Indexing the Pickup and Drop-off Locations of NYC Taxi Trips in PostgreSQL – Lessons from the Road*

Jia Yu[1] and Mohamed Sarwat[2]

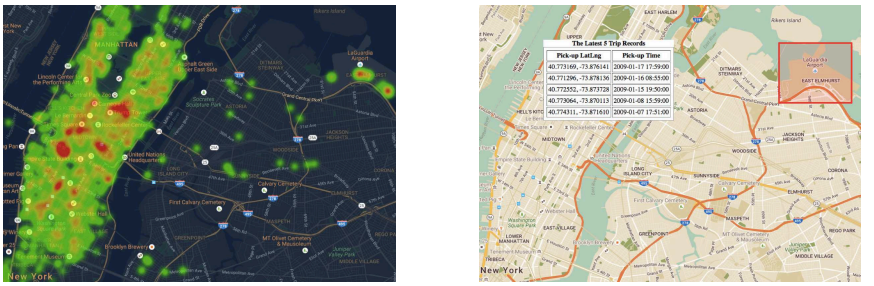
School of Computing, Informatics, and Decision Systems Engineering
Arizona State University, Tempe, Arizona 85281
{jiayu2[1],msarwat[2]}@asu.edu

Abstract. In this paper, we present our experience in indexing the drop-off and pick-up locations of taxi trips in New York City. The paper presents a comprehensive experimental analysis of classic and state-of-the-art spatial database indexing schemes. The paper evaluates a popular spatial tree indexing scheme (i.e., GIST-Spatial), a Block Range Index (BRIN-Spatial) provided by PostgreSQL as well as a new indexing scheme, namely Hippo-Spatial. In the experiments, the paper considers five evaluation metrics to compare and contrast the performance of the three indexing schemes: storage overhead, index initialization time, query response time, maintenance overhead, and throughput. Furthermore, the benchmark takes into account parameters that affect the index performance, which include but is not limited to: data size, spatial query selectivity, and spatial area density, The paper finally analyzes the experimental evaluation results and highlights the key insights and lessons learned. The results emphasize the fact that there is no one size that fits all when it comes to indexing massive-scale spatial data. The results also prove that modern database systems can maintain a lightweight index (in terms of storage and maintenance overhead) that is also fast enough for spatial data analytics applications. The source code for the experiments presented in the paper is available here: https://github.com/DataSystemsLab/hippo-postgresql

1 Introduction

The volume of available geospatial data increased tremendously and such data keeps evolving in unprecedented rates. For instance, New York City Taxi and Limousine Commission has recently released a taxi dataset (abbr. NYC Taxi) [1]. The dataset contains close to 200 Gigabytes of New York City Yellow Cab and Green Taxi trips. The dataset contains detailed records of over 1.1 billion individual taxi trips in the city from January 2009 through December 2016. Each record includes pick-up and drop-off dates/times, pick-up and drop-off precise location coordinates, trip distances, itemized fares, and payment method. Figure 1(a) depicts a heat map of the NYC taxi trips. To make sense of the NYC

* This work is supported by the National Science Foundation Grant 1654861



(a) NYC Taxi Trips Heat Map

(b) Taxi Trips in the Laguardia Airport Regions

Fig. 1: NYC Taxi Trips

taxi data, the first step is to digest the dataset in a database system. The user can then issue spatial queries using SQL, e.g., find all Taxi trips to Laguardia airport (see Figure 1 (b)).

To speed up such queries, a user may build a spatial index, e.g., R-tree, on the location or geometry attribute. Even though classic database indexes [8, 13] improve the query response time, they usually yield close to 15% additional storage overhead. Although the overhead may not seem too high for small databases, it results in non-ignorable cost in massive-scale spatial database scenarios, e.g., Taxi trips locations. Moreover, existing database systems take a lot of time in initializing and bulk loading the spatial index (e.g, R-Tree or Quad-Tree [10, 20, 24]) especially when the size of indexed spatial data reaches hundreds of Gigabytes or more. Furthermore, spatial indexes supported by state-of-the-art spatial database systems, e.g., PostGIS [4], are designed with the implicit assumption that the underlying spatial data does not change much. However, many modern applications constantly insert new spatial data into the database, e.g., inserting a new taxi trip record. Maintaining a database index incurs high latency since the DBMS has to locate and update those index entries affected by the underlying table changes. For instance, maintaining an R-Tree searches the tree structure and perhaps performs a set of tree nodes splitting or merging operations. That requires plenty of disk I/O operations and hence encumbers the time performance of the entire DBMS in update intensive application scenarios.

In this paper, we present our experience in indexing the drop-off and pick-up locations of NYC taxi trips. The paper presents a comprehensive experimental analysis of classic and state-of-the-art spatial database indexing schemes supported in PostgreSQL (a popular open source database system) [23, 5]. The paper evaluates a popular spatial tree indexing scheme (i.e., GiST-Spatial [15, 18, 14, 2]), a Block Range Index [3] (denoted as BRIN-Spatial) provided by PostgreSQL as well as a new indexing scheme, namely Hippo-Spatial [28, 27]. The experiments consider five main evaluation metrics, briefly described as follows: (1) Storage overhead: the extra storage space occupied by the spatial index structure, (2) Index initialization time: the time the system takes to create and bulk load the index. (3) Query response time: the time the database system takes to

search the index and retrieve the corresponding spatial data (i.e., Taxi trips), (4) Maintenance overhead: the time the system takes to maintain the spatial index in response to data insertion or deletion. (5) Throughput: the number of data access operations, given a hybrid query/update workload, the database system can process on the indexed table in a given time period. Furthermore, the benchmark takes into account parameters that affect the index performance, which include but is not limited to: data size, spatial query selectivity, and spatial area density, The paper finally analyzes the experimental evaluation results and highlights the key insights and lessons learned. The results emphasize the fact that there is no one size that fits all when it comes to indexing massive-scale spatial data. The results also prove that modern database systems can maintain a lightweight index (in terms of storage and maintenance overhead) that is also fast enough for spatial data analytics applications.

The rest of the paper is organized as follows. Section 2 describes the spatial database indexing approaches considered in the benchmark. Section 3 describes the experimental environment and setup. Sections 4 to 7 explains the experimental evaluation results and their analysis. Finally, Section 8 highlights the key lessons learned from the benchmark.

2 Studied Spatial Database Indexing Schemes

This section gives an overview of the spatial database indexing schemes considered in the analysis. Section 2.1 summarizes the Generalized Search Tree (GiST-Spatial) indexing scheme while Section 2.2 gives an overview of Block Range Indexes (BRIN-Spatial). Section 2.3 highlights the details of the Hippo-Spatial indexing scheme.

2.1 Generalized Search Tree (GiST-Spatial)

Index structure A Generalized Search Tree is a balanced search tree that accepts arbitrary data types including spatial data [15]. It holds the similar <key, pointer> tree structure like B-Tree or R-Tree [9, 13, 19, 26] but the key varies according to the data type. To index spatial data [11, 12, 29, 13, 16], the key is 2 dimensional rectangle which is the Minimum Bounding Rectangle (MBR) of its child nodes. In a non-leaf node, the pointer points to its child node while in a leaf node, the pointer points to the parent table tuple. In other words, GiST-Spatial's basic idea is to group nearby spatial objects together and use a upper tree node stores their Minimum Bounding Rectangle (MBR) as well as pointers. Let m be the minimum allowed child nodes, N be the number of records, $Levels = [log_m N]$. A GiST-Spatial can contain up to $\sum_{i=1}^{Levels} [N/m^i]$ nodes. The GiST-Spatial bulk-loading [17] algorithm runs in a bottom-up fashion, which is indeed faster than inserting tuple one by one. The bulk loading algorithm generates plenty of tree nodes besides tuples pointers and, in practice, it writes many temporary files onto disk for scalability.

Index search. The index search algorithm takes as input a spatial rectangular range predicate. The algorithm starts at the root node and traverses the child nodes that satisfy the spatial predicate. The algorithm then prunes subtrees in GiST-Spatial, which possess MBRs that do not intersect with the spatial query predicate. The algorithm performs this step recursively until it reaches the tree leaf level and finally returns all spatial objects that lie within the spatial query range. The tree structure of GiST-Spatial offers fast index search on highly selective queries at the cost of excessive indexing and maintenance overhead.

Index maintenance. Insertion/Deletion in GiST-Spatial is similar to the R-Tree in the sense that it maintains a balanced tree structure. The update algorithm first traverses the tree and finds the node where the new index entry <key, pointer> should be inserted in / deleted from. For insertion, in case there is no space available, the target tree node will be split and GiST-Spatial will adjust other tree nodes to ensure the tree balance is preserved. For deletion, GiST-Spatial will merge tree nodes with extra space caused by data deletion and also adjust the tree nodes.

2.2 Block Range Index (BRIN-Spatial)

Index structure. As opposed to GIST, BRIN-Spatial is a sparse index [6, 22, 25, 21] that only stores pointers to disk pages in the indexed table. BRIN-Spatial groups pages into a fixed disk page range unit (128 pages per range by default). Each index entry in BRIN-Spatial contains two components: a static disk page range (e.g., page 1 - 10) and a Minimum Bounding Rectangle (MBR) that encloses all spatial data tuples that are recorded in the page range. The index initialization algorithm scans the indexed table only once to generate BRIN-Spatial. For each page range, BRIN-Spatial reads all tuples to construct the MBR for each index entry. An example of BRIN-Spatial is given in Figure 2.

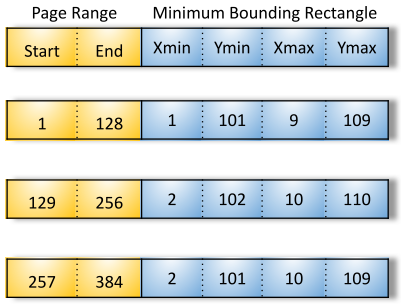


Fig. 2: BRIN-Spatial

Index search. Given a spatial range query, the query processor only searches the index entries for which the MBRs intersect with the spatial query predicate. It is highly recommended to ensure that the indexed spatial objects physically maintain their spatial locality in a certain way, e.g., sorting by longitude/latitude coordinate or Hilbert curve. In that case, the index entries keep the minimal MBR overlap between each other. Under this premise, BRIN-Spatial is able to prune lots of disk pages without scanning them.

Index maintenance. For a newly inserted tuple, BRIN-Spatial first finds the page range it belongs to and then checks the tuple against the MBR of

this page range. If the tuple is outside the MBR, BRIN-Spatial updates the MBR to cover the tuple otherwise BRIN-Spatial does nothing. For deletions, BRIN-Spatial does not update any index entries after delete tuples to improve performance. The underlying database will make a note on deleted tuples and make sure these tuples disappear from the returned tuples even if BRIN-Spatial returns them by mistake.

2.3 Hippo-Spatial

Index structure. Hippo[1] is a data-aware sparse index [28]. In context of spatial data, each Hippo (denoted as Hippo-Spatial) index entry is composed of two components: a dynamic disk page range and a histogram-based page range summary (depicted in Figure 3). In the summary, specifically, the simplified histogram (called partial histogram), each bit shows whether the corresponding two dimensional bucket presents (1) in this page range or not (0). The histogram-based summary is extracted from the two complete load balanced 1D histograms on X and Y axises, respectively (visualized histograms given in Figure 3). Such histograms are widely supported and naturally maintained by most existing DBMSs and execute with no much extra cost. Two 1D histogram buckets, one from X axis and one from Y, represent a 2D bucket. We number a 2D histogram bucket by its 1D buckets on X and Y. For example, bucket (1,1) represents the bucket on the lower-left corner of Figure 3 histogram. Hippo-Spatial iterates each parent table tuple and groups as ranges contiguous similar pages (in terms of data distribution). In the partial histogram of each page range, distinct histogram buckets hit by tuples are marked as 1 in corresponding bits. Hippo-Spatial ensures that the partial histogram in each index entry has the same density:

$$Partial\ histogram\ density\ (D)\ =\ \frac{\#\ Buckets_{value=1}}{\#\ Buckets_{complete\ histogram}}$$

Index search. When a spatial range query is issued, the system first locates the histogram buckets cover / intersect / covered by the query predicate and outputs a partial histogram similar to the histogram-based summary maintained for each index entry. Then, the search algorithm reads each index entry and filters out the index entries for which the histogram-based summary has no common buckets with the query predicate. For all index entries that match the query predicate, the search algorithm inspects the corresponding disk pages and the qualified data tuples are returned.

Index maintenance. When a new tuple is inserted, Hippo-Spatial updates the index entries in an eager manner. It first finds the 2D histogram bucket where the tuple falls in and then runs a binary search on index entry sorted list to locate the page range which the tuple belongs to. The sorted list maintains a list of index entry pointers that are sorted in the ascending order of their start page ID. If this tuple hits a distinct histogram bucket, the partial histogram

[1] Source code: https://github.com/DataSystemsLab/hippo-postgresql

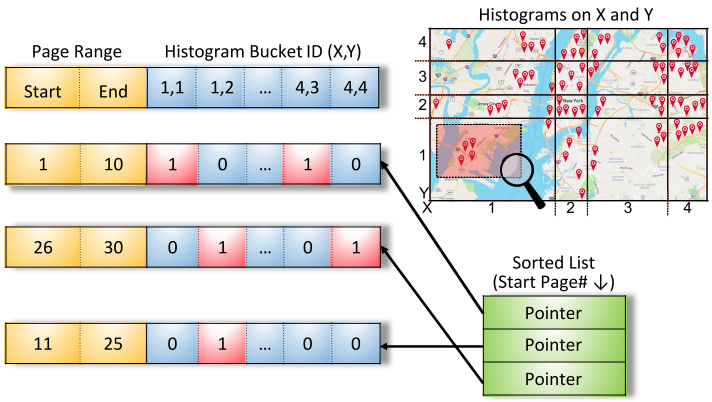


Fig. 3: Hippo-Spatial - Index Structure

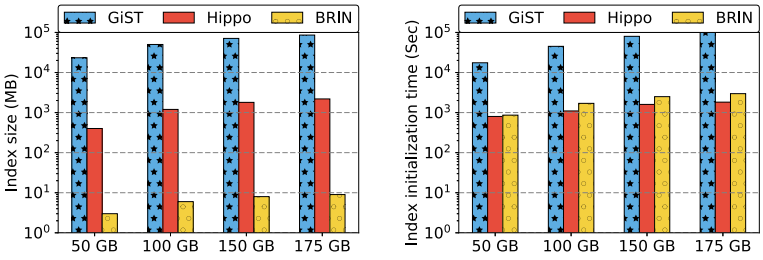
in Hippo-Spatial index entry will set the corresponding bit to 1; if no distinct buckets hit, Hippo-Spatial does nothing instead. On the other hand, Hippo-Spatial deletion runs in a lazy manner. This means Hippo-Spatial updates the index entries only for a batch of deletion operations. During the update, Hippo-Spatial scans the index entries and in case some tuples in a certain page range are deleted, Hippo-Spatial will re-summarize all pages in this page range and update the index entry.

3 Experimental Environment

We conduct the experiments on PostgreSQL 9.6 and PostGIS 2.3 with 128 MB default buffer pool. After fully loading the NYC city taxi trip dataset into PostgreSQL, the corresponding NYC Taxi trips table occupies 25 million PostgreSQL disk pages on the test machine. The size of PostgreSQL default buffer pool is rather small while the operating system memory is too large to be ignored. To avoid the impact of pre-cached data, we clear OS cache before each single transaction. We leverage the `EXPLAIN ANALYZE`, a PostgreSQL built-in performance analysis tool, to capture the execution time of all transactions and count the disk I/O operations. We use the default PostgreSQL 9.6 settings in all experiments. We use the `(CREATE INDEX)` (given below) to build the specified index on top of the NYC taxi trip table in PostgreSQL:

```
CREATE INDEX hippo_idx ON NYCTaxi USING Hippo (PickUpLocation);
```

All indexes are built on the NYC Taxi dataset pick-up location (i.e., latitude and longitude coordinate) attribute. For the sake of GiST-Spatial and BRIN-Spatial, the latitude and longitude coordinates are represented by a single coordinate attribute in PostgreSQL compatible geometry format. Two Hippo-Spatial indexes with the same configuration are built on latitude and longitude, respectively. BRIN-Spatial allows a parameter called Pages Per Range (P) which specifies



(a) Index size (b) Index initialization time

Fig. 4: Indexing overhead on different data scales (logarithmic scale)

the number of parent table pages summarized by each index entry. We use 32, 128 (default) and 512 to tune BRIN-Spatial. Hippo-Spatial accepts a parameter named Density (D) to control the partial histogram density inside each index entry. Its performance is also impacted by the number of buckets in the complete histogram (H). We choose three parameter combinations to tune Hippo-Spatial: (1) D = 20% H = 400 (default setting) (2) D = 40% H = 400 (3) D = 20% H = 800. All indexes use their default settings unless otherwise stated.

We issue a spatial range query on NYC Taxi table with a particular query window and qualified tuples are returned to the psql front-end. The format used in the experiments is:

```
EXPLAIN ANALYZE SELECT count(*) FROM NYCTaxi WHERE <predicate>;
```

The predicate represents a spatial range query window targeted at the pick-up attribute written in an index-dependent format. All insertions work in an eager manner to ensure the query correctness. In the experiments, we use the `(INSERT INTO NYCTaxi VALUES (aTrip))` SQL command to inserts a new Taxi trip tuple in the NYC taxi trips table. We also use `(COPY NYCTaxi FROM aFile)` command to insert a batch of tuples in a single operation in order to avoid unnecessary I/O. Nonetheless, it still performs the insertion/index update tuple by tuple. In PostgreSQL, a `DELETE` operation just makes a note on the deleted tuples and hides them from the output instead of immediately removing them physically. That is due to the fact that clearing and recycling deleted tuples' physical space is a time-consuming process. All physical deletions and corresponding index updates only happen when the `VACUUM` command is invoked. The `VACUUM` command runs periodically but also accepts manual invocation from the user.

4 Studying the indexing overhead

This section studies the indexing overhead incurred by the three compared indexing schemes. We build three indexes on different sizes of the New York Taxi Trip data and record the corresponding overhead (Figure 4) including index size and index initialization time. Results of using different index parameters are described in Figure 5 and Figure 6.

4.1 Index Size

As depicted in Figure 4a, Hippo-Spatial occupies close to two orders of magnitude less storage space than GiST-Spatial. That happens due to the fact that GiST-Spatial stores the pointers of hundreds of millions of Taxi trips in the table and maintains a Minimum Bounding Rectangle in each tree node. On the other hand, Hippo-Spatial only stores disk page ranges and MBR summaries. A tuple pointer is a physical address that consists of a disk page ID and slot ID. Once the index search is completed, GiST-Spatial collects the pointers and passes them to the DBMS. Given a tuple pointer, the DBMS directly jumps to the specified address and retrieves the embedded tuple without any rechecks. Retrieving a small amount of pointers during queries is fast, yet storing 1.1 billion tuple pointers in an index is very space-consuming. In addition, each MBR is represented by four double values, minimum X and Y, maximum X and Y, also occupies non-negligible storage space. On the contrary, each Hippo-Spatial index entry only contains a disk page range and a concise summary. Generally speaking, a disk page may store 50 - 100 tuples, and that is why Hippo-Spatial incurs much less storage overhead.

As given in Figure 4a, Hippo-Spatial leads to more storage overhead than BRIN-Spatial. That happens because Hippo-Spatial, as opposed to BRIN-Spatial, is data-aware and hence speeds up the search process. Each Hippo-Spatial index entry stores a histogram-based page summary instead of a simple MBR. Nonetheless, the extra storage space occupied by Hippo-Spatial is relatively small since its size is less than 1% of the indexed table.

Figure 5 studies the storage overhead of both BRIN-Spatial and Hippo-Spatial using different parameter settings. For instance, Hippo-D20%-H400 denotes a hippo index with density set to 20% and the number of histogram buckets set to 400 and BRIN-P128 denotes a BRIN-Spatial index with 128 pages per range. Hippo-Spatial occupies 100 times larger disk space than BRIN-Spatial. That makes sense because each index entry in Hippo-Spatial maintains a histogram-based summary of a dynamic page range while BRIN-Spatial only stores the Minimum Bounding Rectangle per each page range. Each summary in Hippo-Spatial represents a partial histogram and each bucket in this histogram is represented by a single bit. Although Hippo-Spatial compresses these partial histograms, they are still much larger than a simple MBR. As the number of pages per range increases, BRIN-Spatial occupies less disk space since it summarizes more pages within one range at the cost of slower query response time. For different Hippo-Spatial parameter combinations, The

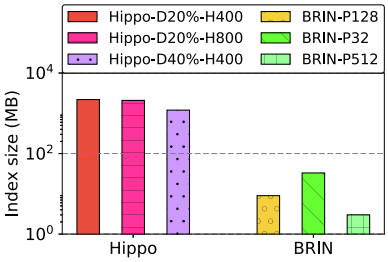


Fig. 5: Index size (log. scale)

higher the histogram density, the more pages each Hippo-Spatial index entry summarizes. That will also lead to more tuples being summarized by each index entry. Maintaining the same density but increasing the total number of histogram buckets leads to an increase in the storage space occupied by Hippo-Spatial. That happens because more complete histogram buckets also leads to more tuples hitting more distinct buckets in each partial histogram.

4.2 Index initialization time

Figure 4b depicts the index initialization time incurred by creating each of the three indexes in PostgreSQL. The system takes the same time to bulk load Hippo-Spatial and BRIN-Spatial because each of them scans the indexed table tuple by tuple and summarizes each encountered tuple using an in-memory validation operation. The only difference is that, given a tuple, Hippo-Spatial finds the histogram bucket to which the tuple belongs using binary search while BRIN-Spatial checks whether the retreived tuple is covered by the temporary MBR and updates the MBR if needed. Moreover, PostgreSQL spends two orders of magnitude more time to bulk load GiST-Spatial compared to BRIN-Spatial and Hippo-Spatial. This happens because the initialization algorithm in GiST-Spatial is rather complex and requires a large number of temporary disk files to decide the boundries of the minimum bounding rectangles. Hence, the intensive disk I/O cost encumbers the initialization performance of GiST-Spatial.

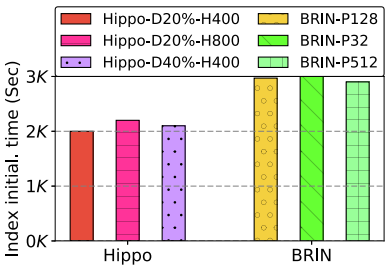
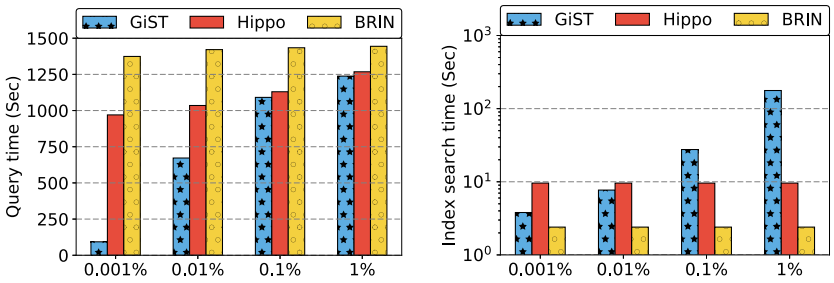


Fig. 6: Initialization time

Figure 6 depicts how a variety of parameters settings impact the initialization time of both BRIN-Spatial and Hippo-Spatial. Hippo-Spatial takes 30% less initialization time than BRIN-Spatial. That happens due to the fact that the index initialization algorithm makes use of a temporary in-memory data structure (denoted TmpEntry) to store the to-be-persisted index entry. For BRIN-Spatial and Hippo-Spatial, TmpEntry keeps summarizing new incoming tuples and updates MBR for BRIN-Spatial (partial histogram for Hippo-Spatial) if needed. This process continues until BRIN-Spatial reaches pages per range limit or Hippo-Spatial reaches the density limit. Then, TmpEntry will be serialized and persisted to disk. However, in most cases of Hippo-Spatial, the TmpEntry data structure is rarely updated because TmpEntry only notes distinct histogram buckets hit by the the scanned tuples. Unlike Hippo-Spatial, BRIN-Spatial initialization algorithm keeps updating the MBR as long as the newly summarized tuple not fully covered by the MBR. Such frequent TmpEntry updates lead to the gap in the initialization time.



(a) Query time (b) Index probe time (log. scale)

Fig. 7: Varying the spatial range query selectivity factor

5 Evaluating the query response time

This section studies the query execution performance time using each of the three considered indexing indexes. To identify the proper scenarios for different indexes, we define two metrics of spatial range query: spatial range query selectivity and query range area size. The categorized results are given in Figures 7 and 9.

5.1 Varying the spatial range query selectivity factor

This section studies the impact of varying the spatial range query selectivity factor on the query response time. The selectivity factor of a given spatial range query is calculated as the ratio of the total NYC taxi trips returned by running the spatial range query over the total number trips stored in the database. We vary the average spatial range query selectivity from 0.001%, 0.01%, 0.1% to 1%. To generate the query workload with average selectivity, we first create GiST-Spatial index on the pick-up/drop-off location and randomly select a set of query points from the table. Then, we use each query point to issue a K Nearest Neighbors (KNN) searches on the NYC taxi table. The number K refers to the number of tuples returned by 0.001% - 1% selectivity queries. For each KNN query, the returned K^{th} nearest neighbor and its mirror point against the query point represent a query range window that has the specified range selectivity. The generated spatial range queries are then used to run the experiments and the reported query execution time in Figure 7 represents the average time PostgreSQL took to run the query workload.

As shown in Figure 7, GiST-Spatial exhibits two orders of magnitude faster query execution performance than Hippo-Spatial and BRIN-Spatial on highly selective queries (0.001% selectivity factor). As the spatial range query selectivity factor becomes higher (lower selectivity), the query execution time gap between GiST-Spatial and Hippo-Spatial diminishes. For 0.1% and 1% selectivity factors, Hippo-Spatial is able to achieve similar query execution performance to that of GiST-Spatial. That happens due to the fact that, for highly selective

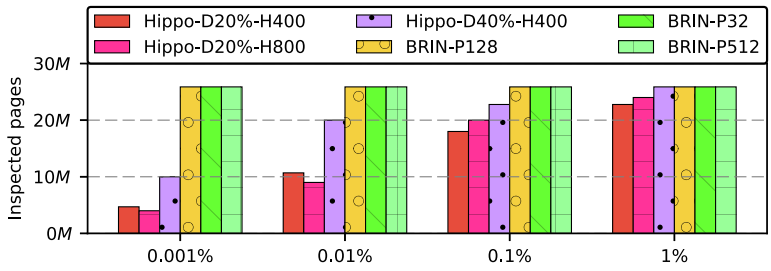
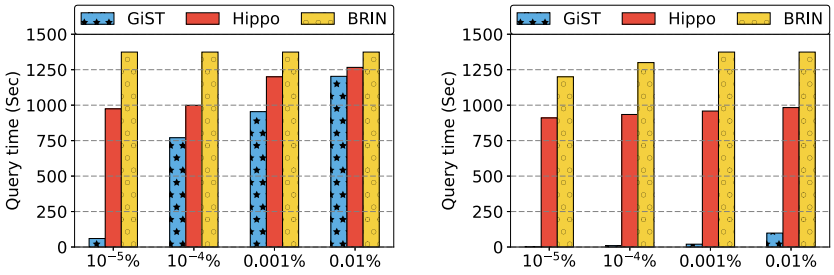


Fig. 8: Inspected data pages on different query selectivities

queries (e.g., 0.001% selectivity), GiST-Spatial's balanced tree structure is able to prune disjoint subtrees and retrieve only a small amount of qualified NYC taxi tuples to recheck. On the other hand, Hippo-Spatial still has much more possible qualified page to inspect. For less selective queries (selectivity factor 0.1% and 1%), GiST-Spatial also has to retrieve more tuples for further inspection and that is why it has similar performance to that of Hippo-Spatial. However, BRIN-Spatial exhibits the slowest query execution performance as compared to GiST-Spatial and Hippo-Spatial. The main reason is that all Minimum Bounding Rectangles store with each index entry in BRIN-Spatial span the entire New York City metropolitan area and BRIN-Spatial actually inspects almost all disk pages occupied by the NYC taxi table to process queries with different selecitvities.

Figure7b describes the index probe time on different selectivity factors. The index probe time refers in particular to the time these indexes spend on searching index entries when a query is issued. That excludes the time the database system takes to read the data pages. For GiST-Spatial, the index probe time stands for the time GiST-Spatial used to find all qualified tuple pointers. The upcoming GiST-Spatial refine and data page retrieval phase is taken care of by PostgreSQL. For BRIN-Spatial and Hippo-Spatial, the index probe time represents the time these indexes spend on traversing all index entries. It is obvious that the index probe time for BRIN-Spatial and Hippo-Spatial is constant for all spatial range selectivity factors. That happens due to the fact that BRIN-Spatial and Hippo-Spatial always scan all index entries. On the other hand, for higher selectivity factors, GiST-Spatial have to expand its probe range and go to lower tree levels. Figure 7b shows that the index probe time of GiST-Spatial, in fact, increases exponentially.

Figure 8 depicts the total number of inspected data pages using different index parameters. Both BRIN-Spatial and Hippo-Spatial need to inspect possible qualified pages for retrieving the truly qualified tuples. As given in Figure 8, Hippo-Spatial inspects less pages than BRIN-Spatial. To be precise, Hippo-Spatial with 20% density inspects up to 6 times less NYC taxi data pages on 0.001% and 0.01% selectivity factors and BRIN-Spatial inspects up to 40% more



(a) Queries in dense locations

(b) Queries in random locations

Fig. 9: Query time issued in different spatial areas

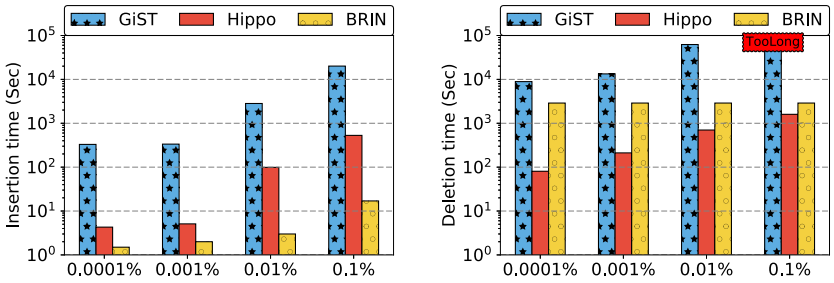
disk pages for queries with 0.1% and 1% selectivity factors. That happens because Hippo-Spatial is able to prune more data pages since it only inspects page ranges which have joint histogram buckets with the spatial query predicate. Hippo-Spatial with 40% density and Hippo-Spatial with 800 histogram buckets (i.e., Hippo-D40%-H800) experience slower query execution time as compare to Hippo-Spatial. The partial histograms of Hippo-D40%-H800 are too full and too many bits set to 1. That increases the probability that each index entry in Hippo-Spatial has joint buckets with the spatial query predicate. It is also worth noting that BRIN-Spatial in general (with various parameters setting) inspects the same number of data pages since it always inspects the entire table due to its data-agnostic nature.

5.2 Varying the spatial range area size

This section studies the impact of varying the size of the spatial range area. The range area represents the area covered by the issued spatial range query. In Section 5.1, we discussed the query response time for different query selectivity factors. However, users rarely issue spatial queries in strict accordance to the selectivity factor. Assume that a user observe the NYC taxi dataset on a web browser. The user usually searches dense areas. In fact, spatial data is alway highly skewed and sparse areas such as deserts are less interesting for analysts. We define two types of queries:

- random area spatial query (studied in Figure 9b): To generate such queries, we issue spatial range queries in random locations that lie within the New York City region.
- dense area spatial queries (studied in Figure 9a): To generate this workload, we limit the spatial queries to dense locations (e.g., Manhattan). A dense location contains a large number of Taxi trips. For instance, the hottest/densest data areas in New York Taxi dataset are Times Square, JFK airport and Laguardia airport.

Furthermore, we vary the range area size from 10^{-5}% to 0.01%. Larger range area such as 0.001% or 0.01% exposes the region of a city while smaller range



(a) Data insertion time

(b) Data deletion time

Fig. 10: Index maintenance performance on different data update percentage

area such 10^{-5}% exhibits the nearby businesses of our current location. Results are given in Figure 9. As it turns out in Figure 9b, GiST-Spatial achieves the best query execution performance for queries generated in random locations within NYC. That happens because spatial data is always skewed and most spatial range queries only return few tuples. On the contrary, in Figure 9a, GiST-Spatial takes much more time for queries issued in dense areas of NYC. That is due to the fact that the number of taxi trip records in the Manhattan (i.e., dense) area are far more than other areas in New York City. Moreover, Hippo-Spatial exhibits just a bit slower query execution performance than GiST-Spatial. BRIN-Spatial, on the other hand, exhibits the slowest query execution performance since it has to inspect a large fraction of data pages.

6 Studying the index maintenance overhead

This section studies the index maintenance overhead of all considered indexing schemes. We study the overhead incurred by two main index maintenance operations, i.e., insertion (see Figure 10a) and deletion time (see Figure 10b).

6.1 Insertion time

This section studies the time the database system takes to update the index when new taxi trip inserted in the NYC taxi table. Note that updating the index due to tuple insertion is deemed necessary to ensure the correctness of future queries. This section compares the three indexing schemes after inserting a certain amount of tuples in the NYC taxi table. We vary the number of inserted tuples as ratio of the original data size, i.e., 0.0001%, 0.001%, 0.01% and 0.1% tuples of the index NYC taxi table and insert them using the `COPY FROM` SQL clause.

As depicted in Figure 10a, GiST-Spatial exhibits the highest index maintenance overhead when new tuples are inserted. That happens because GiST-Spatial spends too much time on locating the proper tree node. Furthermore,

GiST-Spatial spends a non-ignorable amount of time on splitting the tree nodes to accommodate the newly inserted key. Frequent tree structure traverse and adjustments result in tremendous disk I/Os. Hippo-Spatial and BRIN-Spatial exhibit more than two orders of magnitude less maintenance overhead for insertion. That is due to the fact that both Hippo-Spatial and BRIN-Spatial possess a flat index structure which is relatively less complex than GiST-Spatialand hence easier to maintain. A newly inserted tuple leads to updating at most a single index entry. On the other hand, Hippo-Spatial takes more time time to insert a new tuple in contrast to BRIN-Spatial. That happens becuase Hippo-Spatial checks each new tuple against the complete histogram and updates the corresponding on-disk partial histogram if this new tuple hits a new distinct histogram bucket. On-disk updates happens more frequently in Hippo-Spatial since BRIN-Spatial only does physical entry updates when the new tuple is outside the corresponding MBR.

6.2 Deletion time

In this section, we evaluate the time PostgreSQL takes to maintain each of the three tested index structures in response to deleting a tuple(s) from the NYC taxi trip table. Similar to Section 6.1, we vary the percentage of deleted tuple to take 0.0001%, 0.001%, 0.01% and 0.1% values.

As shown in Figure 10b, Hippo-Spatial achieves close to two orders of magnitude better performance than GiST-Spatial) in handling the `DELETE` operation. For the sake of batch deletion, Hippo-Spatial re-summarizes an index entry that contain many deleted tuples in one go meanwhile GiST-Spatial searches for the affected tree nodes and sometimes merges the affected tree nodes in response to tuple deletion. On the other hand, BRIN-Spatial follows a naive lazy update strategy that rebuilds the entire index after a fixed number of tuples is deleted from the indexed table. That explains why Hippo-Spatial achieves close to an order of magnitude better performance BRIN-Spatial on low deletion percentages. The performance gap slightly decreases when a large percentage of the table is deleted because Hippo-Spatial has to re-summarize most index entries in that case, which is equivalent to re-building the whole index.

6.3 Hybrid workload performance

Figure 11 compares the performance of three indexes in hybrid query/update workloads. We generated five query / update workloads that vary the percentage of issued search operations as compared to the update operations, named after the percentage of search operations in the entire workload: 10%, 30%, 50%, 70% and 90%. Each workload consists of a thousand operations, which represent either index search or data update operations. In the experiments, we measure the system throughput achieved for each workload. The throughput is measured in terms of the number of operations per second. In each workload, the average spatial query selectivity factor is set to 0.01% while the average number of updated tuples is set to 0.01%

Table 1: Summery of Results

Metric	**GiST-Spatial**	**Hippo-Spatial**	**BRIN-Spatial**
Storage Overhead	84 GB	2 GB	10 MB
Initialization time	28 hours	30 minutes	45 minutes
Selectivity query	✓0.001% selectivity	✓selectivity between 0.01% and 1%	✗
Dense area query	✓10^{-5}% range query area	✓range query area $\geq 10^{-4}$%	✗
Index insertion	6 minutes for inserting 10^{-4}% data	4 seconds for inserting 10^{-4}% data	1 second for inserting 10^{-4}% data
Index deletion	2 hours for deleting 10^{-4}% data	2 min for deleting 10^{-4}% data	Index rebuilt
Hybrid workload	✓Query-intensive	✓Balanced Workload and Update-intensive	✓Update-intensive

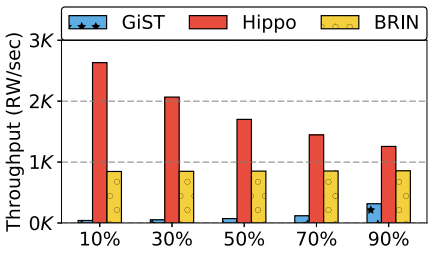


Fig. 11: Throughput

As it turns out in Figure 11, GiST-Spatial yields the lowest system throughput. That happens because GiST-Spatial spends too much time on index maintenance. BRIN-Spatial works faster than GiST-Spatial due to fast index maintenance although it incurs high latency when performing search operations. Hippo-Spatial consistently achieves the highest system throughput, as compared to BRIN-Spatial and GiST-Spatial. That is explained by the fact that Hippo-Spatial exhibits better index maintenance performance than GiST-Spatial and also exhibits a competitive query response time. Although Hippo-Spatial is outperformed by BRIN-Spatial when performing insertion operations, Hippo-Spatial still achieves higher throughput than BRIN-Spatial given its relatively superior query execution performance and fast data deletion operations. In summary, we can conclude that Hippo-Spatial and BRIN-Spatial are more suitable for update-involved workloads while Hippo-Spatial outperforms BRIN-Spatial due to better query response time and faster data deletion.

7 Summary of Results

We summarize the results of the experimental evaluation as follows (see Table 1):

– *Indexing overhead:* Indexing overhead includes two factors: index storage overhead and initialization time. Hippo-Spatial and BRIN-Spatial occupy

orders of magnitude smaller index size as compared to GiST-Spatial. In addition, the index initialization time taken by the system to create GiST-Spatial is two orders of magnitude higher than the others.
- *Query response time:* Hippo-Spatial is two orders of magnitude slower than GiST-Spatial on very highly selective queries (selectivity factor $\leq 0.001\%$) but still holds competitive query response time on queries with selectivity fator between 0.01% and 0.1%. Another observation is that GiST-Spatial executes order of magnitude faster performance when executing spatial range queries over very small area such as $10^{-5}\%$. Also, Hippo-Spatial achieves competitive query time on larger range query area such as $10^{-4}\%$ and 0.001%. BRIN-Spatial always exhibits a slow query execution performance for all query area sizes.
- *Index maintenance overhead:* The data insertion time taken by Hippo-Spatial is ten times more than the time take by BRIN-Spatial, yet still 10 times faster than GiST-Spatial on all update percentages (0.0001% to 0.1%). Hippo-Spatial deletion speed is more than an order of magnitude faster than GiST-Spatial and 2-10 times faster than BRIN-Spatial. In hybrid workloads, Hippo-Spatial achieves two orders of magnitude higher throughput than GiST-Spatial on update-intensive workloads (10%, 30% queries) while GiST-Spatial has higher throughput on query-intensive workloads.

8 Key Insights and Learned Lessons

Through extensive experiments, we presented a comprehensive analysis of classic and state-of-the-art spatial database indexing schemes supported in PostgreSQL, GiST-Spatial, Hippo-Spatial and BRIN-Spatial. Below, we share our key insights through the following learned lessons:

- ***Do not create GiST-Spatial (i.e., spatial tree index) when the database system is deployed on a storage device with high $ per GB.*** The storage overhead introduced by GiST-Spatial created over the NYC taxi dataset is 84 GB (see Table 1), which is close to 50% of the original data size. Note that the dollar cost increases dramatically when the DBMS is deployed on modern storage devices (e.g., SSD and Non-Volatile-Ram) since they are still more than an order of magnitude expensive than classic Hard Disk Drives (HDDs). As per Amazon.com and NewEgg.com, the dollar cost per storage unit for HDD and SSD are 0.04 and 1.4 $/GB, respectively. Instead, the user may consider Hippo-Spatial and BRIN-Spatial to reduce the overall storage cost since these indexes only occupy between 0.1 and 1 % as compared to the original dataset.
- ***Do not use BRIN-Spatial or Hippo-Spatial for Yelp-like applications.*** Applications like Yelp usually issue very highly selective spatial range queries that retrieve point-of-interests (e.g., 0.001% range query selectivity) and present them to the end-user. As per the experiments, GiST-Spatial is deemed a perfect indexing scheme for Yelp-like applications given its superior performance in executive highly selective spatial range queries (see

Table 1). Furthermore, spatial data (i.e., Point-of-Interests) in Yelp are not dense. That is due to the fact that every longitude and latitude location on the surface of the earth contains a few (usually one) Point-of-Interests (or buildings).

- ***Use Hippo-Spatial for spatial analytics applications over dynamic and dense spatial data.*** NASA constantly collects Earth science data (e.g., weather, pollution, socioeconomic data) [7]. Earth science data is quite dense and new data is inserted into the system on a daily basis. Furthermore, since geospatial data in such applications is typically consumed as aggregate visualizations (e.g., Heatmap, Cartogram), spatial range queries on such data are not quite selective (selectivity factor between 0.1% and 1%) as in Yelp-like applications. Having said that, Hippo-Spatial is deemed the perfect for such data given: (1) its small storage footprint and low maintenance overhead compared to GiST-Spatial and (2) its superior query execution performance over selective queries and higher throughput compared to BRIN-Spatial.

References

1. New york city taxi and limousine commission. http://www.nyc.gov/html/tlc/html/about/trip_record_data.html.
2. Paul M Aoki. Generalizing" search" in generalized search trees. In *Data Engineering, 1998. Proceedings., 14th International Conference on*, pages 380–389. IEEE, 1998.
3. Block range index. https://www.postgresql.org/docs/9.6/static/brin.html.
4. Postgis - spatial and geographic objects for postgresql. http://postgis.net.
5. Postgresql: a powerful, open source object-relational database system. https://www.postgresql.org/.
6. Charles Bontempo and George Zagelow. The ibm data warehouse architecture. *The Communications of the ACM*, 41(9):38–48, 1998.
7. Earth science data. https://earthdata.nasa.gov.
8. Douglas Comer. Ubiquitous b-tree. *ACM Computing Surveys, CSUR*, 11(2):121–137, 1979.
9. Antonio Corral, Michael Vassilakopoulos, and Yannis Manolopoulos. Algorithms for Joining R-Trees and Linear Region Quadtrees. In *Proceedings of the International Symposium on Advances in Spatial Databases, SSD*, pages 251–269, 1999.
10. Raphael A. Finkel and Jon Louis Bentley. Quad trees: A Data Structure for Retrieval of Composite Keys. *Acta Informatica*, 4(1):1–9, 1974.
11. Francesco Fusco, Marc Ph Stoecklin, and Michail Vlachos. Net-fli: on-the-fly compression, archiving and indexing of streaming network traffic. *The VLDB Journal*, 3(1-2):1382–1393, 2010.
12. Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. Compressing relations and indexes. In *Proceedings of the International Conference on Data Engineering, ICDE*, pages 370–379. IEEE, 1998.
13. Antonin Guttman. R-trees: a dynamic index structure for spatial searching. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, pages 47–57. ACM, 1984.
14. Joseph M Hellerstein. Generalized search tree. In *Encyclopedia of Database Systems*, pages 1222–1224. Springer, 2009.

15. Joseph M Hellerstein, Jeffrey F Naughton, and Avi Pfeffer. *Generalized search trees for database systems*. September, 1995.
16. Ibrahim Kamel and Christos Faloutsos. Hilbert R-tree: An Improved R-tree Using Fractals. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, September 1994.
17. Ibrahim Kamel, M. Khalil, and V. Kouramajian. Bulk Insertion in Dynamic R-Trees. In *Proc. of the Intl. Symp. on Spatial Data Handling, SDH*, pages 31–42, 1996.
18. Marcel Kornacker, C Mohan, and Joseph M Hellerstein. Concurrency and recovery in generalized search trees. In *ACM SIGMOD Record*, volume 26, pages 62–72. ACM, 1997.
19. Mong-Li Lee, Wynne Hsu, Christian S. Jensen, Bin Cui, and Keng Lik Teo. Supporting Frequent Updates in R-Trees: A Bottom-Up Approach. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, pages 608–619, September 2003.
20. Hanan Samet and Robert E. Webber. Storing a Collection of Polygons using Quadtrees. *ACM Transactions on Graphics, TOG*, 4(3):182–222, 1985.
21. Lefteris Sidirourgos and Martin L. Kersten. Column imprints: a secondary index structure. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, pages 893–904. ACM, 2013.
22. Dominik Ślezak and Victoria Eastwood. Data warehouse technology by infobright. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, pages 841–846. ACM, 2009.
23. Michael Stonebraker and Lawrence A Rowe. The design of postgres. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, pages 340–355. ACM, 1986.
24. Jamel Tayeb, Özgür Ulusoy, and Ouri Wolfson. A Quadtree-Based Dynamic Attribute Indexing Method. *The Computer Journal*, 41(3):185–200, 1998.
25. Ronald Weiss. A technical overview of the oracle exadata database machine and exadata storage server. *Oracle White Paper. Oracle Corporation, Redwood Shores*, 2012.
26. X. Xu, Jiawei Han, and W. Lu. RT-Tree: An Improved R-Tree Indexing Structure for Temporal Spatial Databases. In *Proceeding of the International Symposium on Spatial Data Handling, SSDH*, pages 1040–1049, July 1990.
27. Jia Yu, Raha Moraffah, and Mohamed Sarwat. Hippo in action: Scalable indexing of a billion new york city taxi trips and beyond. In *Proceedings of the International Conference on Data Engineering, ICDE*, page To appear. IEEE, 2017.
28. Jia Yu and Mohamed Sarwat. Two birds, one stone: a fast, yet lightweight, indexing scheme for modern database systems. *Proceedings of the VLDB Endowment*, 10(4):385–396, 2016.
29. Marcin Zukowski, Sandor Heman, Niels Nes, and Peter Boncz. Super-scalar ram-cpu cache compression. In *Proceedings of the International Conference on Data Engineering, ICDE*, pages 59–59. IEEE, 2006.