Hastening Write Operations on Read-Optimized Out-of-Core Column-Store Databases Utilizing Timestamped Binary Association Tables

by

Eric S. Jones

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

May, 2015

Hastening Write Operations on Read-Optimized Out-of-Core Column-Store Databases Utilizing Timestamped Binary Association Tables

Eric S. Jones

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:

Eric S. Jones, Student Date

Approvals:

Dr. Feng Yu, Thesis Advisor Date

Dr. John R. Sullins, Committee Member Date

Dr. Yong Zhang, Committee Member Date

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

Acknowledgments

I would personally like to thank my family for supporting me all the way through this graduate degree program. Without their love and support this part of my life would've been made more difficult. I would like to thank my fiance, Nawal, who always stood by me and encouraged my success. I would like to thank my professors who taught me so much to prepare me for industry life. Lastly, I would like to thank Dr. Feng Yu, my adviser, who believed in me enough to take me as his graduate assistant, for without him I would not have grown nearly as wise or prepared for my future career.

Abstract

The purpose of this thesis is to extend previous research on Out-of-Core column-store databases. Following use of the Asynchronous Out-of-Core update, which kept track of data using timestamps, an appendix is created which holds the newest timestamps and updated data by appending entries to the tables as new tuples. The appendix is naturally unsorted and unindexed by nature, causing need for a linear search that is not only slow, but causes ever-increasing query time as the volume of data within the appendix expands. Although measures exist to merge the appendix with the original body of the data, which is sorted and indexed, it only makes searching on the data swifter once the merging of tuples is complete. For this reason, the use of an offset B-Tree index to allow for more efficient searches on the appendix is proposed.

Contents

List of Figures

List of Tables

1 Introduction

Column oriented database systems (column-stores) have existed since the 1970's. Although their use has caught on slowly, recent advances in hardware have made them an efficient and viable option for data warehouse and other read-heavy workloads. Given the physical layout of a column-stores, read optimizations are simple, because a query will only retrieve the attributes necessary, and rarely will any query require all attributes in a table. Optimizing a column-stores' write operations can be more challenging, given that a single tuple can be spread across multiple blocks or even pages on disk. The efforts of this research seek to continue advancing write optimizations by making reads following an Asynchronous Out-of-Core update more feasible through use of offset B-tree indexes.

The methods discussed later in the research are an attempt at replacing the linear search following updates, thereby making use of column-stores a more well-rounded approach. Section 2, Background on the Column-Store Database, discusses general applications of column-stores and their various optimizations. Section 3, OOC Update Optimization, takes a look at past research in using timestamped binary association tables for write optimizations. Section 4, Experiment Results, takes a look at how well the indexed approach with offset B-trees performed when compared to the old linear-search method. Lastly, Section 5, Conclusion and Future Works, will give a brief summary of the findings and layout potential plans on future research topics. All methods used in the paper focus on out-of-core column-stores.

2 Background on the Column-Store Database

Modern column-store applications are able to take advantage of a plethora of efficiency increases unavailable to older column-oriented systems. Abadi [1] documents many proficient increases in query optimizations that make modern column stores faster than row-stores under nearly any condition. One of the features available and most easily optimized for database systems is the use of different materialization strategies. Materialization is the building of result (intermediate or final) tuples that pass the query predicate(s).

A classic row-store system will construct intermediate tuples after each predicate (query condition) has been passed. This means that the query engine must construct unnecessary results and then apply more predicates to these intermediate results. It is, in many cases, an inefficient waste of time. There now exists the ability to do "late" materialization; "late" because the query will only build result tuples when the finalized product is available. In most cases this saves on time and allows for a much faster query.

The way in which the query engine keeps track of the results is by storing the object-identifiers (OIDs) in memory [10]. Sometimes these are referred to as row-identifiers or tuple-identifiers. Database files are usually stored as dense indexes which keep track of the entire set of tuples, or the location of the elements belonging to some attribute of a table. It will keep track of the record in format: record-1, record-2, ...record-n, and will contain all relevant database information for that attribute in a column-store. Intermediate results contain a subset of this dense index, called a sparse index. A sparse index contains those tuples or elements that pass the predicate(s). If the query optimizer is using a late-

materialization strategy, only the OID(s) necessary for the other query predicates are kept in memory. In most cases this adds to a great increase in efficiency because keeping track of which OIDs are needed, and therefore which data values will later be extracted, is simpler than constructing new tuples every time a predicate is passed. Late materialization, factored in with less data to be read by only accessing relevant attributes, can lead to several orders-of-magnitude efficiency increases for column-stores over row-stores.

Compression on column-stores allows for greater use of in-memory database systems. By keeping columns in memory, the query can be further enhanced by not needing to take data off of disk. The format of column-stores further makes this a possibility when paired with compression algorithms like Lempel-Ziv (LZ), which can further decrease the amount of data to be stored, making in-memory database systems more easily achieved. According to Plattner's research [9], the average compression rate for column-stores is a factor of 10 higher than row-stores. Queries can be sped up by several orders of magnitude for the I/O savings received from transferring a set of data several orders of magnitude smaller (in its compressed state) off of disk.

LZ works by encoding the data with a pointer or dictionary to a previous occurrence of the same value [8]. Therefore, if the DBA uses LZ to encode a name attribute, every occurrence of the name "John" (except the first) will be compressed into a pointer to the original appearance of the name. In a large column-store database where certain values can be repeated multiple times at various points in the entries, this algorithm will greatly reduce the amount of space needed. LZ compressed data usually appears as triples with one of the fields of the triple being an offset, which will show any matches made in

the dictionary window, otherwise a “0” will take the place of non-match. [7] makes use of a block dictionary window as opposed to a sliding window by dividing input streams into equal size blocks, and the block can be accessed directly to only decode the parts of the blocks needed. By using a cached-results buffer, the join results can be stored later to speed up queries once more. Since LZ decoding is a procedure that is swiftly done, the join operation can be done quickly to allow it to operate on compressed data, especially when later combined with the cached-results buffer.

Modern CPU systems can aid in the encoding of data to make writes more efficient. Parallel algorithms exist which can use multiple processor cores, using their own local cache memory to encode the data [6][5] can make it faster with a small sacrifice to general compression ratio, which isn’t consequential given today’s cheap price for storage.

Column-Stores are also able to make better use of general compression algorithms because they can make use of column-specific compression schemes. In row-stores the compression would have to factor into the entire tuple, and if the tuples are variable in size it can severely limit the amount of compression applied. Since columns are stored separately, a column for "age" can benefit more than if it were stored alongside non-numerical data types. If a table or view is sorted by some attribute it can make it even more compressible in certain cases.

Run-Length Encoding (RLE) [2], combines multiple consecutive elements of the same value into a single value by storing a triple; the start position, value, and how many values are in the run. If a “people” table were to be sorted by age, then the age column

could store only a single value for each specific age, no matter how many people it stored. Three-thousand entries for age 20 would be stored as one triple, showing the start position, 20 (age value), and 3000 for the number of runs. An additional type of compression, called bit-vector encoding, which is best used for a small range of possible values, such as the provinces or states of a given nation. The values encoded will be assigned a binary value and repeats of that value are replaced throughout the column by that value.

The two algorithms mentioned above, RLE and bit-vector encoding, are what are known as "lightweight" compression algorithms, compared with LZ which is a "heavy-weight" compression algorithm. LZ is more versatile and able to handle a wider range of data values than either RLE or bit-vector encoding. It does, however, have a drawback in that LZ is unable to do operations directly on compressed data. In [11] there is displayed a sequential algorithm for LZ encoding as well as several advanced parallel (PLZ) encoding techniques which, although slightly reducing compression ratio, add to the swiftness of the encoding and decoding of the data. This makes swift decompression a possibility allowing for LZ to remain a contender in column-store applications and helps in query optimization. To overcome the handicap LZ suffers in comparison to RLE, [7] has come up with a method called the "LZ join" which allows for joins to take place directly on compressed data. LZ join is a late-materialization strategy which performs its joins as late in the query-plan as possible to speed up the query.

The algorithm used in [4] was specifically LZOP, a variant of LZ which allows for greater decompression speeds. It is essentially a performance optimized version which allows for greater use in real-time systems and makes the use of LZ in a DBMS far

more viable, especially in the case of column-store analytical workloads. There is a sacrifice of some compression space involved, however, it is well worth the significant gains received in the reductions of CPU overhead costs. [10] shows the results of various tests run using different algorithms which awards LZOP fastest compression in multiple data categories. Data blocks compressed in the traditional LZO variant are stored in sets of sliding dictionaries called 'matches' which help to quickly identify query results during standard decompression.

For distributed systems compression can also play a big role. In [3], Douglis states that compression algorithms can greatly aid in tackling the problem of bandwidth bottlenecks in situations of resource contention. In database systems, as well as other systems, reducing the amount of data consuming bandwidth over a LAN or WAN connection can speed up queries. The reason being is that if data-transfer can occur more quickly, than the query can finish more swiftly. More nodes in the distributed system may share information more readily and swiftly in the presence of compression algorithms. The data in this compressed format can even be more likely to fit into the cache, allowing for greater reuse of queries and better propagation of queries over the network which thereby can help to avoid I/O as much as possible.

Modern hardware advances make column-stores more feasible and more efficient for a wide range of applications. In [1] it was mentioned that the SIMD applications on processors can allow a single processor core to make use of multiple threads and execute operations on up to 4 values with a single CPU instruction. Despite the file structure of column-stores making writes more difficult, encoding the data with compression algorithms

can be done at up to four times the speed, aiding in insertion of data. Added with the fact that advances in column-stores have allowed for direct operation on compressed data, (which again aids in late materialization strategies) this has allowed for huge gains in query optimization. If a compression ratio of 2 is achieved this means that the I/O savings can be enhanced by reading through 8 values at once. Since it was mentioned above that column-stores achieve far greater compression ratios, this number in reality is likely to be far higher, even in near worst-case scenarios.

(a) Row-Based Table `customer`

id	name	balance
1	Alissa	100.00
2	Bob	200.00
3	Charles	300.00

(b) BAT `customer_id`

oid	int
101	1
102	2
103	3

(c) BAT `customer_name`

oid	varchar
101	Alissa
102	Bob
103	Charles

(d) BAT `customer_balance`

oid	float
101	100.00
102	200.00
103	300.00

Figure 1: `customer` Data in Row-Based and Column-Store (BAT) Format

3 OOC Update Optimization

In general, column-store databases work well in data warehouse environments given their read optimizations. Typically a column-store's tuples are BUNs (Binary UNits) that are composed of pairs; Object Identifiers (OIDs) and attribute-values. OIDs are required to keep track of tuples that are decomposed into BUNs. A grouping of BUNs is referred to as a BAT (Binary Association Table), and the OID is used to keep track of attribute values spread across different BATs. Figure 1 shows a visual representation of a three-attribute customer table stored as a set of BATs.

One of the main problems with column-stores is that write optimizations can be very difficult. The challenge stems from the physical storage model used. Since row-stores physically store their tuples one after another on disk, the tuples are neatly arranged, making writes a straight-forward and simple task. In the case of column-stores, however, the physical storage of a tuple is decomposed over several BATs and writes require accessing several different locations on disk. The multiple ad-hoc accesses over disk can add greatly

to I/O time, making any write a difficult and potentially time-consuming procedure. Even performing an update on tow attributes requires accesses to different storage-locations on disk.

3.1 Timestamped BAT

Other works often focused on a study of in-memory database systems, but this line of research has always worked with Out-of-Core (OOC) secondary storage. Since multiple ad-hoc searches are required to perform most updates, a more efficient measure was sought out to improve query optimization. Traditional OOC updates on a column-store takes places in two phases. The first phase requires the seeking of target OIDs of values that match update predicates. The second phase is the update on the values stored with the OIDs. The traditional seek-update could incur great costs in I/O due to added data-block seeking and writing.

In [12], a new structure called a Timestamped Binary Association table (TBAT) was created to make use of a new procedure called the Asynchronous OOC (AOC) Update. A TBAT, is a traditional BUN/BAT stored with a timestamp. The format of these Timestamped-BUNs (TBUNs) is a triple; timestamp, OID, and attribute-value. A visual representation can be seen in Figure 2, which takes the customer_balance and customer_id BATs and forms them into a TBATs by adding the timestamp. Note that all of the original data is said to be “Time1” because this data is considered to be a single batch-insert.

(a) TBAT `customer_balance`

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

(b) TBAT `customer_id`

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

Figure 2: TBAT Examples

3.2 Asynchronous Out-of-Core Update

3.2.1 An example of an AOC Update

The AOC Update was created to try and prevent the constant ad-hoc data-seeking required for a traditional update on a column-store. It does so by appending new updates to the bottom of the TBATs. The updates will have a later timestamp than the existing data with the same OID(s). The original sorted and indexed data is referred to as the body, and the unsorted and unindexed stream of updates is referred to as the appendix. Figure 3 shows a visual representation in which 1.00 is added to the customer balance of the customer with OID 102. The database, during non-peak times, will be placed into an offline data-cleaning mode, where the appendix will be merged into the body to save on space and make search-times more efficient.

The query is: *update customer set balance=201.00 where id=2*

The new tuple appended on the TBAT will be given an update on its timestamp, making it Time2 (>Time1). Note that Time2 refers to the later, correct data. Incoming update streams will repeat the same process, and any updates to the customer-balance TBAT at OID 102 will be made with a value equivalent of Time3, Time4...Time-n.

optime	oid	float	
time1	101	100.00	
time1	102	200.00	body
time1	103	300.00	
time2	*102*	*201.00*	appendix

Figure 3: TBAT partitioning `customer_balance` following AOC Update

Despite the updates, data-consistency is maintained. All OIDs corresponding to a query target will pass the query predicates despite the timestamps, but the result-tuples will be filtered again by keeping only the OID with the latest timestamp only as output to the user. In this way there are no dirty reads or excess data returned by a query to the user.

3.2.2 Partition of TBAT after AOC Updates

Following the AOC update on a TBAT file, the TBAT is partitioned into the body and the appendix as shown in Table 2 above. The appendix consists of those OIDs that were the targets of various updates. While the body is sorted and indexed, the appendix is not. The body can be scanned over with a binary search while the appendix requires a linear search.

This method of partitioning the TBATs on the basis of updates makes sense in practice when write-optimization is the goal. An update can be appended at the end of a TBAT file more quickly than the various locations on the OOC storage can be found and updated. The goal of quick writes for column-stores can be easily achieved by the AOC update. So long as there exists enough space on disk for the appendix, write speeds will see a more efficient AOC update that requires less overhead.

3.2.3 Cost Analysis of the AOC Update

An experiment was performed using two tables, a BAT and TBAT, both consisting of 10,000 records. Five update streams ranging from 10% - 50% (ten-percent increments) were applied to each table to measure the results of the two update methods. The update experiment, Figure 4, shows significant findings. Results of the AOC update showed an extreme increase to query efficiency and update-query speed. The average running time of a standard BAT update was 7.14 seconds, whilst being only 4.81 milliseconds for the AOC update. This equates to a 1466.436 times faster on average.

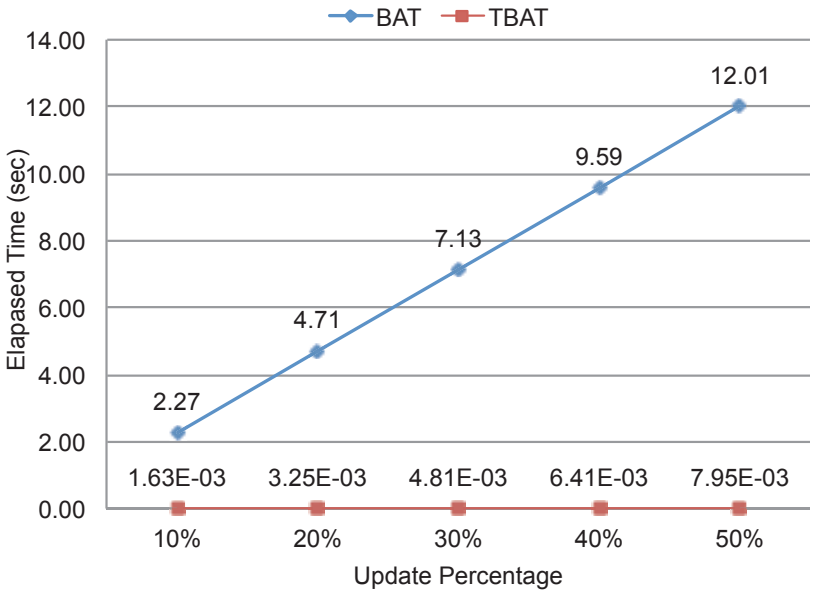

Figure 4: AOC update on TBAT vs traditional update on BAT

Given that most of that standard column-store update costs are associated with the cost for I/O, this significant boost to efficiency is explained. Figure 5 shows the order of magnitude speed increase for the AOC update on a TBAT vs the traditional update on a BAT. Costly random ad-hoc searches, with attributes sometimes spanning multiple disk-blocks

(or even disk pages), taking into account that most updates change multiple attributes inside a tuple, can amount to significant overhead costs.

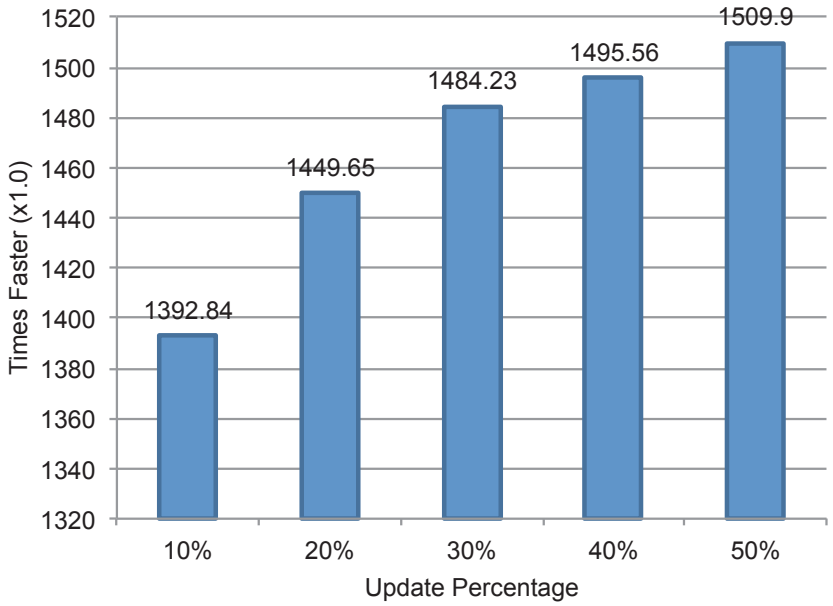

Figure 5: AOC update speed increases compared to traditional updates

3.3 Selection after the AOC Update

In [13] the drawbacks of the AOC update method are discussed in detail, and solutions are proposed to make its use more efficient. There were two main problems with the AOC update. Firstly, conducting the AOC update partitions the TBAT into two sections: body and appendix. As the appendix would grow, the need for data-cleaning became evident, for the search times would would increase as the volume of data in the appendix expands.

Secondly, there existed no means to perform data cleaning without turning the database off (offline data-cleaning). In this context, data cleaning means the merging of the appendix with the body to have the most up-to-date data exist in a fully sorted and indexed

way.

3.3.1 Selection Algorithm

Searching on a BAT file is straightforward. There exists only the BAT itself, which allows for a swift binary search to locate any specific OID targeted by a query predicate. Any search conducted on a TBAT before an AOC update is equally straightforward. It is only after the first AOC update that the search complexity increases. If a selection query is run on any TBAT containing an appendix, it must first search the appendix for any OIDs that pass the predicate(s). Only then does the search move into the body of the TBAT, where any new OIDs are kept but any matching those gathered from the appendix are discarded. Figure 6 shows how this added query overhead increases overall complexity when dealing with an appendix.

For searching on a TBAT file with an appendix the selection query overhead is $\frac{TBAT}{BAT} X 100\%$.

3.3.2 Selection Speed Degeneration after AOC Updates

Given that the overhead increase, so too does the query time. As mentioned earlier, the unsorted nature of the appendix, since it is a random update stream, doesn't allow for use of a binary search. Instead, a linear search must be conducted to check each OID within the appendix.

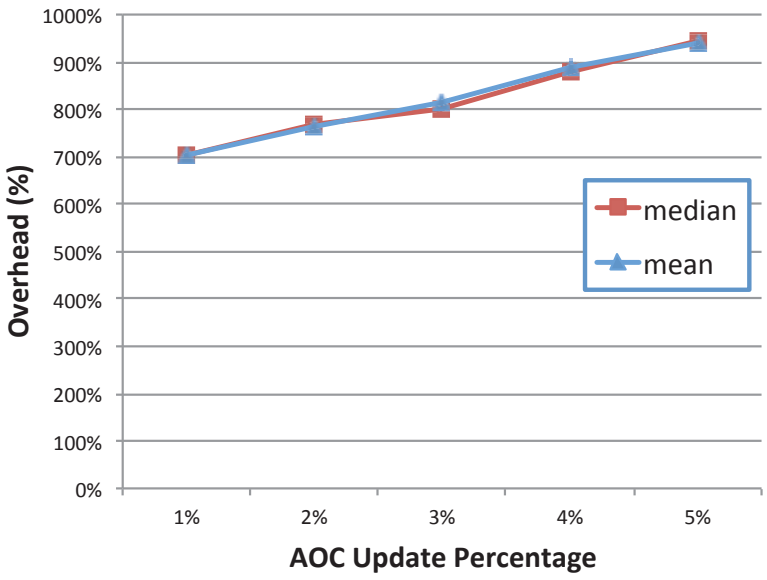

Figure 6: TBAT selection overhead compared to a BAT

If the size of the data in the appendix is small there may be no noticeable difference in query performance. This is true at first, but as the input stream increase the size of the appendix, the degrading of query performance becomes an issue.

3.4 Data Cleaning for AOC Updates

Naturally, the larger the appendix on the TBAT, the greater the increase the query time. Overtime, the obvious need to clean the data emerged. In this line of research, "cleaning" is used to refer to the merging of OIDs from the appendix into the body. In this way, the body will contain all the data; its sorted and indexed format will allow for a binary search upon clean, up-to-date data. In this state, the query performance will be at peak efficiency. In [12] and [13] there was an offline data cleaning method and two online data cleaning methods created, respectively.

3.4.1 Offline Data Cleaning after the AOC Update

The offline data cleaning method, Algorithm 1, is flawed in that it requires the database to be turned off, ignoring incoming query streams until cleaning is complete. While this may be appropriate for databases that handle small workloads, it is entirely inappropriate for databases that have to handle constant input streams. The continuing emergence of big-data environments makes this approach increasingly less suitable for most environments.

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
```

The offline data cleaning algorithm is called "merge-update", displays how the offline data cleaning works. Firstly a merge-sort is applied to the TBAT(s) and the oid are placed in ascending order while the timestamp is placed in descending order. In the second phase the file is read sequentially, picking out the first occurrence of each OID (the one with

the newest timestamp) and removing the rest, thereby keeping only the up-to-date data with the most recent timestamp. The entire complexity of the merge-update method for offline data cleaning is O(n log n). The main problem of this method stems from the offline nature. While the database is offline, incoming queries cannot run and must wait until the data has been fully cleaned. For this reason, the eager data cleaning and the progressive data cleaning methods were created, for speed priority and memory-usage priority, respectively.

3.4.2 Online Eager Data Cleaning

The online data-cleaning algorithms differ from offline data cleaning in that they do not need to turn the database off, making it appropriate for a constant query stream. The online methods also make use of a data-structure called a snapshot. The first online algorithm, called "merge-eager" (online eager data-cleaning), is displayed in Algorithm 2. It begins by making a snapshot of the body, and creating a new appendix file linked to the TBAT. The older appendix will be merged into the body, making use of merge-sorting and binary searching. During this time, the tuples in the appendix will be written to the snapshot as a traditional update like that of a BAT file. When merging is finished, the snapshot's body will replace the original and the appendix file will be purged. Through use of the snapshot, users can still run selection queries upon the body and appendix while the cleaning occurs. The eager data-cleaning method is used for speed priority to merge the appendix into the body swiftly. Typical for use in environments with abundant memory, the eager method merges the entire appendix in one step to reach peak search performance as fast as possible.

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
```

3.4.3 Online Progressive Data Cleaning

The second online algorithm, "merge-progressive" (online progressive data cleaning), shown in Algorithm 3, is used for environments where memory is scarce. In these extreme cases, the entire appendix will be unable to fit inside memory, thus requiring the need for the progressive data cleaning method. This method encapsulates the online eager data cleaning method, but differs in that the appendix file must be partitioned into smaller appendix-blocks. These data blocks, whose size are manually defined by the DBA (block size $<$ available memory), are placed into an appendix queue where each TBAT can contain more than one appendix. The appendix queue will be attached to the TBAT instead of the individual appendix, and will contain enough appendix-blocks to hold the entire appendix. One-by-one an appendix is removed from the appendix queue, and the above online eager data-cleaning method is applied until the entire appendix of the TBAT has been merged into the snapshot and it replaces the body. Any streaming updates can also be added to the appendix queue.

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
```

One experiment was conducted, Figure 7, on a randomly generated 64MB BAT file to mimic the minimum block-size of a big-data environment. Following this the BAT

file was converted into TBAT file with an original-time timestamp. Five update streams were created at 1% - 5% of the original size (1% increments) and searches were applied to each. An appendix queue was created with a block size of 10% of the appendix so that it would take ten increments of data-cleaning to fully merge the appendix with the body. The results show that with each increment the searching speeds increased, because the volume of data inside the appendix requiring a linear search became smaller and smaller.

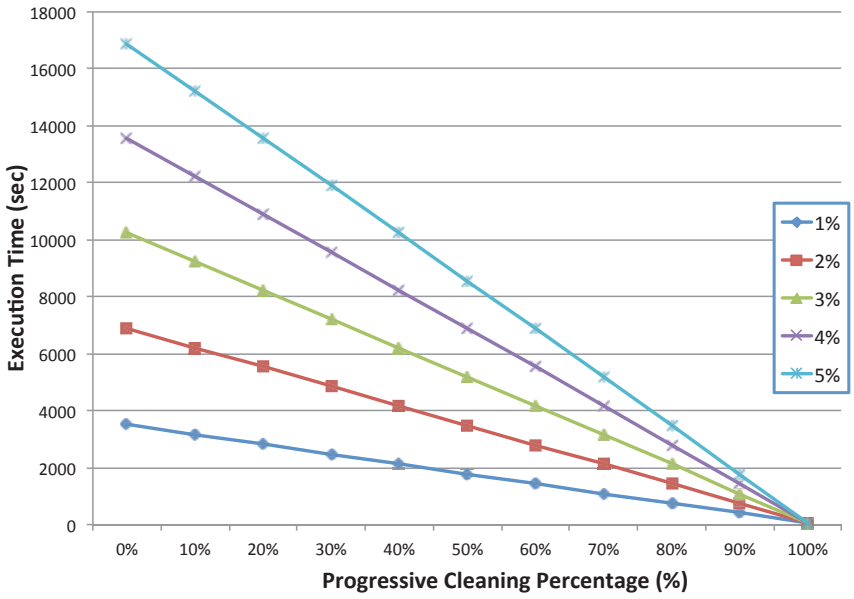

Figure 7: Selection execution time during progressive data cleaning

After ten steps the search speed for each stream merged into its peak performance as the entire appendix was merged into the body of the TBAT. While the greater the size of the appendix, the greater the need is to perform a data-cleaning, but it is important to note that the more data there is the faster it will merge into the TBAT and meet peak performance. Table 1 shows the results of select queries before the cleaning, and after each step is implemented.

	cleaning (%)										
update (%)	0	10	20	30	40	50	60	70	80	90	100
1	3526	3180	2833	2486	2141	1795	1451	1106	762	418	70
2	6901	6224	5549	4868	4188	3502	2815	2129	1443	757	70
3	10248	9244	8239	7229	6215	5200	4181	3155	2125	1100	70
4	13574	12250	10918	9581	8238	6891	5538	4179	2811	1441	70
5	16865	15221	13572	11915	10249	8574	6893	5197	3496	1782	70

Table 1: Selection Execution Time (sec) after Online Progressive Data Cleaning (%) with respect to AOC Update from 1% to 5%

4 Experiment Results

While the online-data cleaning methods for the AOC update were efficient, they did not optimize searching on the appendix. Since the appendix can be a rapidly changing environment, a B-Tree can easily index this workload. By indexing the appendix with a B-Tree it will make search speeds more efficient. In this case, the experiments will include an offset B-Tree which will keep the positions of the OIDs. It is expected that searching with a B-Tree will be more efficient than standard linear search through the appendix of the TBAT.

4.1 Tests of Updates on TBAT and BAT

These experiments are designed to compare searching on a TBAT file's appendix using a linear search, and searching using an offset B-tree index. It is expected that performance will be significantly improved as the volume of data becomes greater and greater. The experiment was performed with an Ubuntu 14.04 Virtualbox utilizing an Intel Core i7-4770 3.4GHz CPU with four processor-cores, 11GB memory, and a 40GB SATA 7200RPM

hard disk. All data for these experiments are kept on disk, however, the B-trees will be constructed to reside in memory.

The experiment is tested on a variety of datasets: 1000 lines, 1MB (47660 lines), 10MB, 32MB, and 64MB BAT files. These datasets consist primarily of OID and value pairs. While each dataset is tested upon, the priority is the 64MB BAT. This is because a 64MB dataset can be used to simulate the standard block size for the big data Hadoop Distributed File System (HDFS). Then a TBAT file is constructed using the current timestamp. Five update query tables are then constructed, consisting of 1% to 5% of the dataset. Five updated BAT and TBAT tables are created to simulate these 1% to 5% input streams; the updates on the TBAT are AOC updates that form an appendix. These updated files replace the original files. Following this, a selection query consisting of a 10% selection ratio will be conducted on the updated TBAT and BAT files.

There will be three methods used for simulating the selection. Firstly is the method that requires a linear search on the appendix; the body is still sorted, allowing for a binary search if the value isn't first found in the appendix. The other two methods consist of offset B-trees to search the appendix fist. A random number generator and a 10% update file are used in these cases, respectively. Upon construction, the offset B-trees remain in memory.

It is important to first look at the base-case – the linear search on the appendix. Figure 8, shows the results of the linear search. As can be seen, the greater percentage of update, the greater the gap between the search speeds. Each percentage increase steadily

raised the search-time, because any search must first check the appendix for an updated value before checking the body of the TBAT. For the 1MB dataset the difference is hardly a hundred milliseconds between the update streams, but by 64MB the gap in the search times for the update streams increase about 260 seconds with every 1% increase to the update stream. The linear search on the appendix for a 5% update stream takes approximately 4396.994 seconds (73.28 minutes). The lack of performance efficiency is obvious in these showings, making the need to index more apparent.

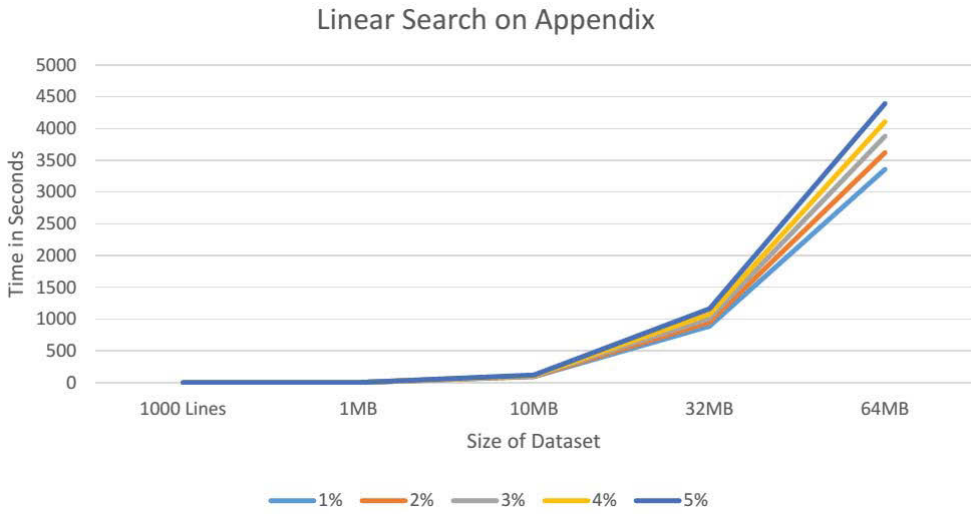

Figure 8: Selection execution time for a Linear Search

The results for the offset B-tree tests for a 10% selection ratio using random number generation and a 10% update file can be seen in Figure 9 and Figure 10 respectively. It can be seen that these graphs are nearly identical, lending no clear advantage to either method. The random number generation is only a fraction of a percentage faster, given that it does not first have to read its search OID from a separate file. It is important to note that the higher the percentage of update on the TBAT, the more efficient the search speed on average. The reason for this boost is that both an appendix "hit" (OID found in offset

B-tree) and an appendix "miss" (OID not found in offset B-Tree) must first check to see if a newer value exists within the B-tree first, and then check the file for the value. In the case of an appendix hit, where the value exists within the index, the much larger body can be skipped and the appendix searched at the point of offset that the B-tree indicated; on average taking fewer steps than even a binary-search on a file. The higher the percentage of update, the more likely an appendix hit will occur, and the search speeds actually decrease gradually with a larger appendix.

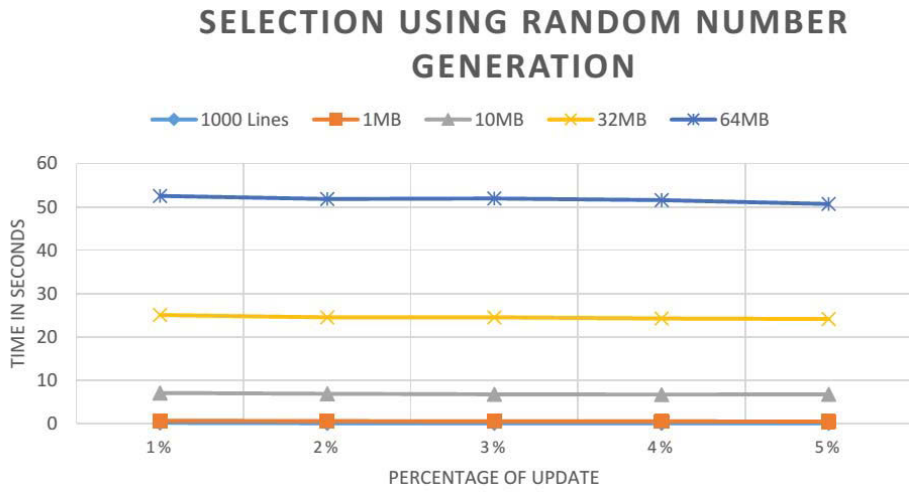

Figure 9: Selection execution time for a Random Number Generator using an Offset B-Tree

In Figure 11 the results of the update file search can be seen and compared against the linear search. Comparing the speed of these two methods at each dataset yields some excellent results. On average the speed of the offset B-tree search is far more efficient than that of a linear search. Figure 12 shows the comparison for the four largest datasets to see how many times faster the indexed search is compared to the linear search. It is important to note that the 1000 lines dataset was left off of this chart because such a small dataset actually saw a degradation in performance. If we compare the 5% update stream search for

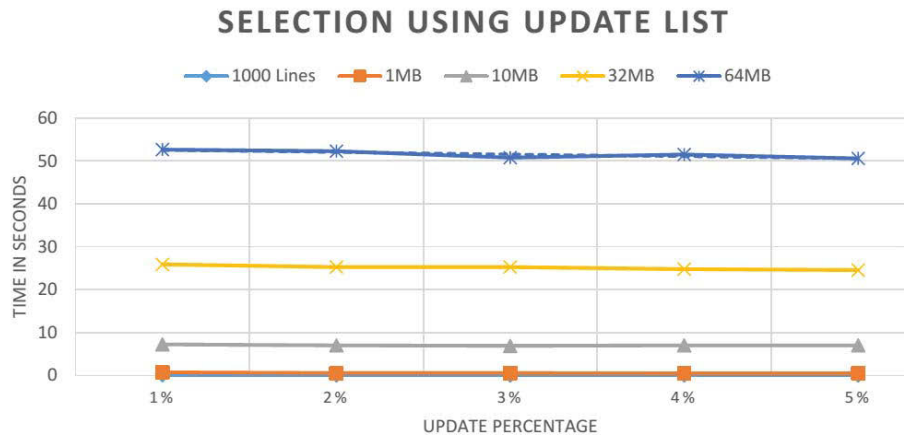

Figure 10: Selection execution time using a 10% file and an Offset B-Tree

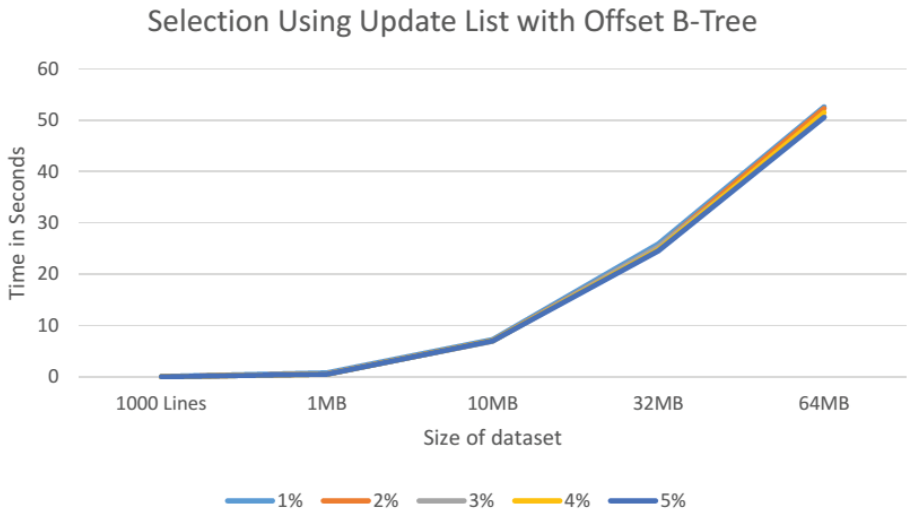

Figure 11: Selection time for a 10% file using an Offset B-tree

the linear method, which took 4396.994 seconds, vs the offset B-tree method, which took 50.626 seconds, we see that the order of magnitude speed increase is 86.852 times faster.

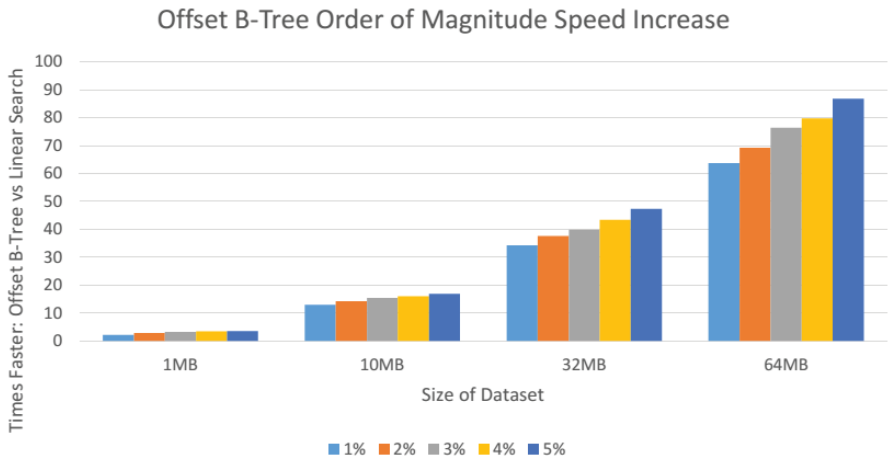

Figure 12: Order of magnitude speed increase: Offset B-Tree vs Linear Search

5 Conclusion and Future Works

In this research, the use of offset B-trees following an AOC updates on TBATs in OOC column-store databases was introduced. It was discovered that for any medium-to-large file, the effect of offset B-trees could greatly increase search speeds on a TBATs appendix section. Also, the larger the update percentage the faster the search-speeds due to an increase in appendix hits. The indexed method, therefore, proved to be a successful optimization and a more efficient alternative to the standard linear search for all but the smallest TBAT files. Future research will seek to use the offset B-tree index to optimize the online data cleaning methods.

References

[1] 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.

[2] Al-laham, M., and M. M. El Emary, I. Comparative study between various algorithms of data compression techniques. *International Journal of Computer Science and Network Security Vol. 7*, No. 4 (2007), 281–291.

[3] Douglis, F. On the role of compression in distributed systems. *ACM/IEEE* (1993), 1–6.

[4] Graefe, G., and Shapiro, L. D. Data compression and database performance. *ACE/IEEE* (1991), 1–9.

[5] Huang, W.-J., Saxena, N., and McCluskey, E. Reliable lz data compressor on reconfigurable coprocessors. *Center for Reliable Computing* (2000), 1–10.

[6] Klein, S. T., and Wiseman, Y. Parallel lempel ziv coding. *Discrete Applied Mathematics Vol. 146* (2005), 180–191.

[7] Liang, G., RunHeng, L., Yan, J., and Xin, J. Join directly on heavy-weight compressed data in column-oriented database. *WAIM* (2010), 357–362.

[8] Mikhail J., A., and Lonardi, S. Authentication of lz-77 compressed data. *ACM* (2014), 407–412.

[9] Plattner, H. A common database approach for oltp and olap using an in-memory column database. *SIGMOD 09* (2009), 1–7.

[10] Schmidt, A., Kimmig, D., and Hofmann, R. Basic building blocks of column-stores. *International Journal of Advances in Software Vol. 6*, No. 1 and 2 (2013), 14–24.

[11] Shun, J., and Zhao, F. Practical parallel lempel-ziv factorization. *Carnegie Mellon University* (2013), 1–10.

[12] 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.

[13] Yu, F., Hou, W.-C., Luo, C., and Jones, E. Online data cleaning for out-of-core column-store databases with timestamped binary association tables. *CATA 2015* (2015), 1–6.