Two Birds, One Stone: A Fast, yet Lightweight, Indexing Scheme for Modern Database Systems

[1]Jia Yu [2]Mohamed Sarwat

School of Computing, Informatics, and Decision Systems Engineering
Arizona State University, 699 S. Mill Avenue, Tempe, AZ
[1]jiayu2@asu.edu, [2]msarwat@asu.edu

ABSTRACT

Classic database indexes (e.g., B$^+$-Tree), though speed up queries, suffer from two main drawbacks: (1) An index usually yields 5% to 15% additional storage overhead which results in non-ignorable dollar cost in big data scenarios especially when deployed on modern storage devices. (2) Maintaining an index incurs high latency because the DBMS has to locate and update those index pages affected by the underlying table changes. This paper proposes Hippo a fast, yet scalable, database indexing approach. It significantly shrinks the index storage and mitigates maintenance overhead without compromising much on the query execution performance. Hippo stores disk page ranges instead of tuple pointers in the indexed table to reduce the storage space occupied by the index. It maintains simplified histograms that represent the data distribution and adopts a page grouping technique that groups contiguous pages into page ranges based on the similarity of their index key attribute distributions. When a query is issued, Hippo leverages the page ranges and histogram-based page summaries to recognize those pages such that their tuples are guaranteed not to satisfy the query predicates and inspects the remaining pages. Experiments based on real and synthetic datasets show that Hippo occupies up to two orders of magnitude less storage space than that of the B$^+$-Tree while still achieving comparable query execution performance to that of the B$^+$-Tree for 0.1% - 1% selectivity factors. Also, the experiments show that Hippo outperforms BRIN (Block Range Index) in executing queries with various selectivity factors. Furthermore, Hippo achieves up to three orders of magnitude less maintenance overhead and up to an order of magnitude higher throughput (for hybrid query/update workloads) than its counterparts.

1. INTRODUCTION

A database system (DBMS) often employs an index structure, e.g., B$^+$-Tree, to speed up queries issued on the indexed table. Even though classic database indexes improve the query response time, they face the following challenges:

- **Indexing Overhead:** A database index usually yields 5% to 15% additional storage overhead. Although the overhead may not seem too high in small databases, it results in non-ignorable dollar cost in big data scenarios. Table 1a depicts the storage overhead of a B$^+$-Tree created on the Lineitem table from the TPC-H benchmark [6] (database size varies from 2, 20 and 200 GB). Moreover, the dollar cost increases dramatically when the DBMS is deployed on modern storage devices (e.g., Solid State Drives and Non-Volatile Memory) because they are still more than an order of magnitude expensive than Hard Disk Drives (HDDs). Table 1b lists the dollar cost per storage unit collected from Amazon.com and NewEgg.com. In addition, initializing an index may be a time consuming process especially when the index is created on a large table (see Table 1a).

- **Maintenance Overhead:** A DBMS must update the index after inserting (deleting) tuples into (from) the underlying table. Maintaining a database index incurs high latency because the DBMS has to locate and update those index entries affected by the underlying table changes. For instance, maintaining a B$^+$-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 big data scenarios. Table 1a shows the B+ Tree insertion overhead (insert 0.1% records) for the TPC-H Lineitem table.

Table 1: Index overhead and storage dollar cost

(a) B$^+$-Tree overhead

TPC-H	Index size	Initialization time	Insertion time
2 GB	0.25 GB	30 sec	10 sec
20 GB	2.51 GB	500 sec	1180 sec
200 GB	25 GB	8000 sec	42000 sec

(b) Storage dollar cost

HDD	EnterpriseHDD	SSD	EnterpriseSSD
0.04 $/GB	0.1 $/GB	0.5 $/GB	1.4 $/GB

Proceedings of the VLDB Endowment, Vol. 10, No. 4

Existing approaches that tackle one or more of the aforementioned challenges are classified as follows: *(1) Compressed indexes:* Compressed B$^+$-Tree approaches [8, 9, 21] reduce the storage overhead but compromise on the query performance due to the additional compression and decompression time. Compressed bitmap indexes also reduce index storage overhead [10, 12, 16] but they mainly suit low cardinality attributes which are quite rare. For high cardinality attributes, the storage overhead of compressed bitmap indexes significantly increases [19]. *(2) Approximate indexes:* An approximate index [4, 11, 14] trades query accuracy for storage to produce smaller, yet fast, index structures. Even though approximate indexes may shrink the storage size, users cannot rely on their un-guaranteed query accuracy in many accuracy-sensitive application scenarios like banking systems or user archive systems. *(3) Sparse indexes:* A sparse index [5, 13, 17, 18] only stores pointers which refer to disk pages and value ranges (min and max values) in each page so that it can save indexing and maintenance overhead. It is generally built on ordered attributes. For a posed query, it finds value ranges which cover or overlap the query predicate and then rapidly inspects the associated few parent table pages one by one for retrieving truly qualified tuples. However, for unordered attributes which are much more common, sparse indexes compromise too much on query performance because they find numerous qualified value ranges and have to inspect a large number of pages.

This paper proposes Hippo a fast, yet scalable, sparse database indexing approach. In contrast to existing tree index structures, Hippo stores disk page ranges (each works as a pointer of one or many pages) instead of tuple pointers in the indexed table to reduce the storage space occupied by the index. Unlike existing approximate indexing methods, Hippo guarantees the query result accuracy by inspecting possible qualified pages and only emitting those tuples that satisfy the query predicate. As opposed to existing sparse indexes, Hippo maintains simplified histograms that represent the data distribution for pages no matter how skew it is, as the summaries for these pages in each page range. Since Hippo relies on histograms already created and maintained by almost every existing DBMS (e.g., PostgreSQL), the system does not exhibit a major additional overhead to create the index. Hippo also adopts a page grouping technique that groups contiguous pages into page ranges based on the similarity of their index key attribute distributions. When a query is issued on the indexed database table, Hippo leverages the page ranges and histogram-based page summaries to recognize those pages for which the internal tuples are guaranteed not to satisfy the query predicates and inspects the remaining pages. Thus Hippo achieves competitive performance on common range queries without compromising the accuracy. For data insertion and deletion, Hippo dispenses with the numerous disk operations by rapidly locating the affected index entries. Hippo also adaptively decides whether to adopt an eager or lazy index maintenance strategy to mitigate the maintenance overhead while ensuring future queries are answered correctly.

We implemented a prototype of Hippo inside PostgreSQL 9.5[1]. Experiments based on the TPC-H benchmark as well as real and synthetic datasets show that Hippo occupies up to two orders of magnitude less storage space than that of the B$^+$-Tree while still achieving comparable query execution performance to that of the B$^+$-Tree for 0.1% - 1% selectivity factors. Also, the experiments show that Hippo outperforms BRIN, though occupies more storage space, in executing queries with various selectivity factors. Furthermore, Hippo achieves up to three orders of magnitude less maintenance overhead than its counterparts, i.e., B$^+$-Tree and BRIN. Most importantly, Hippo exhibits up to an order of magnitude higher throughput (measured in terms of the number of tuples processed per second) than both BRIN and B$^+$-Tree for hybrid query/update workloads.

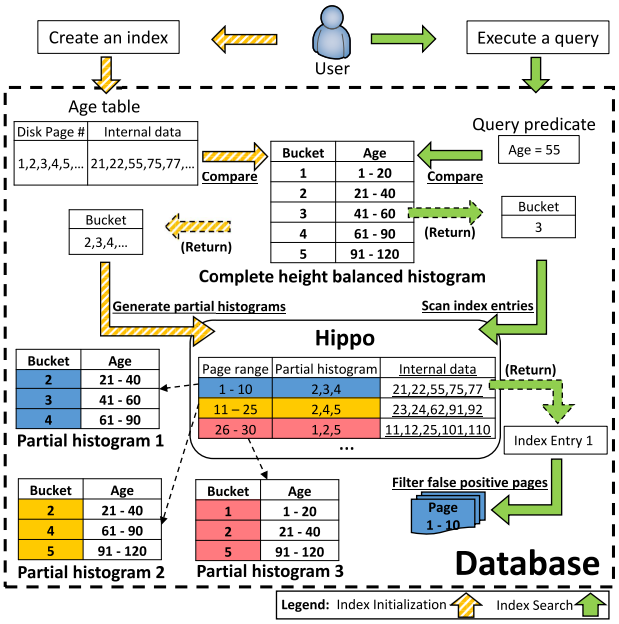


Figure 1: Initialize and search Hippo on age table

The rest of the paper is structured as follows: In Section 2, we explain the key idea behind Hippo and describe the index structure. We explain how Hippo searches the index, builds the index from scratch, and maintains it efficiently in Sections 3, 4 and 5. In Section 6, we deduce a cost model for Hippo. Extensive experimental evaluation is presented in Section 7. We summarize a variety of related indexing approaches in Section 8. Finally, Section 9 concludes the paper and highlights possible future directions.

2. HIPPO OVERVIEW

This section gives an overview of Hippo. Figure 1 depicts a running example that describes the index initialization (left part of the figure) and search (right part of the figure) processes in Hippo. The main challenges of designing an index are to reduce the indexing overhead in terms of storage and initialization time as well as speed up the index maintenance while still keeping competitive query performance. To achieve that, an index should possess the following two main properties: *(1) Less Index Entries:* For better storage space utilization, an index should determine and only store the most representative index entries that summarize the key attribute. Keeping too many index entries inevitably results in high storage overhead as well as high initialization time. *(2) Index Entries Independence:* The index entries

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

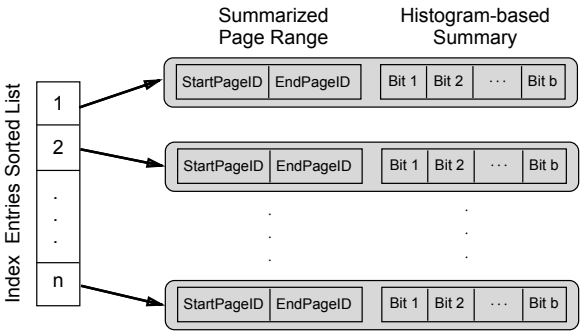


Figure 2: Hippo Index Structure

should be independent from each other. In other words, the range of values that each index entry represents should have minimal overlap with other index entries. Interdependence among index entries, like that in a B^+-Tree, results in overlapped tree nodes. That may lead to more I/O operations during query processing and several cascaded updates during index maintenance.

Data Structure. Figure 2 depicts the index structure. To create an index, Hippo scans the indexed table and generates histogram-based summaries for a set of disk page based on the index key attribute. These summaries are then stored by Hippo along with page ranges they summarize. As shown in Figure 2, Hippo consists of n index entries such that each entry consists of the following two components:

- **Summarized Page Range:** represents the IDs of the first and last pages summarized (i.e., StartPageID and EndPageID in Figure 2) by the index entry. The DBMS can load particular pages into buffer according to their IDs. Hippo summarizes more than one physically contiguous pages to reduce the overall index size, e.g., Page 1 - 10, 11 - 25, 26 - 30 in Figure 1. The number of summarized pages in each index entry varies. Hippo adopts a page grouping technique that groups contiguous pages into page ranges based on the similarity of their index attribute distributions, using the partial histogram density (explained in Section 4).

- **Histogram-based Summary:** A bitmap that represents a subset of the complete height balanced histogram buckets (maintained by the underlying DBMS), aka. partial histogram. Each bucket, if exists, indicates that at least one of the tuples of this bucket exists in the summarized pages. Each partial histogram represents the distribution of the data in the summarized contiguous pages. Since each bucket of a height balanced histogram roughly contains the same number of tuples, each of them has the same probability to be hit by a random tuple from the table. Hippo leverages this feature to handle a variety of data distributions, e.g., uniform, skewed. To reduce the storage footprint, only bucket IDs are kept in partial histograms and partial histograms are stored in a compressed bitmap format. For instance, the partial histogram of the first index entry in Figure 1 is 01110.

Main idea. Hippo solves the aforementioned challenges as follows: (1) Each index entry summarizes many pages and only stores two page IDs and a compressed bitmap.(2) Each page of the parent table is only summarized by one Hippo index entry. Hence, any updates that occur in a certain page only affect a single independent index entry. Finally, during a query, pages whose partial histograms do not have desired buckets are guaranteed not to satisfy certain query predicates and marked as false positives. Thus Hippo only inspects other pages that probably satisfies the query predicate and achieves competitive query performance.

Algorithm 1: Hippo index search

```
   Data: A given query predicate Q
   Result: A set of qualified tuples R
1  // Step I: Scanning Index Entries;
2  Set of Possible Qualified Pages P = φ;
3  foreach Index Entry in Hippo do
4      if the partial histogram has joint buckets with Q then
5          Add the IDs of pages indexed by the entry to P;
6      end
7  end
8  // Step II: Filtering False Positive Pages;
9  Set of Qualified Tuples R = φ;
10 foreach Page ID ∈ P do
11     Retrieve the corresponding page p;
12     foreach tuple t ∈ p do
13         if t satisfies the query predicate then
14             Add t to R;
15         end
16     end
17 end
18 return R;
```

3. INDEX SEARCH

The search algorithm takes as input a query predicate and returns a set of qualified tuples. As explained in Section 2, partial histograms are stored in a bitmap format. Hence, any query predicates for a particular attribute are broken down into atomic units: equality query predicate and range query predicate. Each unit predicate is compared with the buckets of the complete height balanced histogram (discussed in Section 4). A bucket is hit by a predicate if the predicate fully contains, overlaps, or is fully contained by the bucket. Each unit predicate can hit at least one or more buckets. Afterwards, the query predicate is converted to a bitmap. Each bit in this bitmap reflects whether the bucket that has the corresponding ID is hit (1) or not (0). Thus, the corresponding bits of all hit buckets are set to 1.

The search algorithm then runs in two main steps (see pseudo code in Algorithm 1): (1) Step I: Scanning Hippo index entries and (2) Step II: Filtering false positive pages. The search process leverages the index structure to avoid unnecessary page inspection so that Hippo can achieve competitive query performance.

3.1 Step I: Scanning index entries

Step I finds possible qualified disk pages, which may contain tuples that satisfy the query predicate. Since it is quite possible that some pages may not contain any qualified tuple especially for highly selective queries, Hippo prunes these index entries (that index these unqualified pages) that definitely do not contain any qualified pages.

In this step, the search algorithm scans the Hippo index. For each index entry, the algorithm retrieves the partial his-

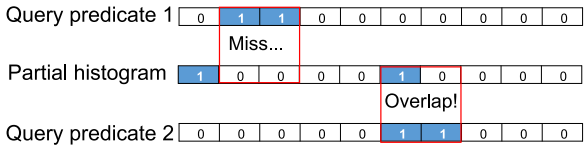


Figure 3: Scan index entries

togram which summarizes the data distribution in the pages indexed by such entry. The algorithm then checks whether the input query predicate has one or more joint (i.e. overlapped) buckets with the partial histogram. To efficiently process that, Hippo performs a nested loop between each partial histogram and the input query predicate to find the joint buckets. Since both the partial histograms and the query predicate are in a bitmap format, Hippo accelerates the nested loop by performing a bitwise 'AND' of the bytes from both sides, *aka. bit-level parallelism.* In case bitwise 'AND'ing the two bytes returns 0, that means there exist no joint buckets between the query predicate and the partial histogram. Entries with partial histograms that do not contain the hit buckets (i.e., the corresponding bits are 0) are guaranteed not to contain any qualified disk pages. Hence, Hippo disqualifies these pages and excludes them from further processing. On the other hand, index entries with partial histograms that contain at least one of the hit buckets, i.e., the corresponding bits are 1, may or may not have qualified pages. Hippo deems these pages as possible qualified pages and hence forwards their IDs to the next step.

Figure 3 visualizes the procedure of scanning index entries according to their partial histograms. In Figure 3, buckets hit by the query predicates and the partial histogram are represented in a bitmap format. According to this figure, the partial histogram misses a query predicate if the highlighted area of the predicate falls into the blank area of the partial histogram, whereas a partial histogram is selected if the predicate does not fall completely into the blank area of the histogram.

3.2 Step II: Filtering false positive pages

The previous step identifies many unqualified disk pages that are guaranteed not to satisfy the query predicate. However, not all unqualified pages can be detected by the previous step. The set of possible qualified pages, retrieved from Step I, may still contain false positives (defined below). During the search process, Hippo considers a possible qualified page p a false positive if and only if (1) p lies in the page range summarized by a qualified index entry from Step I and (2) p does not contain any tuple that satisfies the input query predicate. To filter out false positive pages, Step II inspects every tuple in each possible qualified page, retrieves those tuples that satisfy the query predicate, and finally returns those tuples as the answer.

Step II takes as input the set of possible qualified pages IDs, formatted in a separate bitmap. Each bit in this bitmap is mapped to the page at the same position in the original table indexed by Hippo. For each page ID, Hippo retrieves the corresponding page from disk and checks each tuple in that page against the query predicate. In case, a tuple satisfies the query predicate, the algorithm adds this tuple to the final result set. The right part of Figure 1 describes how to search the index using an input query predicate. First, Hippo finds that query predicate age = 55 hits bucket 3. Since the first one of the three partial histograms nicely contains bucket 3, only the disk pages 1 - 10 are selected as possible qualified pages and hence sent for further inspection in step II. It is also worth noting that these partial histograms summarize different number of pages.

4. INDEX INITIALIZATION

To create an index, Hippo takes as input a database table and the key attribute (i.e., column) in this table. Hippo then performs two main steps (See pseudo code in Algorithm 2) to initialize itself: (1) Generate partial histograms (Section 4.1), and (2) Group similar pages into page ranges (Section 4.2), described as follows.

4.1 Generate partial histograms

To initialize the index, Hippo leverages a complete height balanced histogram, maintained by most DBMSs, that represents the data distribution. A histogram consists of a set of buckets such that each bucket represents the count of tuples with attribute value within the bucket range. A partial histogram only contains a subset of the buckets that belongs to the height balanced histogram. The resolution of the complete histogram (H) is defined as the total number of buckets that belongs to this histogram. A histogram will obviously have larger physical storage size if it has higher resolution. The histogram resolution also affects the query response time (see Section 6 for further details).

Hippo stores a partial histogram for each index entry to represent the data distribution of tuples in one or many disk pages summarized by the entry. Partial histograms allow Hippo to early identify unqualified disk pages and avoid unnecessary page inspection. To generate partial histograms, Hippo scans all disk pages of the indexed table. For each page, the algorithm retrieves each tuple, the key attribute value is extracted from each tuple and then compared to the complete histogram using binary search. Buckets hit by tuples are kept for this page and then compose a partial histogram. A partial histogram only contains distinct buckets. For instance, there is a group of age attribute values like the first entry of Hippo given in Figure 1: 21, 22, 55, 75, 77. Bucket 2 is hit by 21 and 22, bucket 3 is hit by 55 and bucket 4 is hit by 77 (see partial histogram 1 in Figure 1).

Hippo shrinks the storage footprint of partial histograms by dropping all bucket value ranges and only keeping bucket IDs. Actually, as mentioned in Section 2, dropping value range information does not have much negative impact on the index search. To further shrink the storage footprint, Hippo stores the histogram bucks IDs in bitmap type format instead of using an integer type (4 bytes or more). Each partial histogram is stored as a bitmap such that each bit represents a bucket at the same position in a complete histogram. Bit value 1 means the associated bucket is hit and stored in this partial histogram while 0 means the associated bucket is not included. The partial histogram can also be compressed by any existing bitmap compression technique. The time for compressing and decompressing partial histograms is ignorable compared to that of inspecting possible qualified pages.

4.2 Group pages into page ranges

Generating a partial histogram for each disk page may lead to very high storage overhead. Grouping contiguous

Algorithm 2: Hippo index initialization

```
Data: Pages of a parent table
Result: Hippo index
Create a working partial histogram (in bitmap format);
Set StartPage = 1 and EndPage = 1;
foreach page do
    Find distinct buckets hit by its tuples;
    Set associated bits to 1 in the partial histogram;
    if the working partial histogram density > threshold
     then
        Store the partial histogram and the page range
          (StartPage and EndPage) as an index entry;
        Create a new working partial histogram;
        StartPage = EndPage + 1;
        EndPage = StartPage;
    else
        EndPage = EndPage + 1;
    end
end
```

Algorithm 3: Update Hippo for data insertion

```
Data: A newly inserted tuple that belongs to Page a
Result: Updated Hippo
Find the bucket hit by the inserted tuple;
Locate a Hippo index entry which summarizes Page a;
if an index entry is located
 then
    Fetch the located Hippo index entry;
    Update the retrieved entry if necessary;
else
    Retrieve the entry that summarizes the last page;
    if the partial histogram density < threshold then
        Summarize Page a into the retrieved index entry;
    else
        Summarize Page a into a new index entry;
    end
end
```

pages and merging their partial histograms into a larger partial histogram (in other words, summarizing more pages within one partial histogram) can tremendously reduce the storage overhead. However, that does not mean that all pages should be grouped together and summarized by a single merged partial histogram. The more pages are summarized, the more buckets the partial histogram contains. If the partial histogram becomes a complete histogram and covers any possible query predicates, it is unable to filter the false positives and the disk pages summarized by this partial histogram will be always treated as possible qualified pages.

One strategy is to group a fixed number of contiguous pages per partial histogram. Yet, this strategy is not efficient when a set of contiguous pages have much more similar data distribution than other areas. To remedy that, Hippo dynamically groups more contiguous pages under the same index entry when they possess similar data distribution and less contiguous pages if they do not show similar data distribution. To take the page grouping decision, Hippo leverages a parameter called *partial histogram density*. The density of a partial histogram is defined as the ratio of complete histogram buckets that belongs to the partial histogram. Obviously, the complete histogram has a density value of 1. The definition can be formalized as follows:

$$\textit{Partial histogram density } (D) \;=\; \frac{\#\ \textit{Buckets}_{\textit{partial histogram}}}{\#\ \textit{Buckets}_{\textit{complete histogram}}}$$

The density exhibits an important phenomenon that, for a set of contiguous disk pages, their merged partial histogram density will be very low if these pages are very similar, and vice versa. Therefore, a partial histogram with a certain density may summarize more pages if these contiguous pages have similar data, vice versa. Making use of this phenomenon enables Hippo to dynamically group pages and merge partial histograms into one. In addition, it is understandable that a lower density partial histogram (summarizes less pages) has the high probability to be excluded from further processing.

Users can easily set the same density value for all partial histograms as a threshold. Hippo can automatically decide how many pages each partial histogram should summarize. Algorithm 2 depicts how Hippo initializes the index and summarizes more pages within a partial histogram by means of the partial histogram density. The basic idea is that new pages will not be summarized into a partial histogram if its density is larger than the threshold and a new partial histogram will be created for the following pages.

The left part of Figure 1 depicts how the initialization process for an index create on the age attribute. In the example, the partial histogram density is set to 0.6. All tuples are compared with the complete histogram and IDs of distinct buckets hit by all tuples are generated as partial histograms along with their page range. So far, as Figure 1 and 2 shows, each index entry has the following parameters: a partial histogram in compressed bitmap format and two integers that stand for the first and last pages summarized by this histogram (summarized page range). Each entry is then serialized and stored on disk.

5. INDEX MAINTENANCE

Inserting (deleting) tuples into (from) the table requires maintaining the index. That is necessary to ensure that the DBMS can retrieve the correct set of tuples that match the query predicate. However, the overhead introduced by frequently maintaining the index may preclude system scalability. This section explains how Hippo handles updates.

5.1 Data insertion

Hippo adopts an eager update strategy when a new tuple is inserted to the indexed table. An eager strategy instantly updates or checks the index at least when a new tuple is inserted. Otherwise, all subsequent queries might miss the newly inserted tuple. Data insertion may change the physical structure of a table (i.e., heap file). The new tuple may belong to any pages of the indexed table. The insertion procedure (See Algorithm 3) performs the following steps: (I) Locate the affected index entry, and (II) Update the index entry if necessary.

Step I: Locate the affected index entry: After retrieving the complete histogram, the algorithm checks whether a newly inserted tuple hits one or more of the histogram buckets. The newly inserted tuple belongs to a disk page. This page may be a new page has not been summarized by any partial histograms before or an old page which has been summarized. However, because the numbers of pages summarized by each histogram are different, searching Hippo index entries to find the one contains this

target page is inevitable. From the perspective of disk storage, in a Hippo, all partial histograms are stored on disk in a serialized format. It will be extremely time-consuming if every entry is retrieved from disk, de-serialized and checked against the target page. The algorithm then searches the index entries by means of the index entries sorted list explained in Section 5.3.

Step II: Update the index entry: In case the inserted tuple belongs to a new page and the partial histogram density which summarizes the last disk page is smaller than the density threshold set by the system user, the algorithm summarizes the new page into this partial histogram in the last index entry. Otherwise, the algorithm creates a new partial histogram to summarize this page and stores them in a new index entry. In case a new tuple belongs to a page that is already summarized by Hippo, the partial histogram in the associated index entry will be updated if the inserted tuple hits a new bucket.

It is worth noting that: (1) Since the compressed bitmaps of partial histograms may have different size, the updated index entry may not fit the space left at the old location. Thus the updated one may be put at the end of Hippo. (2) After some changes (replacing old or creating new index entry) in Hippo, the corresponding position of the sorted list might need to be updated.

The I/O cost incurred by eagerly updating the index due to a newly inserted tuple is equal to *log(# of index entries) + 4*. Locating the affected index entry yields *log(# of index entries)* I/Os, whereas Step II consumes 4 I/Os to update the index entry. Section 6 gives more details on how to estimate the number of index entries in Hippo.

5.2 Data deletion

The eager update strategy is deemed necessary for data insertion to ensure the correctness of future queries issued on the indexed table. However, the eager update strategy is not necessary after deleting data from the table. That is due to the fact that Hippo inspects possible qualified pages during the index search process and pages with qualified deleted tuples might be still marked as possible qualified page in the first phase of the search algorithm. Even if these pages contain deleted tuples, such pages will be discarded during the "Step II: filtering false positive pages" phase of the search algorithm. However, not maintaining the index at all may introduce a lot of false positives during the search process, which may takes its toll on the query repossess time.

Hippo still ensures the correctness of queries even if it does not update the index at all after deleting tuples from a table. To achieve that, Hippo adopts a periodic lazy update strategy for data deletion. The deletion strategy maintains the index after a bulk of delete operations are performed to the indexed table. In such case, Hippo traverses all index entries. For each index entry, the system inspects the header of each summarized page for seeking notes made by DBMSs (e.g., PostgreSQL makes notes in page headers if data is removed from pages). Hippo re-summarizes the entire index entry instantly within the original page range if data deletion on one page is detected. The re-summarization follows the same steps in Section 4.

5.3 Index Entries Sorted List

When a new tuple is inserted, Hippo executes a fast binary search (according to the page IDs) to locate the affected index entry and then updates it. Since the index entries are not guaranteed to be sorted based on the page IDs (noted in data insertion section), an auxiliary structure for recording the sorted order is introduced to Hippo.

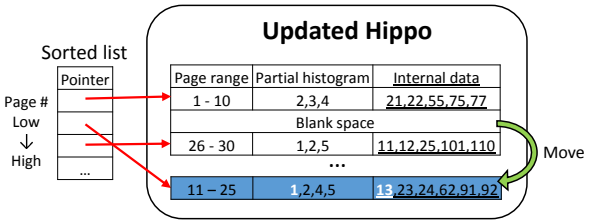


Figure 4: Hippo Index Entries Sorted List

The sorted list is initialized after all steps in Section 4 with the original order of index entries and put at the first several index pages of Hippo. During the entire Hippo life time, the sorted list maintains a list of pointers of Hippo index entries in the ascending order of page IDs. Actually each pointer represents the fixed size physical address of an index entry and these addresses can be used to retrieve index entries directly. That way, the premise of a binary search has been satisfied. Figure 4 depicts the Hippo index entries sorted list. Index entry 2 in Figure 1 has a new bucket ID 1 due to a newly inserted tuple in its internal data and hence this entry becomes the last index entry in Figure 4. The sorted list is still able to record the ascending order and help Hippo to perform a binary search on the index entries. In addition, such sorted list leads to slight additional maintenance overhead: Some index updates need to modify the affected pointers in the sorted list to reflect the new physical addresses.

6. COST MODEL

This section deduces a cost model for Hippo. Table 2 summarizes the main notations. Given a database table R with a number of tuples $Card$ and average number of tuples per disk page $pageCard$, a user may create a Hippo index on attribute (i.e., column) a_i of R. Let the complete histogram resolution be H (it has H buckets in total) and the partial histogram density be D. Assume that each Hippo index entry on average summarizes P data pages and T tuples. Queries executed against the index have an average selectivity factor SF. To calculate the query I/O cost, we need to consider: (1) $\text{I/O}_{scanning\ index}$ represents the cost of scanning the index entries (Phase I in the search algorithm) and (2) $\text{I/O}_{filtering\ false\ positives}$ represents the I/O cost of filtering false positive pages (Phase II).

Estimating the number of index entries. Since all index entries are scanned in the first phase, the I/O cost of this phase is equal to the total pages the index spans on disk ($\text{I/O}_{scanning\ index}=\frac{\#\ of\ index\ entries}{pageCard}$). To estimate the number of Hippo index entries, we have to estimate how many disk pages (P) are summarized by a partial histogram in general, or how many tuples (T) are checked against the complete histogram to generate a partial histogram. This problem is very similar to the Coupon Collector's Problem[7]. This problem can be described like that: "A vending machine sells H types of coupons (a complete histogram with H buckets). Alice is purchasing coupons from this machine. Each time (each tuple) she can get a random type

Table 2: Notations used in Cost Estimation

Term	Definition
H	Complete histogram resolution which means the number of buckets in this complete histogram
pageH	Average number of histogram buckets hit by a page
D	Partial histogram density, which is an user supplied parameter
P	Average number of pages summarized by a partial histogram for a certain attribute
T	Average number of tuples summarized by a partial histogram for a certain attribute
Card	Total number of tuples of the indexed table
pageCard	Average number of tuples per page
SF	The selectivity factor of the issued query

coupon (a bucket) but she might already have a same one. Alice keeps purchasing until she gets $D*H$ types of coupons (distinct buckets). How many times (T) does she need to purchase?" Therefore, the expectation of T is determined by the following equation:

$$T = H\times(\frac{1}{H} + \frac{1}{H-1} + ... + \frac{1}{H-D\times H+1}) \quad (1)$$

$$= H\times \sum_{i=0}^{D\times H-1} \frac{1}{H-i} \quad (2)$$

Note that the partial histogram density $D \in [\frac{pageH}{H}, 1]$. That means the global density should be larger than the ratio of average hit histogram buckets per page to all histogram buckets because page is the minimum unit when grouping pages based on density. Estimating $pageH$ is also a variant of Coupon Collector's Problem: How many types of coupons (distinct buckets) will Alice get if she purchases $pageCard$ coupons (tuples)? Given Equation 2, the mathematical expectation of $pageH$ can be easily found as follows:

$$pageH = H\times(1-(1-\frac{1}{H})^{pageCard}) \quad (3)$$

The number of Hippo index entries is equivalent to the total number of tuples in the indexed table divided by the average number of tuples summarized by each index entry, i.e., $\frac{Card}{T}$. Hence, the number of index entries is given in Equation 4. The index size is equal to the product of the number of index entries and the size of a single entry. The size of each index entry is roughly equal to each partial histogram size.

$$\#\ of\ Index\ entries\ = Card/(H\times \sum_{i=0}^{D\times H-1} \frac{1}{H-i}) \quad (4)$$

Given Equation 4, we observe the following: (1) For a certain H, the higher the value of D, the less Hippo index entries there exist. (2) For a certain D, the higher H there is, the less Hippo index entries there are. Meanwhile, the size of each index entry increases with the growth of the complete histogram resolution. The final I/O cost of scanning the index entries is given in Equation 5.

$$I/O_{scanning\ index} = \frac{Card}{H\times pageCard}\times(\sum_{i=0}^{D\times H-1} \frac{1}{H-i})^{-1} \quad (5)$$

Estimating the number of read data pages. Data pages summarized by each index entry are likely to be checked in the second phase of the search algorithm, filtering false positive pages, if their associated partial histogram has joint buckets with the query predicate. Determining the probability of having joint buckets contributes to the query I/O cost estimation. The probability that a partial histogram in an index entry has joint buckets with a query predicate depends on how likely a predicate overlaps with the highlighted area in partial histograms (see Figure 3). The probability is determined by the equation given below:

$$Prob = (Average\ buckets\ hit\ by\ a\ query\ predicate)\times D$$
$$= SF\times H\times D \quad (6)$$

To be precise, $Prob$ follows a piecewise function as follows:

$$Prob = \begin{cases} SF\times H\times D & SF\times H \leqslant \frac{1}{D} \\ 1 & SF\times H > \frac{1}{D} \end{cases}$$

Given the aforementioned discussion, we observe that (1) when SF and H are fixed, the smaller D is, the smaller $Prob$ is. (2) when H and D are fixed, the smaller SF is, the smaller $Prob$ is. (3) when SF and D are fixed, the smaller H is, the smaller $Prob$ is. It is obvious that the probability in Equation 6 is equivalent to the probability that pages in an index entry are checked in the second phase, i.e., filtering false positive pages. Since the total pages in Table R is $\frac{Card}{pageCard}$, the mathematical expectation of the number of pages in R checked by the second part, as known as the I/O cost of second part, is:

$$I/O_{filtering\ false\ positives} = (Prob\times\frac{Card}{pageCard}) \quad (7)$$

By adding up $I/O_{scanning\ index}$ (Equation 5) and $I/O_{filtering\ false\ positives}$ (See Equation 7), the total query I/O cost is as follows:

$$Query\ I/O = \frac{Card}{pageCard}\times((H\times \sum_{i=0}^{D\times H-1} \frac{1}{H-i})^{-1} + Prob) \quad (8)$$

7. EXPERIMENTS

This section provides a comprehensive experimental evaluation of Hippo. All experiments are run on an Ubuntu Linux 14.04 64 bit machine with 8 cores CPU (3.5 GHz per core), 32 GB memory, and 2 TB magnetic hard disk. We install PostgreSQL 9.5 (128 MB default buffer pool) on the test machine.

Compared Approaches. During the experiments, we study the performance of the following indexing schemes: (1) Hippo: A complete prototype of our proposed indexing approach implemented inside the core engine of PostgreSQL 9.5. Unless mentioned otherwise, the default partial

histogram density is set to 20% and the default histogram resolution (H) is set to 400. (2) B$^+$-Tree: The default implementation of the B$^+$-Tree in PostgreSQL 9.5 (with a default fill factor of 90), (3) BRIN: A sparse Block Range Index implemented in PostgreSQL 9.5 with 128 default pages per range. We also consider other BRIN settings, i.e., BRIN-32 and BRIN-512, with 32 and 512 pages per range respectively.

Datasets. We use the following four datasets:

- TPC-H : A 207 GB decision support benchmark that consists of a suite of business oriented ad-hoc queries and data modifications. Tables populated by TPC-H follow a uniform data distribution. For evaluation purposes, we build indexes on Linitem table PartKey, SuppKey or OrderKey attribute. PartKey attribute has 40 million distinct values while SuppKey has 2 million distinct values and the values of OrderKey attribute are sorted. For TPC-H benchmark queries, we also build indexes on L_Shipdate and L_Receiptdate when necessary.

- Exponential distribution synthetic dataset (abbr. Exponential): This 200 GB dataset consists of three attributes, IncrementalID, RandomNumber, Payload. RandomNumber attribute follows exponential data distribution which is highly skewed.

- Wikipedia traffic (abbr. Wikipedia) [2]: A 231 GB Wikipedia article traffic statistics covering seven months period log. The log file consists of 4 attributes: PageName, PageInfo, PageCategory, PageCount. For evaluation purposes, we build index on the PageCount attribute which stands for hourly page views.

- New York City taxi dataset (abbr. NYC Taxi) [1]: This dataset contains 197 GB New York City Yellow and Green Taxi trips published by New York City Taxi & Limousine Commission website. Each record includes pick-up and drop-off dates/times, pick-up and drop-off locations, trip distances, and itemized fares. We reduce the dimension of pick-up location from 2D (longitude, latitude) to 1D (integer) using a spatial dimension reduction method, Hilbert Curve, and build indexes on pick-up location attribute.

Implementation details. We have implemented a prototype of Hippo inside PostgreSQL 9.5 as one of the main index access methods by leveraging the underlying interfaces which include but not limited to "ambuild", "amgetbitmap", "aminsert" and "amvacuumcleanup". A PostgreSQL 9.5 user creates and queries the index as follows:

```
CREATE INDEX hippo_idx ON lineitem USING hippo(partkey)

SELECT * FROM lineitem
         WHERE partkey > 1000 AND partkey < 2000

DROP INDEX hippo_idx
```

The final implementation has slight differences from the aforementioned details due to platform-dependent features. For instance, Hippo only records possible qualified page IDs in a tid bitmap format and returns it to the kernel. PostgreSQL automatically inspects pages and checks each tuples against the query predicate. PostgreSQL `DELETE` command does not really remove data from disk unless a `VACUUM` command is called automatically or manually. Hippo then updates the index for data deletion when a `VACUUM` command is invoked. In addition, it is better to rebuild Hippo index if there is a huge change of the parent attribute's histogram. Furthermore, a script, running as a background process, drops the system cache during the experiments.

Table 3: Tuning Parameters

Parameter	Value	Size	Initial. time	Query time
Default	D=20% R=400	1012 MB	2765 sec	2500 sec
Density (D)	40%	680 MB	2724 sec	3500 sec
	80%	145 MB	2695 sec	4500 sec
Resolution (R)	800	822 MB	2762 sec	3000 sec
	1600	710 MB	2760 sec	3500 sec

7.1 Tuning Hippo parameters

This section evaluates the performance of Hippo by tuning two main system parameters: partial histogram density D (Default value is 20%) and complete histogram resolution H (Default value is 400). For these experiments, we build Hippo on PartKey attribute in Lineitem table of 200 GB TPC-H benchmarks. We then evaluate the index size, initialization time, and query response time.

7.1.1 Impact of partial histogram density

The following experiment compares the default Hippo density (20%) with two different densities (40% and 80%) and tests their query time with selectivity factor 0.1%. As given in Table 3, when we increase the density Hippo exhibits less indexing overhead as expected. That happens due to the fact that Hippo summarizes more pages per partial histogram and write less index entries on disk. Similarly, higher density leads to more query time because it is more likely to overlap with query predicates and result in more pages are selected as possible qualified pages.

7.1.2 Impact of histogram resolution

This section compares the default Hippo histogram resolution (400) to two different histogram resolutions (800 and 1600) and tests their query time with selectivity factor 0.1%. The density impact on the index size, initialization time and query time is given in Table 3 .

As given in Table 3, with the growth of histogram resolution, Hippo size decreases moderately. The explanation is that higher histogram resolution leads to less partial histograms and each partial histogram in the index may summarize more pages. However, the partial histogram (in bitmap format) has larger physical size because the bitmap has to store more bits.

As Table 3 shows, the query response time of Hippo varies with the growth of histogram resolution. This is because for the large histogram resolution, the query predicate may hit more buckets so that this Hippo is more likely to overlap with query predicates and result in more pages are selected as possible qualified pages.

7.2 Indexing overhead

This section studies the indexing overhead (in terms of index size and index initialization time) of the B$^+$-Tree,

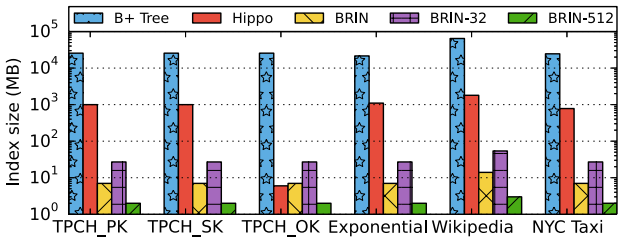


Figure 5: Index size on different datasets (log. scale)

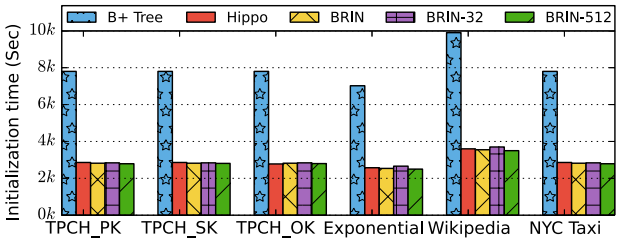
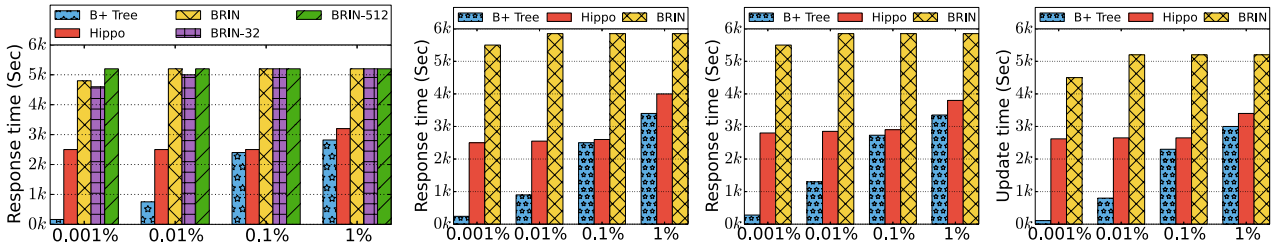


Figure 6: Index initial. time on different datasets



(a) TPC-H **(b)** Exponential **(c)** Wikipedia **(d)** NYC Taxi

Figure 7: Query response time at different selectivity factors

Hippo, BRIN (128 pages per range by default), BRIN-32, and BRIN-512. The indexes are built on TPC-H Lineitem table PartKey (TPCH_PK), SuppKey (TPCH_SK), OrderKey (TPCH_OK) attributes, Exponential table RandomNumber attribute, Wikipedia table PageCount attribute, NYC Taxi table pick-up location attribute.

As given in Figure 5, Hippo occupies 25 to 30 times smaller storage space than the B$^+$-Tree on all datasets (except on TPC-H OrderKey attribute). This happens because Hippo only stores disk page ranges along with page summaries. Furthermore, Hippo on TPC-H PartKey incurs the same storage space as that of the index built on the SuppKey attribute (which has 20 times less distinct attribute values). That means the number of distinct values does not actually impact Hippo index size as long as it is larger than the number of complete histogram buckets. Each attribute value has the same probability to hit a histogram bucket no matter how many distinct attribute values there are. This is because the complete histogram leveraged by Hippo summarizes the data distribution of the entire table. Hippo still occupies small storage space on tables with different data distributions, such as Exponential, Wikipedia and NYC Taxi data. That happens because the complete histogram, which is height balanced makes sure that each tuple has the same probability to hit a bucket and then avoid the effect of data skewness. However, it is worth noting that Hippo has a significant size reduction when the data is sorted on TPC-H OrderKey attribute. In this case, Hippo only contains five index entries and each index entry summarizes thousands of pages. When data is totally sorted, Hippo keeps summarizing pages until the first 20% of the complete histogram buckets (No.1 - 80) are hit, then the next 20% (No. 81 - 160), and so forth. Therefore, Hippo cannot achieve competitive query time in this case.

In addition, BRIN exhibits the smallest index size among the three indexes since it only stores page ranges and corresponding value ranges (min and max values). Among different versions of BRIN, BRIN-32 exhibits the largest storage overhead while BRIN-512 shows the lowest storage overhead because the latter can summarize more pages per entry.

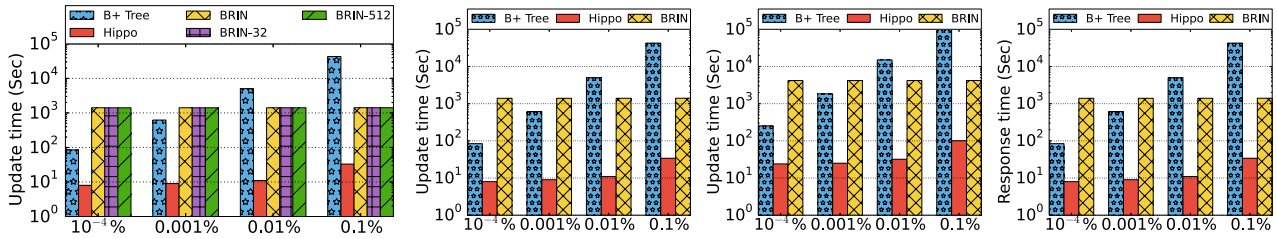
On the other hand, as Figure 6 depicts, Hippo and BRIN consume less time to initialize the index as compared to the B$^+$-Tree. That is due to the fact that the B$^+$-Tree has numerous index entries (tree nodes) stored on disk while Hippo and BRIN have just a few index entries. Moreover, since Hippo has to compare each tuple to the complete histogram which is kept in memory temporarily during index initialization, Hippo may take more time than BRIN to initialize itself. Different versions of BRIN spends most of the initialization time on scanning the data table and hence do not show much time difference.

7.3 Query response time

This section studies the query response time of the three indexes, B$^+$-Tree, Hippo and BRIN (128 pages by default). We first evaluate the query response time of the three indexes when different query selectivity factors are applied. Then, we further explore the performance of each index using the TPC-H benchmark queries which deliver industry-wide practical queries for decision support.

7.3.1 Queries with different selectivity factors

This experiment studies the query execution performance while varying the selectivity factor from 0.001%, 0.01%, 0.1% to 1%. According to the Hippo cost model, the corresponding query time costs in this experiment are $0.2Card$, $0.2Card$, $0.2Card$ and $0.8Card$. The indexes are built on TPC-H Lineitem table PartKey attribute (TPC-H), Exponential table RandomNumber attribute, Wikipedia table PageCount attribute, NYC Taxi table pick-up location attribute. We also compare different versions of BRIN (BRIN-32 and BRIN-512) on TPC-H PartKey attribute.



(a) TPC-H **(b)** Exponential **(c)** Wikipedia **(d)** NYC Taxi

Figure 8: Data update time (logarithmic scale) at different update percentage

As the results shown in Figure 7, all the indexes consume more time to query data on all datasets with the increasing of query selectivity factors. All versions of BRIN are two or more times worse than B$^+$-Tree and Hippo at almost all selectivity factors. They have to scan almost the entire tables due to their very insufficient page summaries - value ranges. Moreover, B$^+$-Tree is not better than Hippo at 0.1% query selectivity factor although it is faster than Hippo at low query selectivity factors like 0.001% and 0.01%. Actually, the performance of Hippo is very stable on all datasets including highly skewed data and real life data. In addition, Hippo consumes much more time at the last selectivity factor 1% because it has to scan much more pages as predicted by the cost model. Compared to the B$^+$-Tree, Hippo maintains a competitive query response time performance at selectivity factor 0.1% but consumes 25 - 30 times less storage space. In contrast to BRIN, Hippo achieves less query response time at the small enough index size. Therefore, we may conclude that Hippo makes a good tradeoff between query response time and index storage overhead at medium query selectivity factors, i.e, 0.1%.

7.3.2 *Evaluating the cost model accuracy*

This section conducts a comparison between the estimated query I/O cost and the actual I/O cost of running a query on Hippo. In this experiment, we vary the query selectivity factors to take the values of 0.001%, 0.01%, 0.1%, and 1%. Hence, the average number of buckets hit by the query predicate ($SF * H$) should be 0.004, 0.04, 0.4, and 4 respectively. However, in practice, no in-boundary queries can hit less than 1 bucket. Therefore, the average hit buckets by predicates are 1, 1, 1 and 4. Given $H = 400$ and $D = 20\%$, the query I/O cost estimated by Equation 8 is $\frac{Card}{pageCard} * (0.05\% + 20\%|20\%|20\%|80\%)$. We observe that: (1) Queries for the first three SF values yields pretty similar I/O cost. That matches the experimental results depicted in Figure 7. (2) The I/O cost of scanning index entries consumes $\frac{Card}{pageCard} * 0.05\%$ which is at least 40 times less than that of filtering false positive pages.

Table 4: The estimated query I/O deviation from the actual query I/O for different selectivity factors

SF	0.001%	0.01%	0.1%	1%
TPC-H	0.02%	0.02%	0.21%	6.18%
Exponential	0.37%	0.37%	0.35%	12.69%
Wikipedia	0.91%	0.91%	1.19%	9.10%
NYC Taxi	0.87%	0.87%	0.51%	13.39%

As Table 4 shows, the cost model exhibits high accuracy. Furthermore, the cost model accuracy is stable especially for the first three lower selectivity factors. However, when $SF = 1\%$, the accuracy is relatively lower especially on Exponential and NYC Taxi table. The reason behind that is two-fold: (1) The 1% selectivity factor query predicate may hit more buckets than the other lower SF values. That leads to quite different overlap situations with partial histograms. (2) The complete height balanced histogram, maintained by the DBMS, does not perfectly reflect the data distribution since it is created periodically using some statistical approaches. Exponential and NYC Taxi tables exhibit relatively more clustered/skewed data distribution. That makes it more difficult to reflect their data distribution accurately. On the other hand, the histogram of a uniformly distributed TPC-H table is very accurate so that predicated I/O cost is more accurate in this case.

7.3.3 *TPC-H benchmark queries*

This section compares Hippo to the B$^+$-Tree and BRIN using the TPC-H benchmark queries. We select all TPC-H benchmark queries that contain typical range queries and hence need to access an index. We then adjust their selectivity factors to 0.1% (i.e., one week reports). We build the three indexes on the L_ShipDate (Query 6, 7, 14, 15 and 20) and L_ReceiptDate (Query 12) attributes in the Lineitem table as required by the queries. The qualified queries, Query 6, 7, 12, 14, 15 and 20, perform at least one index search (Query 15 performs twice) on the evaluated indexes.

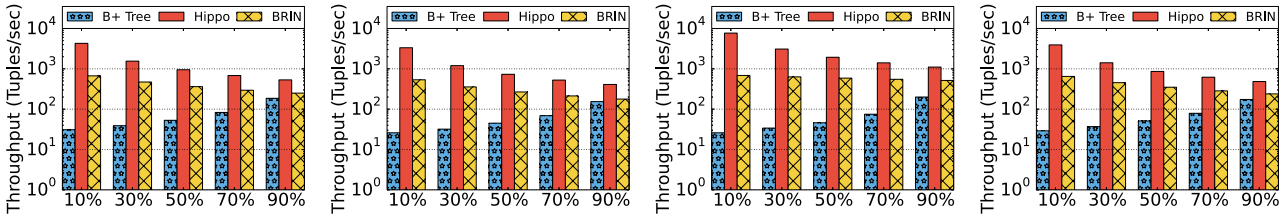
Table 5: Query response time (Sec) on TPC-H

Index type	Q6	Q7	Q12	Q14	Q15	Q20
B$^+$-Tree	2450	259000	2930	2670	4900	3500
Hippo	2700	260400	3200	3180	5400	3750
BRIN	5600	276200	6200	6340	11300	6700

As Table 5 depicts, Hippo achieves similar query response time to that of the B$^+$-Tree and runs around two times faster than BRIN on all selected TPC-H benchmark queries. It is also worth noting that the time difference among all three indexes becomes non-obvious for Query 7. That happens because the query processor spends most of the time joining multiple tables, which dominates the execution time for Q7.

7.4 Maintenance Overhead

This experiment investigates the index maintenance time of three kinds of indexes, B$^+$-Tree, Hippo and BRIN, on



(a) TPC-H **(b)** Exponential **(c)** Wikipedia **(d)** NYC Taxi

Figure 9: Throughput on different query / update workloads (logarithmic scale)

all datasets when insertions or deletions. The indexes are built on TPC-H Lineitem table PartKey attribute, Exponential table RandomNumber attribute, Wikipedia table PageCount attribute, and NYC Taxi table Pick-up location attribute. This experiment uses a fair setting which counts the batch maintenance time after randomly inserting or deleting a certain amount (0.0001% , 0.001%, 0.01%, and 0.1%) of tuples. In addition, after inserting tuples into the parent table, the indexes' default update operations are executed because they adopt an eager strategy to keep indexes up to date. However, after deleting the certain amount of tuples from the parent table, we rebuild BRIN from scratch because BRIN does not have any proper update strategies for deletion. We also compare different versions of BRIN (BRIN-32 and BRIN-512) on TPC-H PartKey attribute.

As depicted in Figure 8 (in a logarithmic scale), Hippo costs up to three orders of magnitude less time to maintain the index than the B$^+$-Tree and up to 50 times less time than all versions of BRIN. This happens because the B$^+$-Tree spends more time on searching proper index entry insert / delete location and adjusting tree nodes. On the other hand, BRIN's maintenance is very slow after deletion since it has to rebuild the index after a batch of delete operations.

7.5 Performance on hybrid workloads

This section studies the performance of Hippo in hybrid query/update workloads. In this experiment, we build the considered indexes on TPC-H Lineitem table PartKey attribute, Exponential table RandomNumber attribute, Wikipedia table PageCount attribute, and NYC Taxi table Pick-up location attribute. We use five different hybrid query/update workloads: 10%, 30%, 50%, 70% and 90%. The percentage here stands for the percentage of queries in the entire workload. For example, 10% means 10% of the operations that access the index are queries and 90% are updates. The average selectivity factor is 0.1%. The index performance is measured by throughput (Tuples/second) defined as the number of qualified tuples queried or updated per a given period of time. The results are given in Figure 9 (in logarithmic scale).

As it turns out in Figure 9, Hippo has the highest throughput on all workloads. Hippo and BRIN can have higher throughput at update-intensive workloads like 10% and 30%. That happens since Hippo and BRIN have less index maintenance time that that of the B$^+$-Tree. On the other hand, B$^+$-Tree achieves higher throughput on query-intensive workloads like 70% and 90%. This is due to the fact that B$^+$-Tree costs less or same query response time compared to Hippo. Therefore, we can conclude that Hippo performs orders of magnitudes better than BRIN and B$^+$-Tree for update-intensive workload. Furthermore, for query intensive workloads, Hippo still can exhibit slightly better throughput than that of the B$^+$-Tree at a much small index storage overhead.

8. RELATED WORK

Tree Indexes B$^+$-Tree is the most commonly used type of indexes. The basic idea can be summarized as follows: For a non-leaf node, the value of its left child node must be smaller than that of its right child node. Each leaf node points to the physical address of the original tuple. With the help of this structure, searching B$^+$-Tree can be completed in one binary search time scope. The excellent query performance of B$^+$-Tree and other tree like indexes is benefited by their well designed structures which consist of many non-leaf nodes for quick searching and leaf nodes for fast accessing parent tuples. This feature incurs two inevitable drawbacks: (1) Storing plenty of nodes costs a huge chunk of disk storage. (2) Index maintenance is extremely time-consuming. For any insertions or deletions occur on parent table, tree like indexes firstly have to traverse themselves for finding proper update locations and then split, merge or re-order one or more nodes which are out of date.

Bitmap Indexes A Bitmap index [12, 16, 21] has been widely applied to low cardinality and read-only datasets. It uses bitmaps to represent values without trading query performance. However, Bitmap index's storage overhead significantly increases when indexing high cardinality attributes because each index entry has to expand its bitmap to accommodate more distinct values. Bitmap index also does not perform well in update-intensive workloads due to tuple-wise index structure.

Compressed Indexes Compressed indexes drop some repeated index information to save space and recover it as fast as possible upon queries but they all have guaranteed query accuracy. These techniques are applied to tree indexes [9, 10]. Though compressed indexes are storage economy, they require additional time for compressing beforehand and decompressing on-the-fly. Compromising on the time of initialization, query and maintenance is not desirable in many time-sensitive scenarios.

Approximate Indexes Approximate indexes [4, 11, 14] give up the query accuracy and only store some representative information of parent tables for saving indexing and maintenance overhead and improving query speed. They propose many efficient statistics algorithms to figure out the most representative information which is worth to be stored. In addition, some people focus on approximate query processing (AQP)[3, 20] which relies on data sampling and error

bar estimating to accelerate query speed directly. However, trading query accuracy makes them applicable to limited scenarios such as loose queries.

Sparse Indexes A spars index, e.g., as Zone Map [5], Block Range Index [17], Storage Index [18], and Small Materialized Aggregates (SMA) index [13], only stores pointers to disk pages / column blocks in parent tables and value ranges (min and max values) in each page / column block so that it can reduce the storage overhead. For a posed query, it finds value ranges which cover the query predicate and then inspects the associated few parent table pages one by one for retrieving truly qualified tuples. However, for unordered data, a sparse index has to spend lots of time on page scanning since the stored value ranges (min and max values) may cover most query predicates. In addition, column imprints [15], a cache-conscious secondary index, significantly enhances the traditional sparse indexes to speed up queries at a reasonably low storage overhead in-memory data warehousing systems. It leverages histograms and bitmap compression but does not support dynamic pages / column block size control to further optimize the storage overhead reduction especially with partially clustered data. The column imprints approach is designed to handle query-intensive workloads and puts less emphasis on efficiently update the index in row stores.

9. CONCLUSION AND FUTURE WORK

The paper introduces Hippo a data-aware sparse indexing approach that efficiently and accurately answers database queries. Hippo occupies up to two orders of magnitude less storage overhead than de-facto database indexes, i.e., B^+-tree while achieving comparable query execution performance. To achieve that, Hippo stores page ranges instead of tuples in the indexed table to reduce the storage space occupied by the index. Furthermore, Hippo maintains histograms, which represent the data distribution for one or more pages, as the summaries for these pages. This structure significantly shrinks index storage footprint without compromising much performance on high and medium selectivity queries. Moreover, Hippo achieves about three orders of magnitudes less maintenance overhead compared to the B^+-tree and BRIN. Such performance benefits make Hippo a very promising alternative to index high cardinality attributes in big data application scenarios. Furthermore, the simplicity of the proposed structure makes it practical for DBMS vendors to adopt Hippo as an alternative indexing technique. In the future, we plan to adapt Hippo to support more complex data types, e.g., spatial data, unstructured data. We also plan to study more efficient concurrency control mechanisms for Hippo. Furthermore, we also plan to extend Hippo to function within the context of in-memory database systems as well as column stores.

10. ACKNOWLEDGEMENT

This work is supported by the National Science Foundation under Grant 1654861.

11. REFERENCES

[1] New york city taxi and limousine commission. `http://www.nyc.gov/html/tlc/html/about/trip\_record\_data.html`.

[2] Page view statistics for wikimedia projects. `https://dumps.wikimedia.org/other/pagecounts-raw/`.

[3] S. Agarwal, H. Milner, A. Kleiner, A. Talwalkar, M. Jordan, S. Madden, B. Mozafari, and I. Stoica. Knowing when you're wrong: building fast and reliable approximate query processing systems. In *Proceedings of the International Conference on Management of Data, SIGMOD*, pages 481–492. ACM, 2014.

[4] M. Athanassoulis and A. Ailamaki. Bf-tree: Approximate tree indexing. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, pages 1881–1892. VLDB Endowment, 2014.

[5] C. Bontempo and G. Zagelow. The ibm data warehouse architecture. *The Communications of the ACM*, 41(9):38–48, 1998.

[6] T. P. P. Council. Tpc-h benchmark specification. *http://www. tcp. org/hspec. html*, 2008.

[7] P. Flajolet, D. Gardy, and L. Thimonier. Birthday paradox, coupon collectors, caching algorithms and self-organizing search. *Discrete Applied Mathematics*, 39(3):207–229, 1992.

[8] F. Fusco, M. P. Stoecklin, and M. Vlachos. Net-fli: on-the-fly compression, archiving and indexing of streaming network traffic. *VLDB Journal*, 3(1-2):1382–1393, 2010.

[9] J. Goldstein, R. Ramakrishnan, and U. Shaft. Compressing relations and indexes. In *Proceedings of the International Conference on Data Engineering, ICDE*, pages 370–379. IEEE, 1998.

[10] G. Guzun, G. Canahuate, D. Chiu, and J. Sawin. A tunable compression framework for bitmap indices. In *Proceedings of the International Conference on Data Engineering, ICDE*, pages 484–495. IEEE, 2014.

[11] M. E. Houle and J. Sakuma. Fast approximate similarity search in extremely high-dimensional data sets. In *Proceedings of the International Conference on Data Engineering, ICDE*, pages 619–630. IEEE, 2005.

[12] D. Lemire, O. Kaser, and K. Aouiche. Sorting improves word-aligned bitmap indexes. *Data & Knowledge Engineering*, 69(1):3–28, 2010.

[13] G. Moerkotte. Small materialized aggregates: A light weight index structure for data warehousing. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, pages 476–487. VLDB Endowment, 1998.

[14] Y. Sakurai, M. Yoshikawa, S. Uemura, H. Kojima, et al. The a-tree: An index structure for high-dimensional spaces using relative approximation. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, pages 5–16. VLDB Endowment, 2000.

[15] L. Sidirourgos and M. L. Kersten. Column imprints: a secondary index structure. In *Proceedings of the International Conference on Management of Data, SIGMOD*, pages 893–904. ACM, 2013.

[16] K. Stockinger and K. Wu. Bitmap indices for data warehouses. *Data Warehouses and OLAP: Concepts, Architectures and Solutions*, page 57, 2006.

[17] M. Stonebraker and L. A. Rowe. The design of postgres. In *Proceedings of the International Conference on Management of Data, SIGMOD*. ACM, 1986.

[18] R. Weiss. A technical overview of the oracle exadata database machine and exadata storage server. *Oracle White Paper. Oracle Corporation, Redwood Shores*, 2012.

[19] K. Wu, E. Otoo, and A. Shoshani. On the performance of bitmap indices for high cardinality attributes. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, pages 24–35. VLDB Endowment, 2004.

[20] K. Zeng, S. Gao, B. Mozafari, and C. Zaniolo. The analytical bootstrap: a new method for fast error estimation in approximate query processing. In *Proceedings of the International Conference on Management of Data, SIGMOD*, pages 277–288. ACM, 2014.

[21] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-scalar ram-cpu cache compression. In *Proceedings of the International Conference on Data Engineering, ICDE*, pages 59–59. IEEE, 2006.