Hippo in Action: Scalable Indexing of a Billion New York City Taxi Trips and Beyond

Jia Yu[1] Raha Moraffah[2] Mohamed Sarwat[3]
Arizona State University, 699 South Mill Ave., Tempe, AZ 85281
[1]jiayu2,[2]rmoraffa,[3]msarwat@asu.edu

Abstract

The paper demonstrates Hippo a lightweight database indexing scheme that significantly reduces the storage and 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 then inspects the remaining pages. We demonstrate Hippo using a billion NYC taxi trip records. Video: http://www.youtube.com/watch?v=wWaOK2-9k9A

I. Introduction

The volume of available data is increasing at staggering rates. For instance, New York City Taxi and Limousine Commission has recently released a 260 Gigabytes dataset of NYC Yellow Cab and Green Taxi trips, *abbr.* NYC Taxi [1]. The dataset contains detailed records of over 1.1 billion individual taxi trips in the city from January 2009 through June 2015. Each record includes pick-up and drop-off dates/times, pick-up and drop-off precise location coordinates, trip distances, itemized fares, payment method, and travel distance. To make sense of such data, the first step is to digest the dataset in a database systems, e.g., PostgreSQL, and then issue queries, e.g., find all taxi trips to the Laguardia airport (Figure 3). To speed up such queries, a user may build an index, e.g., B^+-Tree and R-tree, on frequently accessed attributes (e.g., Taxi pick-up location). Although 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. Moreover, the dollar cost increases dramatically when the DBMS is deployed on modern storage devices, e.g., SSD, because they are still more than an order of magnitude expensive than Hard Disk Drives. Also, initializing an index may be a time consuming process especially when the index is created on a large table.

Maintenance Overhead: A DBMS must update the index after inserting (deleting) tuples into (from) the underlying

This work is supported by the National Science Foundation Grant 1654861

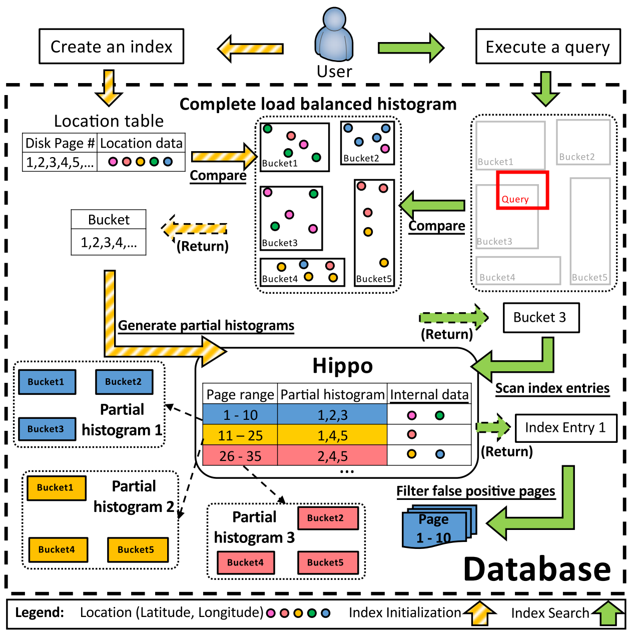


Fig. 1: Initialization and search in Hippo (best viewed in color)

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.

In this paper, we demonstrate a novel database indexing scheme, namely Hippo [2]. The proposed indexing scheme is able to (1) process queries nearly as fast as the B-tree or R-tree for medium selectivity queries, (2) occupies up to two orders of magnitude less storage overhead than de-facto database indexes, e.g.,, B^+-tree and R-Tree, while achieving comparable query execution performance., and (3) handle data updates quickly to handle update intensive applications (e.g., new taxi trips are inserted into the database all the time). Hippo achieves about three orders of magnitudes less maintenance overhead compared to existing indexes supported by major database vendors. We implemented a prototype of Hippo inside the core engine of PostgreSQL 9.6.1[1]. The demo attendee can query/visualize the indexed NYC taxi data on the map and also compare index performance of a variety of indexes, e.g.,

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

B-Tree, R-Tree, and Hippo, using PostgreSQL.

II. System Overview

Figure 1 depicts a running example that describes the index initialization (left part of the figure) and search (right part of the figure) processes. Hippo consists of n index entries such that each entry consists of the following two components:

Summarized Page Range: Two integers that represent the IDs of the first and last pages summarized 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 - 35 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.

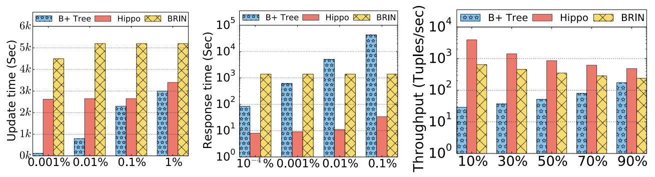
Histogram-based Summary: A bitmap that represents a subset of the complete load balanced histogram buckets (maintained by the underlying DBMS), aka. partial histogram. Each bucket, if exists, indicates that at least one of the tuples of this bucket exists in the summarized pages. Each partial histogram represents the spatial distribution of the data in the summarized contiguous pages. Since each bucket of a load 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 and skewed. We can build a 2 dimension complete histogram or convert 2D data to 1D using Hilbert Curve and create 1D histogram.

The main idea behind Hippo is as follows: (1) Each index entry summarizes many pages and each only stores two page IDs and a compressed bitmap, and (2) Each page of the parent table is only summarized by one Hippo index entry. 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 then 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.

III. Demonstration Scenario

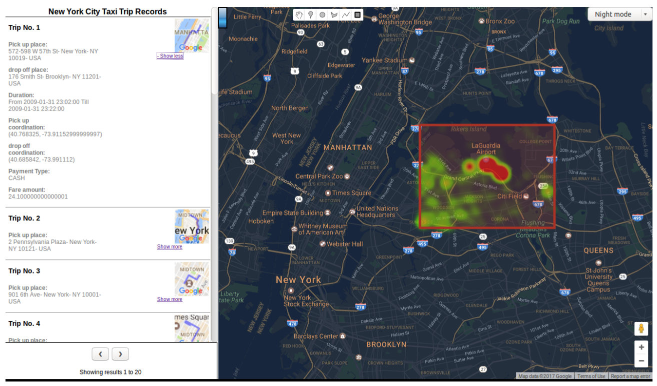
We demonstrate Hippo using the NYC taxi dataset (described in Section I). First, we store the dataset as a table in PostgreSQL and then create an index on the pick-up location attributes (represented using Hilbert Curve). The following SQL shows how we create the index using Hippo:

```
CREATE INDEX hippoidx ON NYCTAXI USING hippo(PickupLocation)
```



(a) Query (b) Update (c) Hybrid workload

Fig. 2: Index performance (2b and 2c in logarithmic scale)



New York City Taxi Trip Records

Trip No. 1

Pick up place:
572-598 W 57th St- New York- NY 10019- USA

- Show less

drop off place:
176 Smith St- Brooklyn- NY 11201- USA

Duration:
From 2009-01-31 23:02:00 Till 2009-01-31 23:22:00

Pick up coordination:
(40.768325, -73.91152999999997)

drop off coordination:
(40.685842, -73.991112)

Payment Type:
CASH

Fare amount:
24.100000000000001

Trip No. 2

Pick up place:
2 Pennsylvania Plaza- New York- NY 10121- USA

Show more

Trip No. 3

Pick up place:
901 6th Ave- New York- NY 10001- USA

Show more

Trip No. 4

Pick up place:

Showing results 1 to 20

Fig. 3: Query Taxi Trips and visualize the results on the Map

Our experiments on comparing indexing overhead have shown that Hippo (780 MB, 0.8 hour) has at least 30 times less storage overhead and 2.8 times less initialization time than B^+-Tree (25 GB and 2.2 hour). The size of Hippo can be further reduced by tuning its parameter. We also compare these two indexes with Block Range Index (BRIN, a sparse index in PostgreSQL). Query response time, data update time, and hybrid workloads (query/update) are depicted in Figure2a, 2b and 2c. Figure 2a indicates that when the selectivity factor is 0.1%-1%, Hippo costs similar query response time with B^+-Tree while queries on BRIN are two times slower than that on Hippo. Figure 2b depicts that, on different data update percentage, Hippo is at least one order of magnitude faster than other two indexes. As shown in Figure 2c, on different percentage of query operations in the entire query/update hybrid workload, Hippo has at least 3 times higher throughput than other indexes.

We developed a web application that visualizes the NYC taxi trips based on their pick-up locations using Google Maps. The demo attendee can interact with the map to visualize the NYC trips within the designated query region (see Figure 3). An example of the SQL query issued to PostgreSQL is:

```
SELECT * FROM NYCTAXI T
WHERE ST_Within(LaGuardiaAirportArea, T.PickupLocation)
```

The details, e.g., fare amount, pick-up time, of each trip located within the query region will appear on the left pane. The user can then change the query window to visualize the taxi trips for which the pick-up/drop-off location is in a different region of NYC.

References

[1] New york city taxi and limousine commission. http://www.nyc.gov/html/tlc/html/about/trip_record_data.shtml.

[2] J. Yu and M. Sarwat. Two birds, one stone: A fast, yet lightweight, indexing scheme for modern database systems. *Proceedings of the VLDB Endowment, PVLDB*, 10(4), 2016. To appear.