an unprecedented rate over the past decade. Data-processing systems that aim to ingest, store, index, and analyze Big Data must deal with such geo-tagged data efficiently. In this paper, among representative, disk-resident spatial indexing methods that have been adopted by major SQL and NoSQL systems, we implement five variants of these methods in the form of Log-Structured Merge-tree-based (LSM) spatial indexes in order to evaluate their pros and cons for dynamic geo-tagged Big Data. We have implemented the alternatives, including LSM-based B-tree, R-tree, and inverted index variants, in Apache AsterixDB, an open source Big Data management system. This implementation enabled comparison in terms of real end-to-end performance, including logging and locking overheads, in a full-function, query-based system setting. Our evaluation includes both static and dynamic workloads, ranging from a "load once, query many" case to a case where continuous concurrent incremental inserts are mixed with concurrent queries. Based on the results, we discuss the pros and cons of the five index variants.**

I. Introduction

During the past decade, diverse geo-tagged data such as texts, photos, and videos have been generated at an unprecedented rate from GPS-enabled devices. The trend will be further accelerated by the advent of the Internet-of-Things era, where literally everything could involve GPS-enabled sensors and generate geo-tagged data. In addition, popular NoSQL systems such as [1], [2], [3], [4] have adopted Log-Structured Merge (LSM) Trees [5] as their storage structure in order to support such high-frequency data generation. LSM-trees amortize the cost of writes by batching updates in memory before writing them to disk, thus avoiding random writes.

There have been many studies in the spatial data processing area, from proposing new spatial index data structures to evaluating and analyzing those proposed indexes [6], [7]. Most of the studies, however, have dealt with "load once, query many", i.e., static workloads. Also, many of the measures in those studies consider only I/O without including CPU or index access costs instead of an overall system's end-to-end processing cost. Moreover, LSM-based spatial indexing methods have received little attention. A few recent studies [8], [9] have investigated LSM-based or insert-optimized spatial indexing methods with dynamic workloads that reflect continuous incremental inserts and concurrent queries.

Given this context, now is an appropriate time to revisit spatial indexing methods. The contributions of this work are:

1) Among representative, *disk-resident* spatial indexing methods that have been adopted by major SQL and NoSQL systems (see Table I), we implement five variants as Log-Structured Merge-tree-based (LSM) spatial indexes in Apache AsterixDB, an open source Big Data management system [9].

R-tree-based methods	Non-R-tree-based methods
Oracle, IBM Informix Spatial DataBlade, PostgreSQL (which also supports GiST), MySQL, Couchbase	Oracle, IBM DB2, MS SQL Server, MongoDB, Apache Lucene

TABLE I: Spatial indexing methods supported by major SQL and NoSQL systems, where Non-R-tree-based methods use a well-known scheme which maps two dimensional point objects into one dimensional sequence values and stores/retrieves them into/from a typical one dimensional index structure, e.g., a B-tree.

2) Among the five variants, three are based on B-trees, one on R-trees, and one on inverted indexes, which are three of the most popular disk-resident indexes. With this setup, we can answer an interesting question: If built-in indexes like B-tree and inverted indexes are used to implement spatial indexing, can they be as efficient as or even superior to R-trees?

3) We focus on real-world geo-tagged point data in our evaluation and cover a broad spectrum of workloads, from a "load once, query many" case without incremental inserts to a case with continuous concurrent insertions and queries.

The remainder of the paper is organized as follows. First, we briefly explain the basic idea of LSM-trees and review how AsterixDB transforms traditional indexes into LSM-tree indexes. Next, we summarize the details of the five spatial indexes, then present our evaluation strategy and results.

II. Background

A. LSM-trees

An LSM-tree [5] is an ordered, persistent index structure that supports typical operations such as insert, delete, and search. It is optimized for frequent or high-volume updates. By first batching updates in memory, the LSM-tree amortizes the cost of an update by converting what would have been several disk seeks into some portion of a sequential I/O. Entries being inserted into an LSM-tree are initially placed into a component of the index that resides in main memory – an *in-memory component*. When the space occupancy of the in-memory component exceeds a specified threshold, entries are sequentially *flushed* to disk – a *disk component*. As the number of disk components increases, disk components are periodically merged together subject to a *merge policy* that decides when and what to merge.

B. Storage management in Apache AsterixDB

Apache AsterixDB [10], [11] is a parallel, semi-structured information management platform that provides the ability to

†This study was performed while this author was at UC Irvine.

2375-026X/17 $31.00 © 2017 IEEE
DOI 10.1109/ICDE.2017.61

ingest, store, index, query, and analyze mass quantities of data. Its storage layer includes a framework for converting a class of indexes (including conventional B-trees, R-trees, and inverted indexes) with basic operations such as insert, delete, and bulkload into LSM-based secondary indexes. The framework serves as a coordinating wrapper that orchestrates the creation and destruction of LSM components and delegates operations to the appropriate components as needed. See [9] for more details of the framework and [12] for details of the LSM indexes in AsterixDB, such as LSM B-trees, LSM R-trees, and LSM inverted indexes, with transactional support.

III. FIVE LSM SPATIAL INDEXES

This section describes the five indexes, which are named the R-tree, DHB-tree, DHVB-tree, SHB-tree, and SIF indexes. The R-tree is the LSM R-tree index. The DHB-tree, DHVB-tree, and SHB-tree are each based on an underlying LSM B-tree index. SIF is based on an LSM inverted index.

A. R-tree

An LSM R-tree consists of an in-memory component and zero or more disk components. An in-memory component of an LSM R-tree consists of a traditional R-tree along with a deleted-key B-tree that captures deleted entries. A disk component is a variant of an R-tree, where it orders indexed entries using a Hilbert curve when loading the tree. The deleted-key B-tree is useful when merge-based $reconciliation$ (a process to find non-deleted entries throughout all in-memory and disk components) is not available; in contrast, merge-based reconciliation is available for LSM B-trees since they maintain entries in a totally ordered manner [9].

An insert operation inserts an entry $e = \langle sk, pk \rangle$ into the in-memory component via traditional R-tree insertion logic, with sk and pk representing a secondary key and its corresponding primary key, respectively. A delete operation deletes an entry $e = \langle sk, pk \rangle$ from an in-memory R-tree if e exists. It also inserts e into the associated deleted-key B-tree; this step serves as a sentinel by preventing entries of e in an older component's R-tree from being returned. A flush operation creates a new disk component as follows: Entries in the in-memory component are sorted using a Hilbert curve, as are entries in the associated deleted-key B-tree. During both sorts, the entries are compared relatively based on the Hilbert curve. (Relative comparisons are explained in Section III-B.) A new disk component is then created by merging and bulk-loading the two sets of sorted entries. During this process, an entry e from the deleted-key B-tree becomes an anti-matter entry e' in the new disk component if an identical entry e does not appear in the sorted R-tree entry list. Otherwise, e from the deleted-key B-tree is ignored. A search operation creates a range-scan cursor consisting of a heap of sub-cursors on the disk components and gets their reconciled entries; those entries must be checked against the in-memory component's deleted-key B-tree, and surviving entries from the check are returned by the range cursor at the end. A merge operation gets reconciled entries like the search operation, but without checking the in-memory component's deleted-key B-tree since a merge operation only merges disk components. [12] offers further details.

B. DHB-tree and DHVB-tree

Details of indexing and querying 2D spatial point objects using a B-tree with space-filling curves (such as Hilbert and Z curves) are described in previous studies [13], [14], [15], [16]. We use the Hilbert curve since it is considered to be superior to other space-filling curves [14]. The DHB-tree and DHVB-tree were implemented using the main ideas from [16], which details how to index 2D spatial points using a B-tree with space-filling curves and how to support spatial range queries.

In general, both our DHB-tree and DHVB-tree indexes store point objects and support range queries. However, there is a key difference in terms of how they compare points. One approach is to compute Hilbert sequence numbers for two given points and then compare the resulting numbers. An alternative is to compare the points relatively, i.e., without computing full sequence numbers. Relative comparisons can be faster than the absolute sequence number comparisons if the two points lie far away from each other. The DHB-tree uses the relative-comparison method, so stored entries in DHB-tree index have coordinates and primary key fields but no stored sequence number. Note that this means that whenever a comparison is needed, the relative comparison steps are repeated. The DHVB-tree adopts the absolute-comparison method, so its index entries have the coordinates, primary key fields, and also a Hilbert sequence number. The tradeoffs between these two methods will be examined in the experiments.

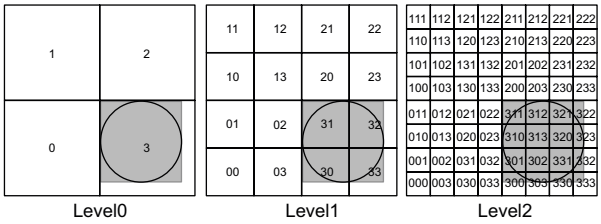


Fig. 1: A 3-level 2×2 grid hierarchy and cell numbers at each level

C. SHB-tree

In the SHB-tree indexing method, a two-dimensional space is statically decomposed into a k-level $2^n \times 2^n$ grid hierarchy, where k and n are numbers chosen when the index is created. The top level ($level\ 0$) has $2^n \times 2^n$ cells, and each successive level further decomposes a cell at the previous level into $2^n \times 2^n$ cells. Also, the top-level cells are numbered in a linear fashion by using a Hilbert curve, and so are the sub-cells belonging to a parent cell at the previous level (by prepending the ascendant cell numbers). Figure 1 depicts a 3-level 2×2 grid hierarchy and its cell numbers at each level. Since the top level has four cells and each cell has four sub-cells, the sequence numbers are from the first-order Hilbert curve.

To store a spatial object in the SHB-tree, the MBR of the object is decomposed into a set of cells, each of which either overlaps with the MBR or is completely covered by the MBR according to the following $covering\ rule$. If the MBR completely covers some cell at a level, that cell is said to be *covered* by the MBR. A covered cell is included in the result set and not decomposed at lower levels. The rest of the overlapping cells, if any, are decomposed further at the next level. This rule applies at all levels of the grid hierarchy except that overlapping cells at the lowest level are included in the result set without further decomposition. Note that a point object will always be mapped into a cell at the lowest level. Once the set of cells are computed for an object being inserted, each cell number (as a key) is inserted into the underlying B-tree, where a cell number also includes the level number of the cell at the end. Each SHB-tree index entry thus consists of a $\langle cell\ number, primary\ key \rangle$ pair.

When a query region is given, a set of cell numbers for the MBR of the query region are computed and used to search for matching cell numbers of indexed objects. Searching for the set of MBR cell numbers over the index can be optimized, when there are consecutive cell numbers, by forming a range search with the consecutive numbers. This optimization can effectively reduce the number of index searches. False positives caused by (1) the MBR of the query region, (2) the MBR of the stored spatial objects, and/or (3) the overlapping, non-covered cells at the lowest level are all discarded during the post-processing step. Cause (1) is not relevant to a rectangular region query, and cause (2) is not relevant to point objects. In general, a finer granularity of cells at the lowest level can reduce the number of false positives due to cause (3), but it may also increase the number of index searches resulting from reading more cells for a given query region's MBR.

This grid-based approach is particularly interesting since it is used for spatial indexing in MS SQL Server [17]. The Linear Quadtree [18] in Oracle Spatial [19] works in a similar way, except that its number of sub-cells in a cell is fixed (at four) and its cells are numbered in a quadtree-based manner.

D. SIF

SIF provides a way to support spatial indexing based on inverted indexing. The main idea is similar to the SHB-tree except that SIF uses an inverted index. A spatial object is mapped to a set of cell numbers, which are stored in an inverted index as a set of tokens. Similarly, a region query goes through a mapping step to obtain a set of cell numbers and then searches these cell numbers just like how a set of keyword tokens are searched for in an inverted index.

In general, an inverted index supports exact match searches for a given string token, where a set of string tokens can be generated from a given query string. Some inverted indexes provide richer search types, such as prefix or range searches, based on the underlying data structure (such as trie or B-tree) used to implement the index. We used the LSM inverted index in AsterixDB as a black box to implement SIF; as a result, our SIF index only supports exact matches. SIF thus has more overhead than the SHB-tree due to its lack of range search support. As described earlier, a range search in the SHB-tree can reduce the number of searches when contiguous cell numbers result from a give query region. Nonetheless, a related optimization is available for SIF to reduce the number of searches over the underlying LSM inverted index: a set of child cell numbers can be replaced with their common parent cell number if these child cells completely cover the parent cell. This optimization requires a complete spatial object to be captured at all levels. However, it comes with an increased index size due to storing a point object at each of the levels in a grid hierarchy. We will see that the increased index size can degrade this indexing method's performance.

IV. Evaluation Plan

Our evaluation used an 8-node cluster to host an AsterixDB instance, with each node running CentOS Linux on a Quadcore Intel Xeon CPU E3-1230 V2 3.30GHz, 16GB RAM, 1 GBit Ethernet NIC, and three locally attached 7,200 rpm SATA drives set up as RAID 0. In each node, a dataset was stored in four separate partitions to provide more parallelism. In total, a dataset in the 8-node AsterixDB instance was horizontally partitioned into 32 partitions based on primary key. In AsterixDB, the secondary indexes of a given dataset are local to the primary index. See [12] for more on our experimental setup.

Index	Fields of an index entry (size in bytes)
DHB-tree	point (16), pk (8)
DHVB-tree	Hilbert sequence number (8), point (16), pk (8)
R-tree	point (16), pk (8)
SHB-tree	cell number (7), point (16), pk (8)
SIF	cell number (7), pk (8), point (16)

TABLE II: Each index entry's fields with their sizes in bytes

We implemented the DHVB-tree with 64-bit Hilbert sequence numbers and the same resolution was given to the DHB-tree. We configured both the SHB-tree and SIF index to have a 6-level $2^4 \times 2^4$ grid hierarchy. A cell's size at the bottom level is around 2 meters $\times$ 1 meter (2m$\times$1m), with 2m for the x-axis and 1m for the y-axis. Table II shows the details of each index entry's fields and their sizes at the logical level. Each index's entry also includes the original point in order to support *index-only scans* when a query includes predicates that can be evaluated using just the fields in the secondary index. See [12] for more on how index-only scan queries and non-index-only scan queries are processed in AsterixDB.

A. Spatial Dataset

We obtained a set of real-world GPS point data from $OpenStreetMap$ [20], which includes more than 2.7 billion points. Because the obtained data includes only the points themselves, represented as a latitude and a longitude, we generated synthetic tweets using the obtained point data for this evaluation. We uniformly sampled 1.6 billion points out of the 2.7 billion points and augmented them with tweet data. These 1.6 billion tweet records amount to 1.6 TB.

B. Workloads

We used two classes of workloads: one is static and the other is dynamic. The first has no insertions after data is loaded and it is called the *Static workload*. In contrast, the dynamic class involves continuous data arrivals; one variant of this class, *Dynamic workload 1*, is ingestion-only, while another variant has concurrent queries as well and is called *Dynamic workload 2*. For the queries, we examine both select and join queries, both of which use a secondary spatial index as their access path. These queries fall into two categories; one is an index-only-scan query that only accesses a secondary spatial index, and the other is a non-index-only-scan query that searches a secondary spatial index first and then accesses the primary index due to a predicate not covered by the secondary index. See [12] for further workload details.

V. Experimental results

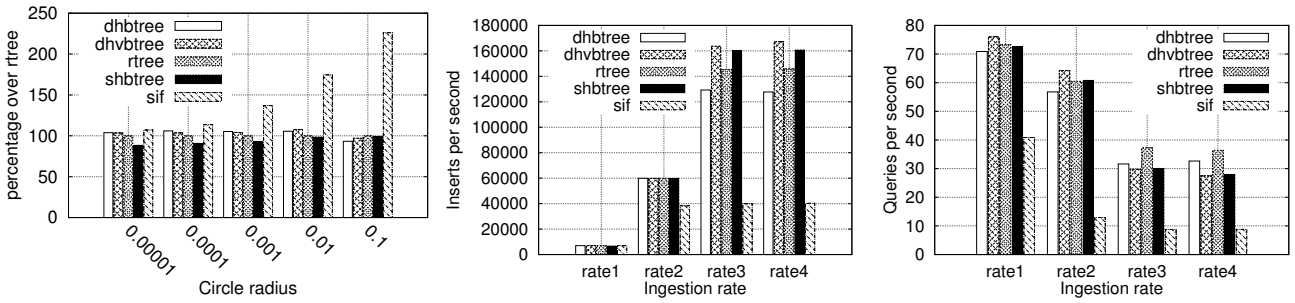
Due to space limitations, we present only a subset of our full results here. See [12] for additional results.

A. Static workload results

Table III shows the primary index ("pidx") size after loading 1.6 billion records as well as the elapsed time for data loading. Also, it shows the elapsed times for creating each spatial secondary index on the sender location field of the 1.6 billion tweet records and the corresponding index sizes. The results for the index-only-scan select queries are shown in Figure 2(a). This figure shows, for each spatial index, its query response time percentage relative to the query response time of the LSM R-tree.

B. Dynamic workload 2 results

Dynamic workload 2 tests each index's ability to ingest and query concurrently while the data arrival rate varies. Figures



(a) Static workload: select query response time (b) Dynamic workload 2: inserts per second (IPS) (c) Dynamic workalod 2: queries per second (QPS)

Fig. 2: Experimental results: for the queries in the workloads, the spatial region size, i.e., the circle radius, was varied from 0.00001 degree (about 1 meter) to 0.1 degree (about 10km). For the dynamic workload 2, the rate 1, rate 2, rate 3, and rate 4 represent making tweet generators sleep 1 millisecond after every 1, 10, 100, and 1000 generated record(s), respectively.

	pidx	dhbtree	dhvbtree	rtree	shbtree	sif
Index size (GB)	1001.53	46.24	59.66	47.74	62.64	353.70
Index creation time (minutes)	85.02	25.74	18.02	17.08	17.14	46.60

TABLE III: Index size and creation time

2(b) and 2(c) show the results of the index-only-scan query case for this workload. Figure 2(b) shows the inserts per second (IPS) for each spatial index for each ingestion rate after one hour of ingestion while concurrent queries are being processed. Figure 2(c) shows the corresponding queries per second (QPS).

C. Summary of the results

Here we summarize the main lessons from the experiments. First, without concurrent data arrivals (static workload), for small-circle range queries, the SHB-tree outperformed the R-tree in terms of query response time due to the index-range-scan mechanism enabled by the underlying B-tree. For larger-circle range queries, however, this effect was washed out by more false positives and more searches caused by having more cells. Second, with ingestion but no concurrent querying (dynamic workload 1), the DHVB-tree and SHB-tree outperformed the R-tree in terms of IPS [12]. Third, with concurrent data ingestion and querying (dynamic workload 2), the DHVB-tree outperformed the R-tree in terms of QPS at slower data arrival rates; there was no IPS difference among the indexes except for SIF. For higher incoming data rates, the DHVB-tree and SHB-tree outperformed the R-tree in terms of IPS, but the R-tree outperformed them in terms of QPS. In addition, the DHB-tree's insert performance was worse than the DHVB-tree's due to the overhead of its relative comparisons. SIF, whose underlying data structure is an inverted index, performed the worst overall, due to its lack of range and prefix search support, both for ingestion performance (IPS) and query performance (QPS).

Except for SIF, there was neither a clear winner nor a clear loser considering both insert and query performance. Query differences were mostly modest in our real end-to-end system setting. This result was especially true for large-circle non-index-only-scan queries with many results, where final primary key lookups were costly. If we had to pick one winner from an end user's perspective, we would choose the R-tree index since it performed well and does not need tuning such as picking a proper number of grid levels and the number of cells in each grid; tuning was required for the SHB-tree and SIF to find a sweet spot to trade off false positives versus more searches.

Acknowledgements The AsterixDB project has been supported by an initial UC Discovery grant, by NSF IIS awards 0910989, 0910859, 0910820, and 0844574, and by NSF CNS awards 1305430 and 1059436. It has also enjoyed support from Amazon, eBay, Facebook, Google, HTC, Microsoft, Oracle Labs, Infosys, and Yahoo! Research. We thank Pierre-Jean Rollando for asking a seemingly innocent question about SQL Server spatial indexing that inspired us to look more deeply at these issues. We also thank Odej Kao of TU Berlin for providing our access to a cluster there for the experiments.

REFERENCES

[1] F. Chang *et al.*, "Bigtable: A distributed storage system for structured data," *ACM Trans. Comput. Syst.*, vol. 26, no. 2, 2008.

[2] "Cassandra," http://cassandra.apache.org/.

[3] "LevelDB," https://code.google.com/p/leveldb/.

[4] "HBase," http://hbase.apache.org/.

[5] P. E. O'Neil *et al.*, "The Log-Structured Merge-Tree (LSM-Tree)," *Acta Inf.*, vol. 33, no. 4, pp. 351–385, 1996.

[6] H. Samet, *Foundations of Multidimensional and Metric Data Structures*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005.

[7] V. Gaede and O. Günther, "Multidimensional access methods," *ACM Comput. Surv.*, vol. 30, no. 2, pp. 170–231, 1998.

[8] C. M. Jermaine *et al.*, "The partitioned exponential file for database storage management," *VLDB J.*, vol. 16, no. 4, pp. 417–437, 2007.

[9] S. Alsubaiee *et al.*, "Storage management in AsterixDB," *PVLDB*, vol. 7, no. 10, pp. 841–852, 2014.

[10] "Apache AsterixDB," https://asterixdb.apache.org/.

[11] S. Alsubaiee *et al.*, "AsterixDB: A scalable, open source BDMS," *PVLDB*, vol. 7, no. 14, pp. 1905–1916, 2014.

[12] Y.-S. Kim, "Transactional and spatial query processing in the big data era," Ph.D. dissertation, University of California, Irvine, 2016.

[13] J. A. Orenstein and T. H. Merrett, "A class of data structures for associative searching," in *PODS Conference*, 1984, pp. 181–190.

[14] H. V. Jagadish, "Linear clustering of objects with multiple atributes," in *SIGMOD Conference*, 1990, pp. 332–342.

[15] C. Faloutsos and Y. Rong, "DOT: A spatial access method using fractals," in *ICDE Conference*, 1991, pp. 152–159.

[16] J. K. Lawder, "The application of space-filling curves to the storage and retrieval of multi-dimensional data." Ph.D. dissertation, Birkbeck College, University of London, 2000.

[17] Y. Fang *et al.*, "Spatial indexing in Microsoft SQL Server 2008," in *SIGMOD Conference*, 2008, pp. 1207–1216.

[18] I. Gargantini, "An effective way to represent quadtrees," *Commun. ACM*, vol. 25, no. 12, pp. 905–910, 1982.

[19] K. V. R. Kanth *et al.*, "Quadtree and R-tree indexes in oracle spatial: a comparison using GIS data," in *SIGMOD Conf.*, 2002, pp. 546–557.

[20] "OpenStreetMap's bulk GPS point data," https://blog.openstreetmap.org/2012/04/01/bulk-gps-point-data/.