A Novel Index Method for Write Optimization on Out-of-Core Column-Store Databases

by

Tyler J. Matacic

A thesis submitted to Youngstown State University in partial fulfillment of the requirements for the degree of

Master of Science

in the

Computer Information Systems

Program

YOUNGSTOWN STATE UNIVERSITY

December, 2016

A Novel Index Method for Write Optimization on Out-of-Core Column-Store Databases

Tyler J. Matacic

I hereby release this thesis to the public. I understand that this thesis will be made available from the OhioLINK ETD Center and the Maag Library Circulation Desk for public access. I also authorize the University or other individuals to make copies of this thesis as needed for scholarly research.

Signature:

Tyler J. Matacic, Student Date

Approvals:

Dr. Feng Yu, Thesis Advisor Date

Dr. Yong Zhang, Committee Member Date

Dr. Alina Lazar, Committee Member Date

Dr. Salvatore A. Sanders, Dean of Graduate Studies Date

Acknowledgments

I would like to thank my family and friends for supporting me through the process of this graduate degree program. This has been the most challenging part of my entire life, and I owe it to them for giving me the strength to make it through. I want to thank my fiance, Maria, who is the love of my life and has always encouraged me and gave me the support I needed. I would like to thank the professors who taught me so much, and the best secretary in the world, Connie who never tired no matter how many times I needed her to open a door, retrieve papers from the department mailboxes, or help me register for classes. Lastly, I would like to thank Dr. Feng Yu, my adviser, who has helped me the entire way through this challenging learning process.

Abstract

The purpose of this thesis is to extend previous research on write optimization in out-of-core column storage databases. A new type of storage model titled Timestamped Binary Association Table (TBAT) will be explored, a new update entitled Asynchronous Out-of-Core Update (AOC Update) designed to leverage the TBAT will be explained, and a new type of B-Tree titled Offset B^+ Tree (OB-tree) will be examined. The performance of the OB-tree and TBAT when utilized for selection tasks will be demonstrated through experiments comparing TBAT selection with an OB-tree index, TBAT selection without an index, and the traditional method of binary selection on a Binary Association Table (BAT). The selection speed of these three methods will be recorded and conclusions will be drawn.

Contents

List of Figures

List of Tables

1 Introduction

Column-store databases are databases that vertically partition data and store it in separate columns. Their history dates back to the 1970's when transported files were implemented in the early development of database management systems (DBMS). By the mid 1980's the advantages of early column-store databases (titled decomposed storage model or DSM) were well documented [9, 10]. Although DSM's advantages for running queries were apparent, more then three decades ago the market dictated that traditional row-based systems would remain popular for the years to come. In recent years column-store systems have attracted renewed attention because of their unique structure, specific efficiencies, and flexibility in usage. As mentioned previously Column-stores store each database attribute separately in columns, the attribute values that belong to the same column are stored contiguously, allowing them to be densely packed and compressed leading to tremendous space savings on disk. This runs counter to traditional database systems that store data in rows one after the other. When a database query is run on a traditional row-based database the entire row must be read first, and then the specific requested attributes are pulled. This method of extracting data is costly and uses unneeded resources. The structure of column-store databases lends themselves perfectly to read heavy tasks because instead of needing to return entire tables like the row-based database, column-store databases return only the requested attribute column [1].

The basic architectural design of a column-store is dictated by the columnar

data layout. Different column-store databases employ different techniques for data compression, such as the "projections" utilized by C-Store [17]. In C-Store these projections are groups of columns that are sorted on the same attribute, this architecture allows the support of many sort-orders without a large increase in space, and opens opportunities for optimization. C-store was so successful that it was adapted for wide commercial use and re-named to Vertica seven years after its original inception [16].

Column-store databases work very well with large stores of data. Because of this fact it is only natural that a column-store would be created to work with one of the most popular distributed computing frameworks for processing big data, the Hadoop Distributed File System (HDFS). The column-store build on top of Hadoop is titled *HBase* [12] and it was built to integrate perfectly into HDFS utilizing features such as the ability for fast lookups on large tables, low latency access to single rows, and the storage of indexed HDFS files for reduced look-up speed [19, 8].

Column based layouts are extremely flexible in their implementation, and can even be tailored to work with cutting edge technology such as Solid State Drives (SSDs). The PAX data organization model, originally proposed to improve CPU cache performance is able to facilitate reading from the SSD only the attributes that participate in a query. This efficient data reading reduces the data read and saves time and resources [18, 6, 7].

It is clear that column-store databases are adept and optimized at reading data efficiently and quickly. Writing optimizations on the other hand are more of a

challenge with column-stores, and the data presented in this paper works to alleviate this challenge.

The methods discussed later in this work highlight the usage of OB-Trees to quickly search for records within a column-store database, Section 2, Background of column-store Databases, discuses the history of column-store databases and their uses. Section 3 Timestamped BAT, examines a new storage method for column-store databases. Section 4 Selection Speed Degradation after AOC Updates, discusses methods of cleaning the Timestamped BAT in order to restore its speed. Section 5, Offset B^+ Tree, examines a new data structure created to work with the TBAT. Section 6, OB-tree selection, explains the use of the OB-tree to increase the speed of selection tasks on the TBAT data structure. Section 7, Experiment Results, showcases the results of the OB-tree selection tests and compares them to the selection speed of a traditional BAT and that of a TBAT without an OB-tree index. Section 8, Conclusion and Future Works, will give a summary of the experiment findings and detail the direction of future work.

2 Background of Column-Store Databases

Column store databases differentiate themselves from traditional databases by their unique method of storing data [21]. Instead of storing data on disk as a horizontally organized table block, column-store databases break tables into individual vertically partitioned columns. Figure 1(a) depicts an example of a conventional table titled

id	name	fee
1	Maria	100.00
2	Brandon	200.00
3	John	300.00

(a) Row-Based `customer`

oid	int
101	1
102	2
103	3

(b) `customer_id`

oid	varchar
101	Maria
102	Brandon
103	John

(c) `customer_name`

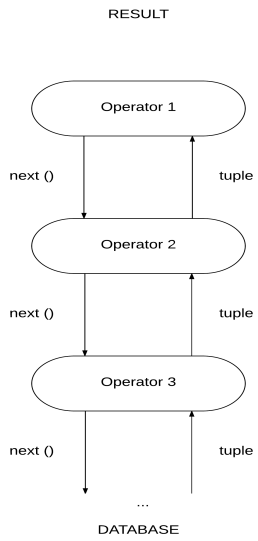
oid	float
101	100.00
102	200.00
103	300.00

(d) `customer_fee`

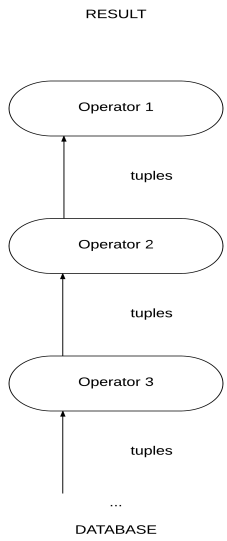
Figure 1: `customer` Data in Row-Based (a) and BAT (b) (c) (d) format

customer, consisting of three attributes, **name**, **fee**, and **id**. In a column-store database these three attributes would be separated and stored randomly in memory as three different columns titled BAT's (binary association tables) shown in Figures 1(b), 1(c), and 1(d). The BAT is a special storage model that is categorized as a Decomposed Storage Model [9]. Column-store databases have a special method of retrieving this randomly stored data in the form of an object identifier (OID). The OID is a numerical value connected to each attribute in the vertical columns. When a query is run to retrieve an attribute it searches for the OID using a sequential search and returns the appropriate value. The combination of the OID and attribute value is titled a BUN (Binary UNits). Modern column-store databases have the advantage of the latest processes and methods, enabling them to work faster than ever before. Even if a traditional relational database is modified heavily it still cannot contend with the reading speed of a column-store [4]. In Abadi's "The Design and Implementation of Modern Column-Oriented Database Systems" [2] many of the most notable strategies implemented by column-stores are discussed. One of the most heavily optimized database features is materialization. Put simply, materialization is a constructed tuple, or table, that is the result of a series of database queries.

For the query execution layer of database processing there are two primary



(a) Volcano Processing



(b) Full Materialization

Figure 2: Materialization

methods to achieve this materialization, the "Volcano-style" iterator model and full materialization. In volcano processing, shown in Figure 2(a), one tulpe at a time is pushed through the query plan tree using a method called **next()**. This method creates one new tuple at a time because each operator calls **next()**, this allows each operator to retain its own state and pass the tuple along to the next operator.

In full materialization, shown in Figure 2(b), instead of each operator working in tandem to create a tuple, each operator works in isolation to fully consume an input from disk and writes an output back to disk. The column-store MonetDB uses full materialization, the reason being is its BAT algebra and the desire to make interactions with operators simpler.

Although Volcano-processing and full materialization have their uses a column-store titled VectorWise utilizes a new query execution method titled vectorized execu-

tion. The goal of vectorized execution is to strike a balance between volcano-style and full materialization through the separation of query processing and data processing. The first step in accomplishing this is the emulation of the **next()** method with one important change. Instead of **next()** only returning a single tuple it has the ability to return any number of tuples, thus making a significant change to control flow from volcano-style processing. In regards to data processing, vectorized execution processes data vector-at-a-time, comparing data values. This hybrid architecture gives the VectorWise column-store multiple advantages such as parallel memory access, profiling, and adaptive execution highlighting the flexibility of column-store databases. As of 2012 VectorWise was leading in the TPC-H benchmark for non-clustered database systems [27, 14, 26].

As mentioned earlier data compression is a natural fit for column-store databases. There are four main attributes that lead to this efficient data compression. First is the ability to compress one column at a time. Since column-stores store all similar data in columns, compressing these columns is much simpler than row-stores that have multiple data types in a row. Moreover, compression algorithms are able to compress larger amounts of data that have common similarities, since more data of the same type will fit on a single page [3, 5, 11]. Also assisting in compression is the fact that the codes for compressing like-data will be smaller, which leads to enhanced compression. This efficient and effective compression leads to many benefits. In a business or academic setting less CPU cycles are always preferred, with fewer cycles data is processed faster. When data is compressed less data is needed to be read into

memory leading to faster processing.

According to Abadi, *et al.*, the best compression schemes to utilize for column-store databases are lightweight schemes that value the speed of decompression over the brute force of higher compression ratios [1]. Even better if these compression schemes can compress the columns into fixed-width small values, the reasoning behind this is that the compressed column can be treated as an array which is much easier to iterate through with modern CPUs. Compression's positive effects on column-stores are undeniable but there are even secondary benefits to compression that may not be immediately apparent, such as using the freed-up storage space on disk to store database copies.

There are several compression algorithms that have been evaluated for use with column-stores [2].

- **Run Length Encoding (RLE)**, takes runs of the same value in a column and compresses it into a single compact representation. Naturally RLE lends its self well to columns that are sorted, and how it works is with the replacement of runs with a triple that contains **value**, **start position**, and **runLength**. These triples are given a fixed number of bits. As an example lets say that we have a column whose first 20 elements contain the value 'T'. The triple representation for these elements would be ('T', 1, 20), *T* for the element, *1* for where in the column the element begins and *20* for how many elements are in sequence. It's easy to see how RLE would be utilized in column-stores where similar attributes

are stored consecutively with similar values [20].

- **Dictionary Encoding**, works by constructing a separate dictionary table off of a column-store table and sorts itself based on the frequency of attributes from the column-store table. Representing these attributes as an integer value in the dictionary table. These integers are then compressed using an integer specific compression scheme. The benefit of dictionary encoding is the ability to create fixed width columns if the system chooses to do so.

- **Frame of Reference (FOR)**, if a column-store has commonality with its values it can utilize this interesting compression technique. With FOR a common value is stored as a common "base" and the values that share that base are organized as simple integers following it. For an example the values: 901, 902, 904, 906, 909 would be represented as: 900, 1, 2, 4, 6, 9.

- **The Patching Technique**, FOR suffers if data contains outliers that break its chain of simplified integers, a response to this problem is to allow these outliers to be exception values that are not compressed. The patching technique creates a linked list using these exception values and has the ability to patch their information into decompressed output.

- **Operating Directly on Compressed Data**, to gain the absolute best performance boost compressed data can be operated on directly because the system saves input and output time by reading less data and doesn't have to pay the time cost of decompressing said data. How this is accomplished is through the

use of a compression block, which is a representation of the compressed data. This compression has an application program interface (API), and a buffer of compressed column data. The API allows the query optimizer to access the buffer and perform several common methods such as **getSize(), isSorted(), and getFirstValue()**.

A study conducted by MIT's computer science and artificial intelligence laboratory pitted column-store databases against their row-based counterparts. The column-stores were shown to outperform traditional row-oriented database structures by a large margin. Their study focused on comparing the performance of column and row based storage systems according to disk bandwidth, CPU cache latency, and CPU cycles. To keep their study unbiased they chose not to use techniques such as run-length-encoding that heavily favor column-store data, instead they used the compression schemes of Bit Packing, Dictionary Encoding, and Frame of Reference.

The bit packing scheme stores each table attribute using as many bits that are required to match the maximum value in the domain. The techniques of Dictionary compression and FOR-delta were previously mentioned. Using these compression techniques the study concluded column-stores to be incredibly efficient beating row-stores in almost every category, in fact on selecting 10 percent of a line-item table the column-store never took more than twelve seconds where the row-store took more than fifty [13].

Column-store databases have an important role in real-world business appli-

optime	oid	float
time1	101	100.00
time1	102	200.00
time1	103	300.00

(a) TBAT `customer_fee`

optime	oid	id
time1	101	1
time1	102	2
time1	103	3

(b) TBAT `customer_id`

optime	oid	varchar
time1	101	Maria
time1	102	Brandon
time1	103	John

(c) TBAT `customer_name`

Figure 3: TBAT Examples

cations. In this modern time enterprise database systems have come to be classified into one of two categories, optimization for online transaction processing (OLTP), and optimization for online analytical processing (OLAP). Business applications today primarily focus on the day-to-day transaction processing needed for them to run profitably, understanding the analytical processing needed to run the business is considered "after the fact". Heads of companies want to see increases in profit, not algorithms. Over time these enterprise applications have become more complex to meet the needs of OLAP and OLTP, at the same time data sets have increased and the time allotted for processing these sets has decreased. Retrieving data from this time sensitive large data set environment is where column-stores shine, and with modern compression techniques more data is able to be stored on disk and accessed quickly [15].

The background of column-store databases is complex and multifaceted, from spectacular compression optimization made possible by their column structure to business applications, column-stores are valuable methods of storing data. Their dominance in reading tasks is unmatched, but they traditionally fall behind row-store databases in writing tasks. The creation of a new data structure aims to eliminate this deficiency.

3 Timestamped BAT

Although column-store databases excel at retrieving data, their performance in writing data has always been a challenge. The random storage nature of BATs are a primary concern when optimizing write operations is considered. Recent work to alleviate this challenge has come in the form of a new way of organizing data titled the Timestamped Binary Association Table (TBAT) [22]. This TBAT differs from the BAT in one very important way, the addition of a new value. This new value is a timestamp, titled **optime** and it is used to record the time when an insert, update, or deletion is performed on a BAT tuple. The data type for **optime** is naturally a 4 byte TIMESTAMP, and it occupies only a small space in memory.

Figure 3 demonstrates created TBATs for customer_fee, customer_id, and customer_name from the **customer** table shown earlier. The **optime** value is the same for all of the attributes because this example emulates a single bulk loading performed at the same time. This new TBAT storage model can be leveraged to create fast updates by utilizing the timestamp value.

3.1 AOC Update

With a traditional BAT an update on a tuple involves two phases. First the **oid** is sought out using a sequential search according to the given target value. The second phase is to update the target value according to the **oid** found in phase one,

optime	oid	float	
time1	101	100.00	
time1	102	200.00	body
time1	103	300.00	
time2	*103*	*301.00*	appendix

Figure 4: TBAT Following AOC Update

generating random I/O on out-of-core storage (OOC). If these tasks are needed to be performed on large sets of data, the seeking of the **oid** and the changing of the values associated with it is time consuming because of the cost associated with data block seeking and writing. A data block is an interface of randomly held values compressed in memory, and its random nature takes time to seek.

Based off of the TBAT a new method of updating tuples was created called Asynchronous Out-of-core Update (AOC update) [23]. This update leverages the timestamp value of the TBAT to increase update speed significantly. The principle behind the AOC update is the avoidance of OID seeking and writing completely, and instead uses the timestamp field of the TBAT to label the newly updated data that has been directly appended to the end of the TBAT.

3.2 AOC Update Example

Figure 4, depicts an example of an update to the value *balance* where customer id equals 3. A SQL query used to accomplish this would appear as follows.

UPDATE customer SET balance = 301.00 WHERE id = 3

The target tuple for the update is the record with **oid** equal to 103. Instead of searching for the OID for every customer id that equals 3 the AOC update will instead append and the end of the TBAT a new tuple with the updated value and the latest timestamp, *time2*, effectively saving tremendous time. The **oid** remains the same, but the updated value is inserted and accompanied by the newest timestamp, in this case *time2*.

4 Selection Speed Degradation after AOC Updates

The AOC update is an extremely fast update method when used on column-store databases. Although its method of appending new data to the bottom of the TBAT appendix increasingly creates new data and increases the TBAT's size over time. For a small TBAT this has no noticeable effect, but for a large TBAT with many AOC updates the decrease in selection speed must be alleviated [25].

4.1 Data Cleaning

In order to restore the selection speed of the TBAT after AOC updates data cleaning methods must be used. The intention behind these cleaning methods is to detect the latest version of the updated data and merge it back into the body of the TBAT, the methods are split into two groups, *offline data cleaning* and *online data cleaning*.

Algorithm 1 `merge_update` ▷ offline data cleaning

Input: tbat: the TBAT file to perform data cleaning on

```
1: tbat = merge_sort(tbat, oid)        ▷ merge sort tbat on oid in ascending order
2: tbat_output = new tbat_file                                   ▷ empty TBAT file
3: line1 = tbat.read()                          ▷ read a line from input TBAT file
4: if tbat_output or line1 is NULL then
5:     exit(FILE_ERROR)
6: while TRUE do
7:     line2 = tbat.read()                                        ▷ read next line
8:     if line2 is NULL then
9:         tbat_output.write(line1)
10:        break
11:    if line2.oid>line1.oid then
12:                                          ▷ the tuple with next oid is read in
13:        tbat_output.write(line1)
14:        line1=line2                                          ▷ line1 moves forward
15:    else if line2.timestamp>line1.timestamp then
16:                                      ▷ oids are the same, but line2 is newer
17:        line1=line2                                  ▷ only keep the newer record
18: tbat_output.close()                                ▷ cleaned TBAT file produced
19: return SUCCESS
```

4.2 Offline Data Cleaning

Offline data cleaning, shown in Algorithm 1, is performed after the database is locked from use in order to avoid inconsistent data during the cleaning process. Offline data cleaning first employs a merge sort on the entire TBAT file including the body and appendix, and then deletes the duplicated TBUNs in a sequential manner. This is an effective method but it is not optimized, the challenge is that it takes a large amount of time if many AOC updates have been performed. A more time efficient method is preferred.

4.3 Online Data Cleaning

Online data cleaning fundamentally differs from offline data cleaning in its central idea. This central idea of online data cleaning is to enable employees to continue querying the TBAT while the data cleaning procedure is being performed. The allowance of a continual data stream while the cleaning is being processed is made possible of a data structure called a *snapshot*.

First the online approach takes a snapshot of the body and create a new appendix file linked to the TBAT. The older non-snapshot version of the appendix will be merged into the snapshot of the body using merge sorting and binary searching. At the same time, the TBUN's in the appendix will be written back to the body in the same method as a traditional update on a BAT. Finally after merging is complete the snapshot of the body will replace the original TBAT body and the outdated appendix will be deleted.

In a data restricted environment though the updated data might be to large to fit into main memory, the solution to this is to split the online data cleaning process into two different approaches, the eager approach and the progressive approach.

4.3.1 Online Eager Data Cleaning

The primary idea behind online eager data cleaning, shown in Algorithm 2, is to increase the merging speed by reading the entire appendix of the TBAT into memory.

Algorithm 2 `MERGE_EAGER` ▷ online eager data cleaning

Input: tbat: the TBAT file after AOC updates

```
1: function MERGE_EAGER(tbat)
2:     appendix = tbat.appendix ▷ get the current appendix
3:     if appendix is empty then
4:         exit(NO_NEED_TO_MERGE)
5:     tbat.appendix=new_appendix ▷ create a new empty appendix linked to TBAT
6:     appendix = MERGE_SORT(appendix, oid, ascending, timestamp, descending) ▷ merge sorting the appendix by oid in ascending order and timestamp in descending order
7:     body = snapshot(tbat.body) ▷ make a snapshot of the current body part of TBAT
8:     line1 = appendix.read() ▷ read a line from appendix
9:     while TRUE do
10:        line2=appendix.read()
11:        if line2 is NULL then ▷ end of appendix
12:            BINARY_UPDATE(body, line1)
13:            break
14:        else if line2.oid > line1.oid then
15:            BINARY_UPDATE(body, line1) ▷ only merge the line with the latest timestamp
16:            line1=line2
17:    temp_body=tbat.body ▷ the original body of TBAT
18:    tbat.body=body ▷ TBAT links to the updated body snapshot
19:    delete(temp_body) ▷ purge the original body
20:    delete(appendix) ▷ purge the original appendix
21:    return SUCCESS
1: function BINARY_UPDATE(body, line) ▷ update line to mirror of body using binary search by line.oid
2:     rownum=BINARY_SEARCH(body, line.oid) ▷ search the row number in body containing line.oid
3:     if rownum is NULL then
4:         body.append(line) ▷ append line to the end of body
5:     else
6:         body.update(rownum, line) ▷ update line to the body at the rownum-th line
```

Algorithm 3 `MERGE_PROGRESSIVE` ▷ online progressive data cleaning

Input: tbat: the TBAT file after AOC updates; appendix_queue: the queue of the split appendixes; streaming_update: streaming update input; block_size: the block size of an individual split appendix

```
1: function MERGE_PROGRESSIVE(tbat, appendix_queue)
2:     while appendix_queue is not NULL do
3:         appendix=appendix_queue.dequeue()
4:         MERGE_EAGER(tbat, appendix)
5:     return SUCCESS
```

Once the TBAT is read, online merging is performed into a snapshot of the body. In short, this approach merges the appendix in its entirety at once, and then the merged snapshot replaces the original body in the TABT file.

4.3.2 Online Progressive Data Cleaning

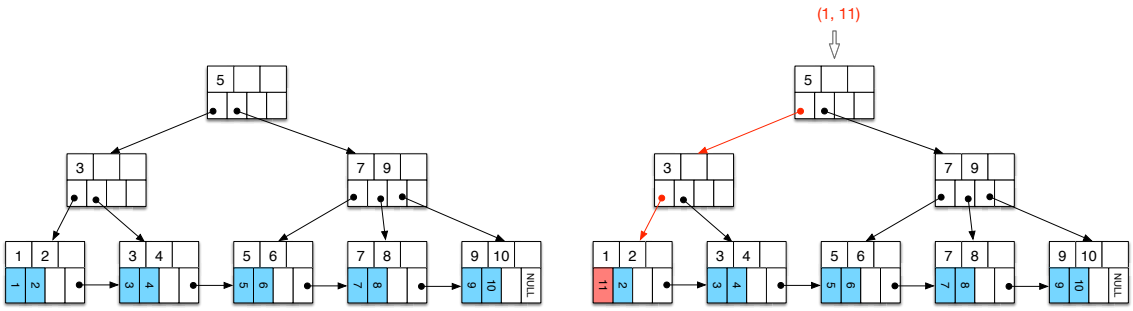
Online progressive data cleaning, shown in Algorithm 3, is tailored for data intensive scenarios where the entire appendix can not fit into memory. Naturally, in these cases the eager approach cannot apply. The primary concept in the progressive data cleaning approach is the appendix queue, each TBAT can contain more than one appendix in this queue. The size of each appendix is titled the *block size* and the administrator for the database must define each of these blocks. He/she must be careful not to exceed the available memory on the system. The original appendix of the TBAT file will be split into separate appendixes according to this specified block size and this *appendix que* will be attached to the TBAT instead of a single appendix like the eager approach.

In practice the progressive approach leverages the eager approach. During

the progressive data cleaning procedure each time an appendix is retrieved from the appendix queue, in order to merge the split appendix with the snapshot of the body the eager data cleaning approach is performed.

5 Offset B$^+$ Tree

In "Hastening data retrieval on out-of-core column-store databases using offset B$^+$ Tree" a new data structure was introduced. This new data structure is known as the *Offset B$^+$ Tree* or by its shorter title *OB-tree* and is a variant of a B$^+$ Tree. The OB-tree was developed specifically to work with the TBAT data structure and has several unique and notable features that leverage the TBAT. Firstly the OB-tree has a succinct data structure, this means that it can be easily adopted by existing column-store databases. Secondly, the OB-tree's index is sparse and only holds the updated TBAT records, moreover the OB-tree can be stored in main memory or serialized on the hard disk to be retrieved later. When the OB-tree is stored in main memory the retrieval speed is orders of magnitude faster. Lastly, and most unique, the OB-tree permits the storage of duplicated keys. Specifically the key in the OB-tree is the **oid** in the TBAT [24].



(a) A Simplified OB-Tree Example (b) OB-Tree with Duplicate Keys Inserted

Figure 5: OB-Tree

5.1 OB-Tree Data Structure

In the OB-tree structure, shown in Figure 5(a) there are two categories of nodes, internal and leaf nodes. The topmost node is the root node, but it is considered a leaf node if the OB-tree has only one layer and an internal node if the OB-tree has multiple layers. Every OB-tree has a parameter that determines the layout of each and every node inside, this parameter is labeled n and is integrated in a few key ways.

- Each internal node will have space for n search keys, these search keys are the **oid**s from the TBAT. Moreover $n + 1$ pointers will be used to point to other nodes in the tree to aid in quick traversal to adjacent nodes. At a minimum $[(n + 1)/2]$ of the pointers are used. If the internal node is a root node it is only required that at least 2 pointers are used regardless of n's value.

- Each leaf node has enough space allocated for n search keys. Although among the $n + 1$ node space only a single allocation is used to point to the next leaf node in the sequence. This pointer allocation is the right-most node and all of

the other n unites to the left of this pointer are reserved space for the OB-tree's special *offset* value, which is used to point to the location of the updated record in the TBAT's appendix.

- The method of assignment of each **oid** to each node is the same as in a B^+ Tree. First a search is performed to determine the node that the **oid** should be inserted into, if the node is not full add the **oid**, if the node is full split the node and allocate a new leaf node filling it with half of the previous node's **oids**. The final step is the insertion of the new **oid**.

An *offset* in the OB-tree is a scalar recording of the relative location of an updated record inside an appendix that has been appended to the end of the body of the TBAT. Safely, we can assume that the number of lines of the body is l_b and the number of lines of the appendix is l_a. For any given record located at the kth line, $1 \leq k \leq l_a$ of the appendix, the offset associated with this updated record is k. Naturally the updated record with offset k is located at the offset plus the length of the TBAT body, in other words $(l_a + k)^{\text{th}}$ line of the TBAT.

It is important to note that the space cost of the offset is relatively smaller than that of a pointer in a traditional operating system, think of the offset as a simplified method of eliminating pointers. Since pointers can occupy additional spaces inside any B^+ Tree, we use this scalar offset to replace most of the pointers at the leaf nodes. The actual location of a record is calculated by the definition of offset mentioned earlier, that is the length of the body plus the offset. The offset's data

Algorithm 4 OB-Tree Loading on Cold Data

Input: tbat: TBAT file
Output: obtree: OB-tree

```
 1: procedure OB-Tree-Load-Cold(tbat)
 2:     obtree = new root( )                          ▷ create a root node
 3:     file = tbat.open( )                           ▷ open the TBAT file
 4:     file.seekToAppendix( )                        ▷ seek to the appendix
 5:     offset=1
 6:     while file.next != EOF do
 7:         line = file.read_next_line( )
 8:         oid = line.OID;
 9:         obtree.insert(oid, offset++)
10:     file.close( )
11:     return obtree
```

type is user selectable, typically it's chosen to be the same data type as the **oid**, that is a 4 byte or 8 byte integer. With computing, space is always a concern, and in the case of the offset this is taken into account. In order to save more space smaller bytes of the integer data type may be used for the offset.

5.2 Bulk Loading OB-Tree

In order to load the TBAT appendix into the OB-tree bulk loading is necessary. Bulk loading takes the **oids** and offsets from the TBAT appendix and loads them into the OB-tree. Before this loading can occur the current state of the TBAT must be determined, this will decide if the data is in "cold", or "hot" condition, and will determine the appropriate method of loading the data[24].

5.3 Bulk-Loading on Cold Data

If the appendix of the TBAT is not currently in use it is considered to be in "cold" condition, at this point Algorithm 4 is used to load the OB-tree with the updated records. Algorithm 4 reads from the appendix of the TBAT (remember, the appendix of the TBAT starts from the first updated record directly under the TBAT body) and it loads the **oid**s and offsets into the OB-tree starting at one for the offset. Naturally the offset increases by one for every **oid** insertion.

5.4 Bulk-Loading on Hot Data

Now that the procedure for loading a "cold" appendix has been explained, the next procedure to examine is the loading of the OB-tree when the TBAT is currently in use, or in "hot" condition. In this condition there is a high probability that a new update will be inserted into the TBAT appendix. The loading solution for this instance is the use of a hot-data buffer that acts as a temporary appendix for the new records being loaded into the TBAT. In this hot condition the existing already loaded TBAT appendix is considered to be "cold" and is loaded using Algorithm 4.

6 OB-Tree Selection

The OB-tree has the ability to perform two kinds of searching, ad-hoc and range searching. Ad-hoc searching is when a single **oid** is given and the intention is to

Algorithm 5 TBAT Search with OB-Tree Index

Input: tbat: TBAT file, obtree: OB-tree, oid: OID

```
1: procedure TBAT-SEARCH-OB(tbat, oid)
2:     tbat.open( )
3:     offset = obtree.searchOffset(oid)
4:     if offset != NOT_FOUND then
5:         tbat.seek(num_of_lines_body+offset-1)
6:         return tbat.readLine( )
7:     else
8:         return tbat.binarySearchBody(offset)
```

return to the user the corresponding **offset**. With Range searching a range of **oids** are given and the intention is to return the range of offsets corresponding to the **oids**. It is important to note that for both searching types the OB-tree follows the same methods, exactly like how a B^+ Tree handles ad-hoc and range searches [24]. For the best result with range searches it's recommended that the **oids** are arranges into ranges, this reduces search time on the OB-tree and read time on the TBAT file.

6.1 TBAT Search Using OB-Tree

Once an OB-tree is created it can be utilized to quickly search the corresponding TBAT file, the procedure for doing so is straight forward. First a **oid** is selected, the algorithm shown in Algorithm 5 searches the OB-tree to determine if the record with the selected **oid** has been updated previously. If the **oid** is found in the OB-tree, then an offset can be retrieved enabling the direct reading of the record in the appendix. The calculation of the line number of the updated record in the appendix is calculated as

target_line_number = num_of_lines_body + offset

num_of_lines_body is the representation of the total number of lines in the body of the TBAT file, and the offset is used to determine how far into the appendix must be searched to insert the record. It's important to note that if the **oid** is found in the OB-tree it means that it has already been updated. In this case the procedure is a simple binary search conducted on the body of the TBAT.

7 Experiment Results

The architecture of the OB-tree allows it to be used as an index to increase selection speeds when paired with the TBAT. It is expected that searching with an OB-Tree will be more efficient than standard linear search through the appendix of the TBAT. The experiments simulate three different selection techniques in order to compare selection on a TBAT file's appendix using a linear search, selection on a TBAT utilizing an OB-tree and traditional selection on a BAT after it's been updated. The experiments were executed using Java code and shell script, and the log files of all of the data presented in this work can be found in my BitBucket repository [1].

[1] https://bitbucket.org/tjmatacic/column-store-tbat-2016

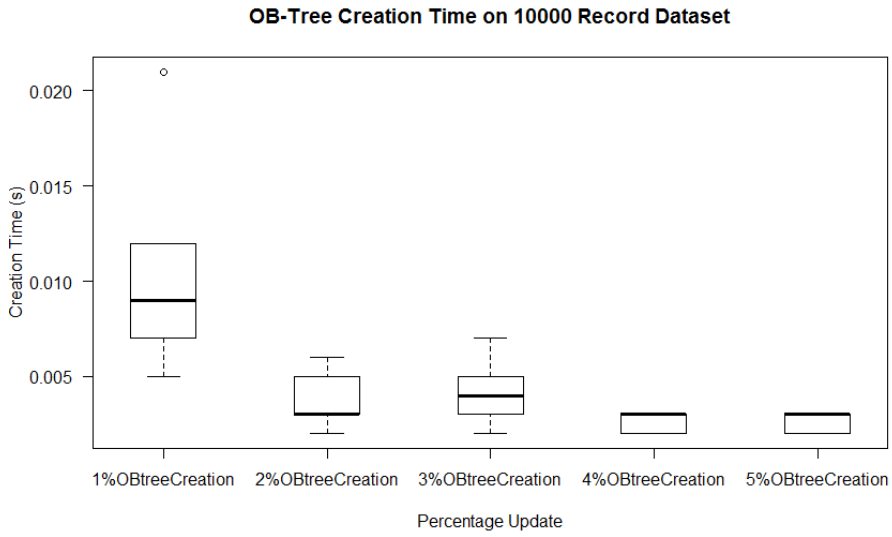


Figure 6: OB-Tree Creation Time 10000 Record Dataset

7.1 Creation time OB-Tree

Before examining the results of the selection experiments, it would be appropriate to showcase the impressively fast creation time of the OB-trees themselves. Figure 6 depicts the OB-tree creation time on the 10000 record dataset. The fastest creation time is an incredible 0.002 seconds occurring on 2%, 3%, 4% and 5% updates. The slowest creation time is 0.021 seconds and the mean creation time is 0.0048 seconds.

Figure 7 depicts the OB-tree creation time on the 10MB dataset. The fastest creation time was on the 1% update and it clocked in at a mere 0.066 seconds, the mean creation time is 0.7282 seconds, and the slowest time is 1.743 seconds.

Figure 8, the final figure for OB-tree creation time, depicts creation time on a 64MB dataset. Even though this dataset is significantly larger than the 10MB dataset its OB-tree creation time is impressive. The fastest being 2.582 seconds, the

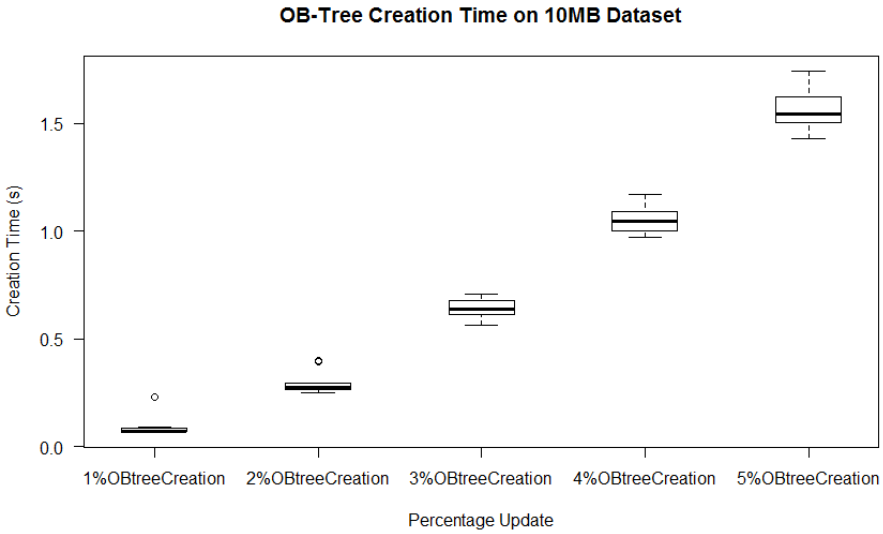


Figure 7: OB-Tree Creation Time 10MB Dataset

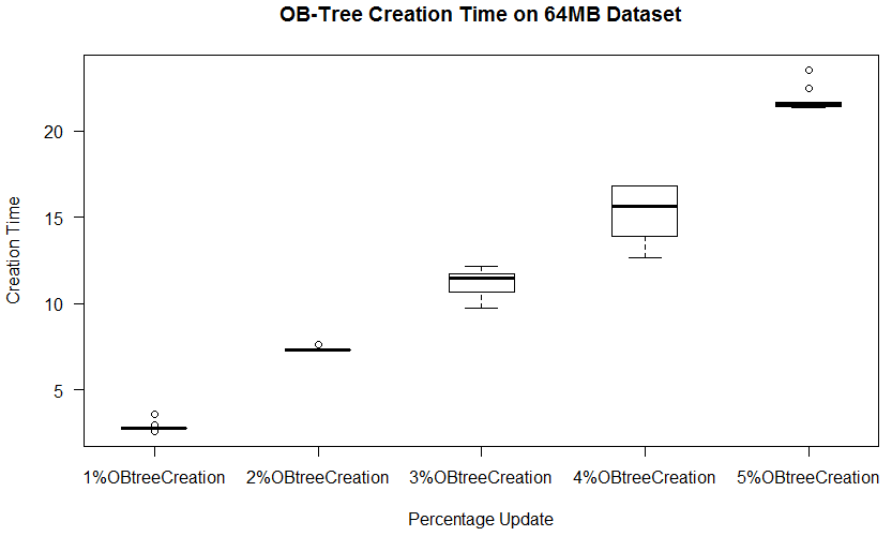


Figure 8: OB-Tree Creation Time 64MB Dataset

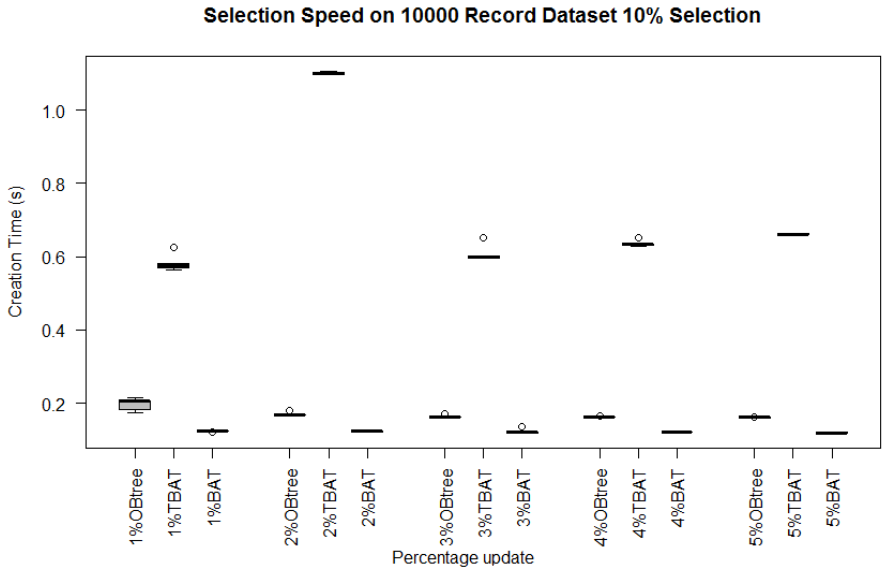


Figure 9: Selection Speed Experiment 10000 Record Dataset 10% Selection

slowest being 21.657 seconds and the mean is 11.70702 seconds.

7.2 Tests of Searches on TBAT and BAT

The selection experiment is conducted on a Ubuntu Linux 16.04 release virtual machine created on a Dell PowerEdge T630 in the "Sarah Cloud" of the YSU Data Lab. The experiment times are recorded in wall time for accuracy. The preliminary experiment is conducted on a 10,000 record dataset, and tested the selection time on 10% of the original created records. Figure 9 shows the results of these selection types. The figure is grouped by selection type and update percentage from 1% to 5%. The mean selection time for the TBAT with OB-tree index is 0.17008 seconds, the mean selection time for the TBAT without an OB-tree index is 0.61288 seconds, and the mean time for the fully updated BAT using the traditional method is 0.12172

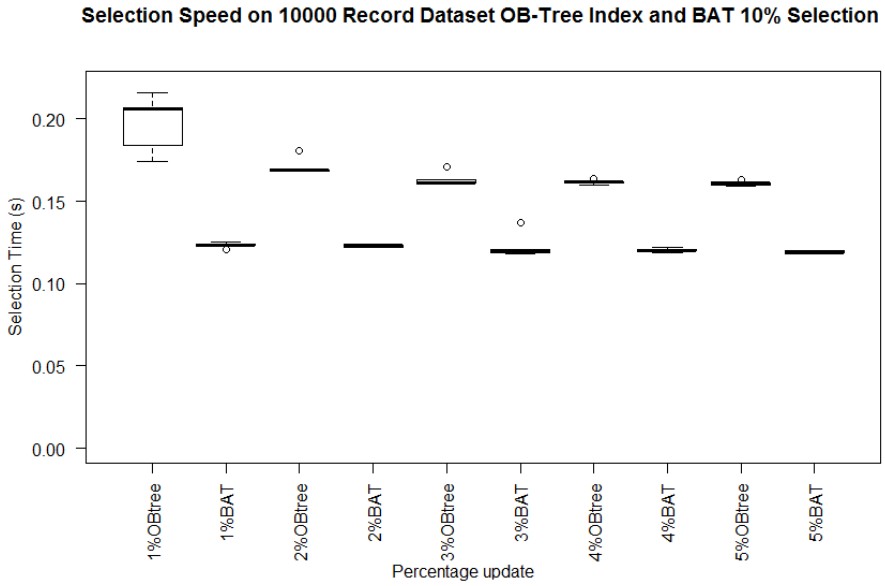


Figure 10: Selection Speed 10000 Record Dataset OB-Tree Index and BAT 10% Selection

seconds. This result clearly shows that the TBAT utilizing the OB-tree index is more than three times as fast as the TBAT utilizing the traditional **oid** search.

If we look closely at Figure 10 we can clearly see that the range from the upper quartile to the lower quartile is very small for the OB-tree search. Moreover as the selection percentages increase the performance of the OB-tree indexed TBAT stays consistent. Mean time for 1% selection is 0.194, for 2% is 0.171, for 3% is 0.163 seconds, for 4% is 0.161 seconds and finally for 5% is 0.160 seconds. This demonstrates the reliability and consistency of the OB-tree search. What performance does the OB-tree demonstrate when a larger dataset with the addition of a 20% selection percentage is experimented on?

In Figure 11 the selection experiment is conducted on a 10MB dataset, to retain consistency the update percentages were 1%, 2%, 3%, 4%, and 5% the same as

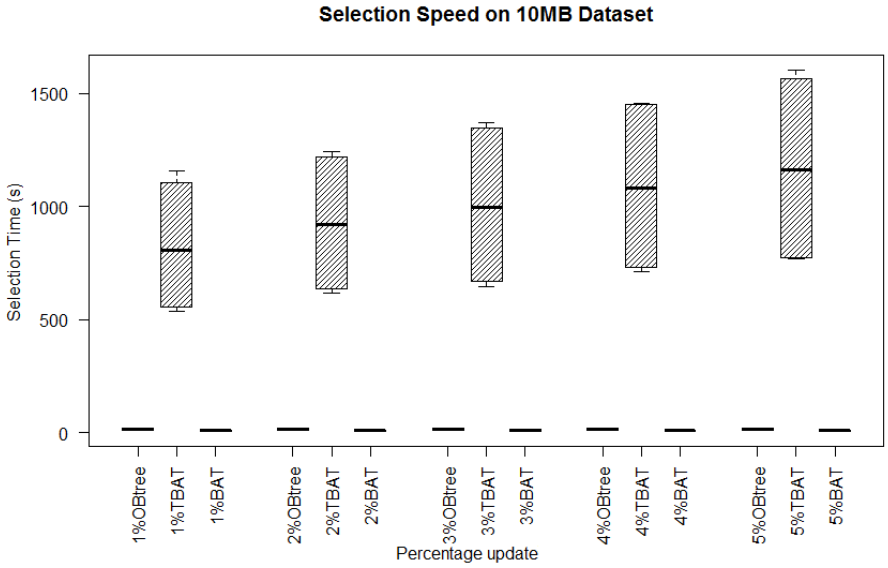


Figure 11: Selection Speed 10MB Dataset

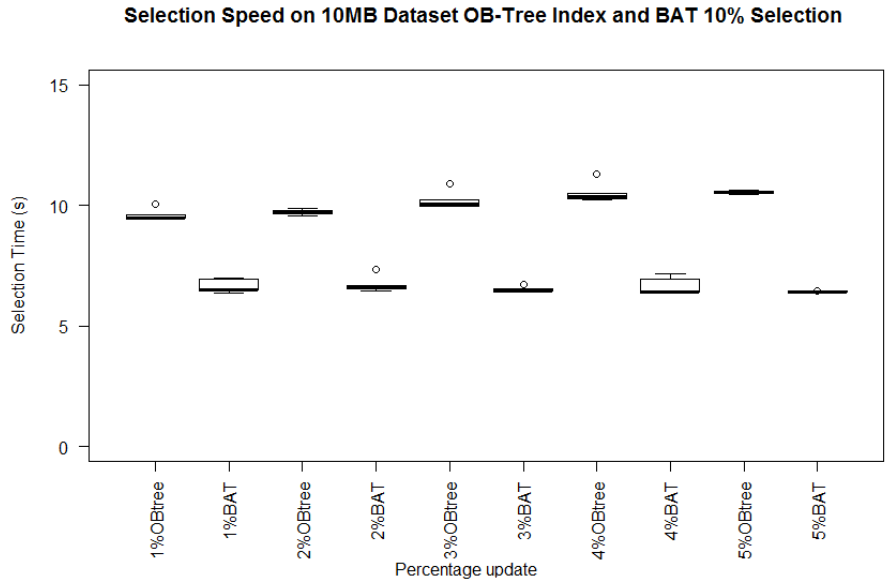


Figure 12: Selection Speed 10MB Dataset OB-Tree Index and BAT 10% Selection

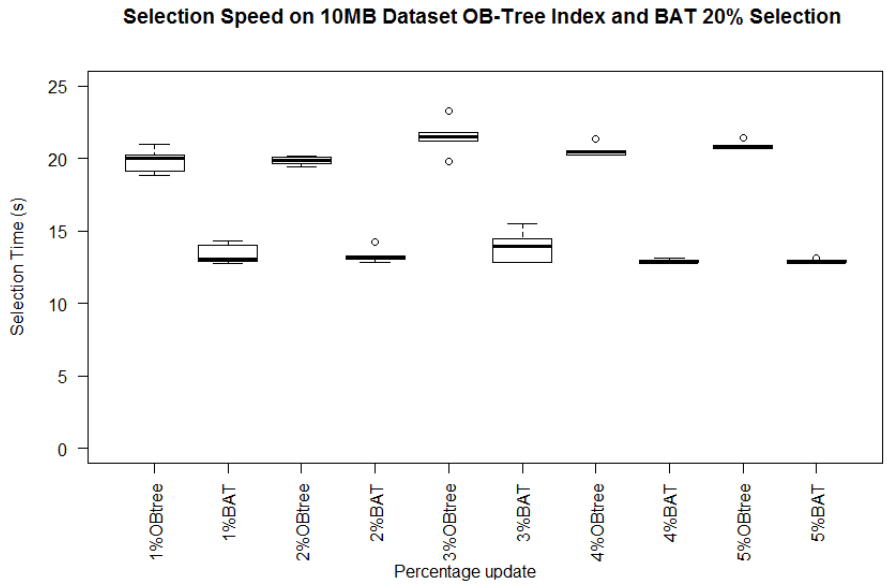


Figure 13: Selection Speed 10MB Dataset OB-Tree Index and BAT 20% Selection

the 10,000 record experiment. Both 10% and 20% selection percentage results were aggregated together for readability (the next figures examine both update percentages separately and in greater detail). The performance gap between the TBAT with an OB-tree index and a TBAT without an index becomes apparent with this larger dataset. The mean selection time for the TBAT with OB-tree index is 15.32 seconds, the mean selection time for the TBAT without an OB-tree index is 1002.58 seconds, and the mean time for the updated BAT is 9.94 seconds. This clearly demonstrates how quickly the performance gap widens between the TBAT using the OB-tree index and the TBAT without the index.

For greater clarity on the 10MB selection experiment Figures 12 and 13 are included to show the separate performance of the OB-tree indexed TBAT on 10% selection and 20% selection. The slowest search with the TBAT utilizing the OB-tree index is a mere 23.257 seconds, and the fastest search is 9.442 seconds. In great

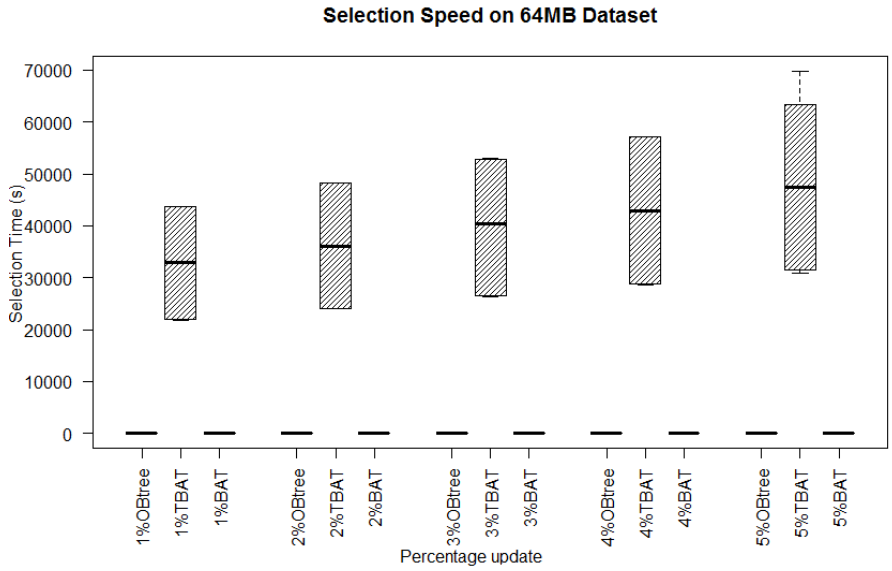


Figure 14: Selection Speed 64MB Dataset

contrast, the fastest search on the un-indexed TBAT utilizing the traditional linear search is 546.248 seconds, this means that the OB-tree indexed TBAT is more than 50 times faster than the un-indexed linear search.

In Figure 14 the final selection experiment is conducted on a 64MB dataset. This dataset size was chosen in order to mimic a data block in the Hadoop framework, the gold standard for processing large stores of data. Like the previous tests, this figure shows the aggregated selection percentages for 10% and 20%, and the update percentages were 1%, 2%, 3%, 4%, and 5%, the results are impressive. The aggregated mean selection time for the TBAT with OB-tree index is 121.8882 seconds, the mean selection time for the TBAT without an OB-tree index is 39968.18 seconds, and the mean selection time for the fully updated BAT using the traditional method is 72.27496 seconds. This impressive performance shows that the TBAT utilizing the OB-tree index is exponentially faster than the TBAT without the index, moreover

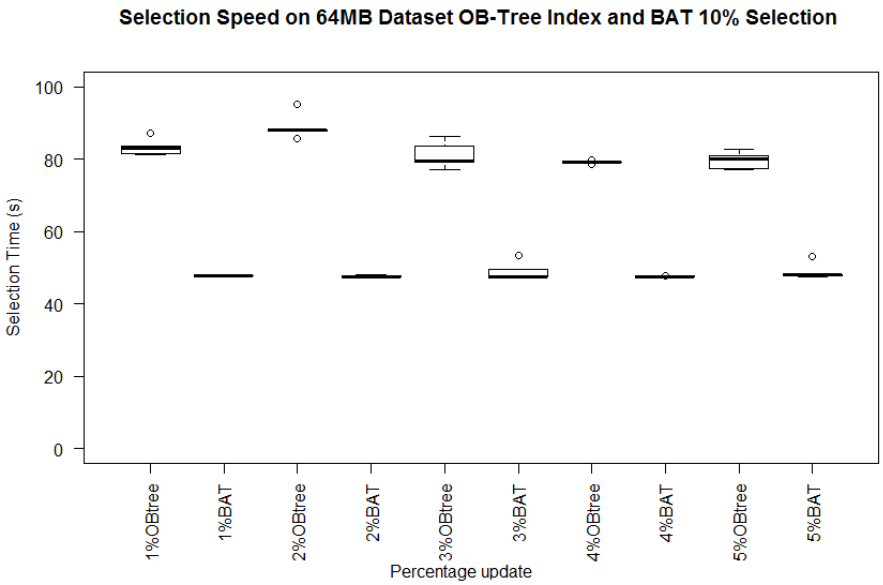


Figure 15: Selection Speed 64MB Dataset OB-Tree Index and BAT 10% Selection

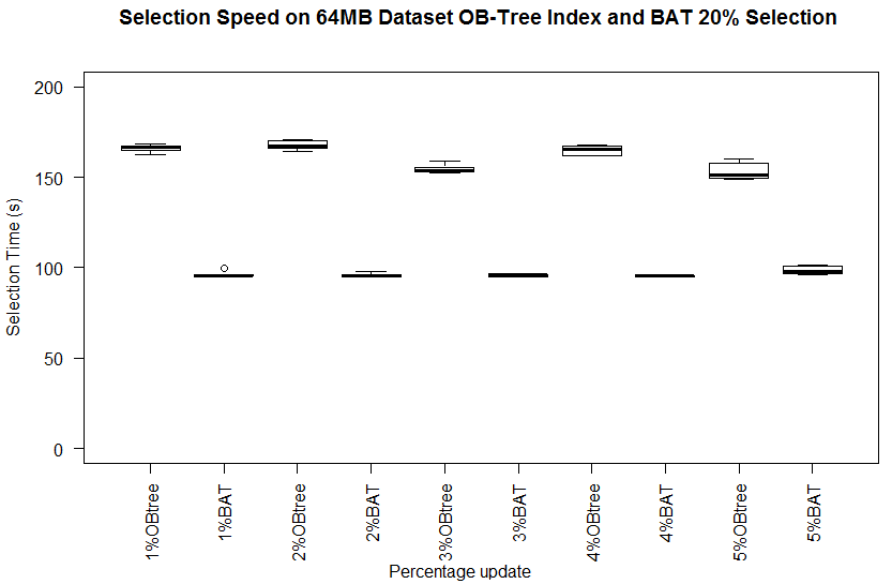


Figure 16: Selection Speed 64MB Dataset OB-Tree Index and BAT 20% Selection

Table 1: Mean Selection Times (s) For TBAT, TBAT with OB-Tree and BAT

Dataset	Selection (%)	TBAT	TBAT with OB-Tree	BAT
10000 Records	10%	0.61288	0.17088	0.12172
10MB	10%	670.0837	10.12456	6.59756
10MB	20%	1335.094	20.53524	13.2966
64MB	10%	26671.67	82.41064	48.16828
64MB	20%	53264.7	161.3658	96.38164

the OB-tree indexed TBAT retains the impressive consistency that it showed in the previous 10000 record test with mean selection times ranging from 128.30 to 116.54, a range of no more than 8 seconds.

A closer look at Figures 15 and 16 demonstrates the continued impressive performance of the OB-tree index. Between 10% and 20% selection percentages the fastest OB-tree indexed selection time is 78.611 seconds and the slowest is 170.722 seconds. These results are striking considering the volume of data the OB-tree successfully searched. In continuing the consistency trend of the OB-tree, the 20% selection results showed no outliers. This final experiment truly shows the lightning fast selection speed of the OB-tree indexed TBAT, the fastest TBAT search without the OB-tree index on the 64MB dataset is 22169.63 seconds, meaning that the OB-tree indexed TBAT is an incredible 282.01 times faster than the un-indexed sequential search.

As a summary all of the mean selection times of the selection experiments have been included in Table 1 and separated by dataset, selection percentage, and selection type. By examining the table we can clearly see that the OB-tree indexed TBAT is orders of magnitude faster than the un-indexed TBAT. Even though the

BAT was sorted and searched using a binary search, it was only marginally faster than the OB-tree, an amazing feat considering the unsorted nature of the data the OB-tree was tasked to search.

8 Conclusion and Future Works

In this work, we proposed a novel index, called Offset B^+ Tree (OB-Tree), on column-store databases based on the Timestamped Binary Association Table (TBAT) storage model. OB-tree is a succinct index structure that works well on TBAT data with many updated records. It was shown that the selection speed on a TBAT with an OB-tree index is significantly improved to exponentially faster than TBAT selection without an OB-tree index. In addition, without the need of time-consuming data cleaning process, the selection speed on TBAT with an OB-tree index is nearly the same as the selection speed on a cleaned Binary Association Table (BAT). Future work will expand upon the generalization of OB-tree for more functionalists on TBAT, such their use with data cleaning. We will also investigate the application of the OB-tree on big data and methodologies to increase in the speed of OB-tree creation.

References

[1] Abadi, D., Boncz, P., and Harizopoulos, S. Column-oriented database systems. *Proceedings of the VLDB Endowment* (2009), 1664–1665.

[2] Abadi, D., Boncz, P., Harizopoulos, S., Idreos, S., and Madden, S. The design and implementation of modern column-oriented database systems. *Foundations and Trends Vol. 5*, No. 3 (2012), 197–280.

[3] Abadi, D., Madden, S., and Ferreira, M. Integrating compression and execution in column-oriented database systems. *SIGMOD '06* (2006), 671–682.

[4] Abadi, D., Madden, S., and Hachem, N. Column-stores vs. row-stores: how different are they really? *SIGMOD '08* (2008), 967–980.

[5] Abadi, D. J. *Query execution in column-oriented database systems*. PhD thesis, Massachusetts Institute of Technology, 2008.

[6] Ailamaki, A., DeWitt, D., and Hill, M. Data page layouts for relational databases on deep memory hierarchies. *VLDB Best papers* (2001).

[7] Ailamaki, A., DeWitt, D. J., Hill, M. D., and Skounakis, M. Weaving relations for cache performance. In *VLDB* (2001), vol. 1, pp. 169–180.

[8] Carstoiu, D., Lepadatu, E., and Gaspar, M. Hbase-non sql database, performances evaluation. In *in Computer Science (1986), Master in Computer Science (1990), and PhD in Computer Science* (2010), Citeseer.

[9] Copeland, G., and Khoshafian, S. A decomposition storage model. *SIGMOD '85 Proceedings of the 1985 ACM SIGMOD international conference on Management of data* (1985), 268–279.

[10] Copeland, G., Khoshafian, S., Jagodits, T., Boral, H., and Valduriez, P. A query processing strategy for the decomposed storage model. *IEEE* (1987).

[11] Ferreira, M. C. *Compression and query execution within column oriented databases*. PhD thesis, Massachusetts Institute of Technology, 2005.

[12] George, L. *HBase: The Definitive Guide*. O'Reilly Media Inc., 2011.

[13] Harizopoulos, S., Liang, V., Abadi, D., and Madden, S. Performance tradeoffs in read-optimized databases. *VLDB '06 Proceedings of the 32nd international conference on Very large data bases* (2006).

[14] Inkster, D., Zukowski, M., and Boncz, P. Integration of vectorwise with ingres. *ACM SIGMOD Record 40*, 3 (2011), 45–53.

[15] Krueger, J., Grund, M., Tinnefeld, C., Plattner, H., and Zeier, Alexander Faerber, F. Optimizing write performance for read optimized databases. *Lecture Notes in Computer Science* (2010), 291–305.

[16] Lamb, A., Fuller, M., Varadarajan, R., Tran, N., Vandiver, B., Doshi, L., and Bear, C. The vertica analytic database: C-store 7 years later. *Proceedings of the VLDB Endowment 5*, 12 (2012), 1790–1801.

[17] StoneBracker, M., Abadi, D., Batkin, A., Chen, X., Cherniack, M., Ferreira, M., Lau, E., Lin, A., Madden, S., O'Neil, E., O'Neil, P., Rasin, A., Tran, N., and Zdonik, S. C-store: a column-oriented dbms. *VLDB* (2005), 553–564.

[18] Tsirogiannis, D., Harizopoulos, S., Shah, M., Wiener, J., and Graefe, G. Query processing techniques for solid state drives. *SIGMOD* (2009), 59–72.

[19] Vora, M. N. Hadoop-hbase for large-scale data. In *Computer science and network technology (ICCSNT), 2011 international conference on* (2011), vol. 1, IEEE, pp. 601–605.

[20] Watson, V. Run-length encoding, May 10 2002. US Patent App. 10/143,542.

[21] Yaman, S. Introduction to column-oriented database systems (nov 2012).

[22] Yu, F., and Hou, W.-C. A framework of write optimization on read-optimized out-of-core column-store databases. *6th International Conference, DEXA 2015* (2015), 155–169.

[23] Yu, F., Hou, W.-C., Luo, C., and Jones, E. Asynchronous update on out-of-core column-store databases utilizing the timestamped binary association table. *CAINE-2014* (2014), 215–221.

[24] Yu, F., and Jones, E. Hastening data retrieval on out-of-core column-store databases using offset b+-tree. *Proc. CAINE'15* (2015), 313–318.

[25] Yu, F., Matacic, T., Xiong, W., Hamdi, M. A., and Hou, W.-C. Data cleaning in out-of-core column-store databases: An index-based approach. *IKE* (2016), 16–22.

[26] Zukowski, M., and Boncz, P. From x100 to vectorwise: Opportunities, challenges and things most researchers do not think about. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (2012), ACM, pp. 861–862.

[27] Zukowski, M., van de Wiel, M., and Boncz, P. Vectorwise: A vectorized analytical dbms. *Data Engineering (ICDE), 2012 IEEE 28th International Conference* (2012).