Data Cleaning in Out-of-Core Column-Store Databases: An Index-Based Approach

Feng Yu¹, Tyler Matacic¹, Weidong Xiong², Mohammed Ali A Hamdi², Wen-Chi Hou²
¹Computer Science and Information Systems, Youngstown State University, Youngstown, OH, USA
²Department of Computer Science, Southern Illinois University, Carbondale, IL, USA

Abstract

Write optimization in out-of-core (or external memory) column-store databases is a well-known challenge. Timestamped Binary Association Table (or TBAT) and Asynchronous Out-of-Core Update (or AOC Update) have shown significant improvements for this problem. However, after a time period of AOC updates, the selection query performance on TBAT gradually decreases. Even though data cleaning methods can merge update records in TBAT to increase ad-hoc searching speed, it could be a time-consuming process. In this work, we introduce multiple data cleaning methods utilizing the index structure called offset B⁺-tree (or OB-tree). When the OB-tree and updating records can be fit into the system memory, an eager data cleaning approach is introduced for fast cleaning speed. In a data intensive environment, the OB-tree index or the updating records might be too large to fit into memory; therefore, a progressive data cleaning approach is introduced which can divide the update records into small slips and clean the data a memory-economic manner.

keywords: column-store database, data cleaning, index, B⁺-tree

1. Introduction

Column-store databases (also known as columnar databases or column-oriented databases) have drawn much attention recently. They refer to the databases that vertically partition data and separately store each column. The history of column-store databases can be traced back to 1970s when transposed files were implemented in the early development of DBMS, followed by applying vertical partitioning as a technique of table attribute clustering. By the mid-1980s, the advantage of a fully decomposed storage model (DSM) over the traditional row-based storage model (NSM or Normalized Storage Model) was studied [1]–[3].

TAXIR (TAXonomic Information Retrieval) is the first automatic application of column-store database focusing on biological information retrieval and management [4], [5]. KDB and Sybase IQ were the first two commercially available column-store databases developed in 93 and 95, respectively. It’s not until about 2005 when many open-source and commercial implementations of column-store databases took off [6]. The well-known column-store databases include: Apache Cassandra [7], Apache HBase [8], MonetDB [9], KDB, SAP HANA [10], [11], and Vertica [12].

The data storage in a column-store database is vertically partitioned and sharded by projecting each column into a separate fragment. A vertical fragment is referred as a BAT (Binary Association Table) [9], which is stored contiguously on a large enough disk page in order to mitigate seeking overheads across multiple ranges of data. The data in each BAT is densely patched in order to improve I/O performance, and also rapidly compressed utilizing light-weight compression schema to improve storage efficiency.

One of the benefits of a column-store database is its information retrieval speed, which is much faster than a row based database. Thanks to the DSM feature, the column-store database fits well into the write-once-and-read-many environment. The column-store database works especially well for OLAP and data mining queries that retrieve a large number of tuples but only considers a small collection of attributes. Put simply, it can retrieve only the attributes included in the query prediction without the need to read the entire tuple. Another featured benefit of the column-store database is data compression, which can reach a higher compression rate and higher speed than traditional row-based databases. One of the major reasons for this higher compression is that the information entropy in the data of a single column is lower than that of traditional row-based data.

Optimizing write operations in a column-store database has always been a challenge. Existing works focus on write optimizations in a main-memory column-store database. Krueger et al. [13], [14] introduced the differential update to improve the write performance in MonetDB. A special columnar data structure called the delta buffer was introduced to temporarily store decomposed row-based input data. However, to the best of our knowledge, very few works focused on optimizing the write performance on the out-of-core (OOC or external memory) column-store databases.

Vertica [12], a column-store database for large vol-
ume OOC storage, introduces a specially designed data storage procedure called \textit{k-safety} to ensure ACID of update transactions on large volumes of data and improve the data importation efficiency. Nevertheless, \textit{k-safety} focuses more on the transaction control rather than the write performance improvement for high velocity update query streams.

In [15], an efficient solution was proposed to optimize the write operations (update and deletion) on an OOC column-store database. An operation called \textit{Asynchronous Out-of-Core Update} (or \textit{AOC Update}) was originally designed based on a new data structure called \textit{Timestamped Binary Association Table} (or \textit{TBAT}). There is a potential problem that, after a period of time of AOC updates, the selection query performance on TBAT gradually deteriorates [16].

In order to address the problem of performance deterioration after multiple AOC updates, data cleaning methods [15] can be used to clean up update records from the body of the TBAT into the appendix of the TBAT. In [16], online data cleaning methods are introduced in order to clean up TBAT without the need of locking the file. However, the data cleaning procedure can be time-consuming on large data sets [17].

In [18], a new index structure called \textit{Offset B^+}-tree (or \textit{OB-tree}) is introduced for fast data retrieval in the TBAT file. OB-tree is a succinct sparse index specially designed for a TBAT where AOC updates are performed. It replaces the global pointers with relative pointers, called \textit{offset}, to save storage space. In addition, OB-tree supports fast searching queries including ad-hoc and range queries on TBAT.

In this research, we aim to introduce data cleaning methods utilizing the OB-tree index to achieve a higher speed performance. Based on the concurrent usage of a TBAT file while data cleaning is performed, we divide the TBAT conditions into \textit{cold data} and \textit{hot data} conditions. The cold data condition is when no other users could possibly change the TBAT while data cleaning is performed. However, updates could happen when data cleaning is performed simultaneously. In the interest of simplicity, this work focuses on the data cleaning algorithms on cold data. Furthermore, to adapt our research for a data intensive environment, we develop OB-tree based cleaning methods based on the data size compared with the memory size. When the OB-tree and the updating records can be fit into the system memory, we introduce an \textit{eager data cleaning} approach for fast data cleaning. On the other hand, for memory bottleneck scenarios, we introduce the \textit{progressive data cleaning} approach.

The rest of the paper is structured as follows. Section 2 is the background introduction of column-store databases. Section 3 shows the performance degeneration after AOC updates. Section 4 introduces data cleaning methods without using index. The OB-tree data structure is revisited in section 5. In section 6, we introduce multiple data cleaning methods utilizing the OB-tree index. The preliminary experiment results are shown in section 7. Section 8 is the conclusion and future works.

2. Inside Column-Store Databases

The data structure of a column-store database exclusively uses \textit{BATs} (Binary Association Tables). A BAT is a fragment of an attribute in the original row-based storage. It usually consists of an \textit{oid} (Object Identifier) or \textit{ROWID}, along with a column of attribute values, which in a pair is called a \textit{BUN} (Binary UNits). It is a physical model in a column-store database and the sole bulk data structure it implements. The BAT is categorized in a special group of storage models called \textit{Decomposed Storage Model} (or DSM) [1], [2].

The row-based storage data is the original user input data, called the \textit{front-end} data or \textit{logical} data. To input the data into a column-store database, a mapping rule should be defined from the logical data structure to the physical data structure, namely BAT.

\textbf{Example 1.} (From Row-Based Table to BAT) Suppose a row-based table is \textit{customer}. It consists of three attributes \textit{id}, \textit{name}, \textit{balance}, and \textit{id} the primary key. The row-based data is shown in Fig. 1(a). In a columnar database, this logical table will be decomposed into 3 BATs namely \textit{customer_id}, \textit{customer_name}, \textit{customer_balance}. Each BAT contains two columns: an \textit{oid} and an attribute value column with the column name as the corresponding column data type.

In Example 1, the logical table is fully decomposed into 3 BATs, Fig 1(b)-1(d), with each BAT containing one of the attributes. This is also referred to as \textit{full}}
vertical fragmentation [6]. Full vertical fragmentation has many advantages. First of all, data accessing is efficient for queries accessing many rows but with fewer columns involved in the query. Another advantage is the reduction of the workload on the CPU and memory generated by OLAP and data mining queries, which typically consider only a few columns in a logical table.

Compared to fully vertical fragmentation, the other pattern is partial vertical fragmentation [19]. It assumes the prior knowledge of which columns are frequently accessed together. Also, it employs the attribute usage matrix to determine optimal clustering of columns into vertical fragments. However, OLAP and data mining are application areas that indicate ad-hoc queries, as a good OLAP or data mining system must be able to quickly answer queries involving attributes of arbitrary combinations. Nevertheless, the partial vertical fragmentation is useful to detect the data block location in a distributed database system.

3. Selection Speed Degeneration after AOC Updates

The AOC update [15] is a fast update method on column-store database. It will increasingly create pending data in the appendix of the TBAT. For small TBAT files and a small amount of AOC updates, the effect on selection queries can hardly be noticeable. However, for larger files and a large portion of AOC updates, the decreased selection speed cannot be ignored. The overhead comparing the selection execution time on TBAT vs BAT is illustrated in Figure 2. Using a randomly generated 1MB TBAT file, we perform AOC updates by 1% to 5% of the original file. We perform randomly generated ad hoc selection queries, with a selection ratio of 10%, on both normally updated BAT files and AOC updated TBAT files. The mean overhead and median overhead are 819% and 822%, respectively.

4. Data Cleaning without an Index

To improve the selection performance on the TBAT after AOC updates, multiple data cleaning methods which don’t use indexes are developed in [15] and [16]. The purpose of data cleaning in an OOC column-store database is to detect the latest version of updated data and merge them into the body of the TBAT. By the requirement of locking the database, those methods are divided into two groups, namely offline data cleaning [15] and online data cleaning [16].

4.1 Offline Data Cleaning

Offline data cleaning can only be performed after the database has been locked to avoid inconsistent data during the cleaning process. An offline data cleaning method is introduced in [15], merge_update which can remove the duplicated TBUNs in the TBAT with same oid but different timestamp’s.

Offline data cleaning first employs a merge sort on the entire TBAT file including the body and appendix, and then deletes the duplicated TBUN’s in a sequential manner. This requires profound time in execution when a large amount of AOC updates are accumulated. Even though the time issue is not the first concern for an offline data cleaning approach, a better time efficient manner is obviously preferred.

4.2 Online Data Cleaning

The online data cleaning approaches are developed in [16], which consist of an eager approach and a progressive approach for speed-priority and memory-priority, respectively.

The central idea of online data cleaning, compared with the offline approach, is to enable the users to continue querying the TBAT during the data cleaning procedure time. This is a major focus of the online approach especially when the database is a streaming environment, where the input is non-stop. The main difference for online data cleaning is the employment of a sophisticated data structure called a snapshot.

The online approach will first make a snapshot of the body and create a new appendix file linked to the TBAT. The older version of the appendix will be merged into the snapshot of the body utilizing merge sorting and binary searching. During this time, the TBUN’s in the appendix will be written to the body as the traditional update on the BAT. After the merging is complete, the snapshot of the body will replace the original body in the TBAT, and the older version of appendix will be purged.

In a data intensive environment, the updated data might be too large to be fit into the main memory. Thus we separate the online data cleaning process into two different approaches, namely an eager approach and
a progressive approach, for speed priority and memory priority, respectively.

### 4.2.1 Online Eager Data Cleaning

The central idea of online eager data cleaning is to increase the merging speed. The entire appendix of the TBAT will be read into the memory and perform online merging into a snapshot of the body. Put simply, this approach merges the entire appendix at once. After that, the merged snapshot will replace the original body in the TBAT file.

### 4.2.2 Online Progressive Data Cleaning

Online progressive data cleaning fits for the data intensive scenarios where the entire appendix may not fit into memory. In these cases, the eager approach cannot be applied. The key concept in online progressive data cleaning is the appendix queue, where each TBAT can contain more than one appendix. The size of each appendix, or block size, needs to be manually defined by the database administrator, which cannot exceed the size of the available memory on the system. The original appendix of a TBAT file will then be split into separate appendixes according to the block size. The appendix queue will be attached to the TBAT instead of a single appendix.

During the progressive data cleaning procedure, each time an appendix is retrieved from the appendix queue, an eager data cleaning approach is then performed to merge the split appendix with the snapshot of the body. Simultaneously, the appendix queue can create a new split appendix file to accept streaming updates and enqueue the appendix once its size researches the block size.

### 5. Offset B⁺-Tree Index

#### 5.1 Data Structure

*Offset B⁺-tree*, or simply *OB-tree* is introduced in [18]. Namely, the OB-tree is a variant of B⁺-tree. It is developed based on the B⁺-tree and has several important properties that the TBAT requires.

1. **OB-tree** has a succinct data structure that can be easily adopted by existing column-store databases.
2. **OB-tree** is a sparse index for only the updated records in a TBAT. An OB-tree can be either stored in main memory or serialized on hard disk. When the OB-tree is stored in main memory, the data retrieving speed can be orders of magnitude faster.
3. **OB-tree** allows the insertion of duplicated keys. In fact, the key in an OB-tree is the *oid* in the TBAT data file. It is possible that a record associated with one *oid* is updated multiple times.

Inside an OB-tree, there are two categories of nodes, namely internal nodes and leaf nodes. The top node is the root node. It is a leaf node when the OB-tree has only one layer, and an internal node when the OB-tree has multiple layers. Associated with each OB-tree, the parameter *n* determines the layout of every node inside the OB-tree.

- Each internal node will have space for *n* search keys, i.e. *oids*, and *n + 1* pointers that can be used to point to other nodes. At least, \([n + 1]/2\) of the pointers are used. If the internal node is a root, we only require that at least 2 pointers are used regardless of the value of *n*.
- Each leaf node will also have space for *n* search keys. But among the *n + 1* space units, only 1 allocation is used to point to the next leaf node in the sequence. The left *n* units are reserved to save a special value, called *offset*, used to point to the location of the updated record in the appendix of a TBAT.
- How to assign each *oid* to each node is the same as in a B⁺-tree.

An *offset* in the OB-tree is a scalar recording of the relative location of an updated record inside an appendix that is appended at the end of the body of TBAT. We assume 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 *k*th line, **1 ≤ k ≤ l_a**, of the appendix, the *offset associate with this updated record* is *k*. Intuitively, the updated record with offset *k* is at the \((l_a + k)\)th line in the entire TBAT.

**Remark 1.** The offset is a scalar pointer to the target record in the TBAT. The space cost of an offset can be relatively smaller than a pointer in any operating system. The employment of the offset can be considered as a simplified method of pointer elimination.

Since pointers can occupy additional spaces inside any B/B⁺-tree, we use this scalar offset to replace most of the pointers at the leaf nodes, and the actual location of a record can be promptly calculated by the definition of offset. The data type of an offset can be flexible and decided by the user. Typically, it can be the same data type of the *oid*, which can be a 4 byte or 8 byte integer. To save more space, smaller bytes of an integer can also be considered with the assumption that at most a smaller portion of the TBAT records are updated.

**Example 2** (A Basic Example of OB-Tree). We demonstrate a simple OB-tree. We set the parameter *n* to be 3, i.e. each node in the OB-tree can store up to 3 *oids* and 4 pointers. From the appendix, we can generate *oid*-offset pairs of (*oid*, offset), indicating the *oids* and the associated offsets. Suppose the given *oid*-offset pairs are \{(1, 1), (2, 2), ..., (10, 10)\}. Figure 3 demonstrates the OB-tree after inserting each *oid* and offset.
The major advantage of using an OB-tree for data cleaning is to hasten the searching speed when looking for those updated records and their locations. The oid’s are well organized in the OB-tree, and their locations can be easily calculated by retrieving their offsets associated with oid’s in the OB-tree. Compared with data cleaning without any index [15], [16], OB-tree-based data cleaning features a fast retrieval speed on updated records because of its B⁺-tree nature. In addition, there is no need to pre-sort the appendix of the TBAT which also saves significant system time.

Because the OB-tree is a variant of a B⁺-tree, range searchings for oid’s are rather fast, we can output from the OB-tree the updated oid’s with their offsets into an update list and perform merging while reading this update list. Based on the size of the updated records in the appendix, one can choose to use either an eager cleaning approach when the update list can be fit into the main-memory, or a progressive cleaning approach when the update list is too large to be fit into the main-memory. Please note that in the latter situation, OB-tree itself could also be too big to be fit into the main-memory; thus, we need to serialize the OB-tree and progressively retrieve updated oid’s so that the entire procedure can be processed.

Data cleaning on the TBAT can happen on both cold data and hot data. Cold data refers to the scenario when the TBAT is in a locked condition and no writing from other users is allowed. On the other hand, hot data means there could be other users writing to the TBAT file while data cleaning is in execution. In this work, we focus on the scenario of cold data cleaning since data cleaning on hot data is discussed in [16].

### Algorithm 1 OB-Tree Eager Cleaning on Cold Data

**Input:** tbat: TBAT file; obtree: OB-tree

**Output:** tbat: TBAT file after data cleaning

```
1: procedure OB-Tree-Clean-EAGER(tbat, obtree)
2:   n_update ← update_list.getLength()
3:   update_list ← obtree.outputUpdateList()
4:   n_body ← tbat.body_length
5:   row_num1 ← 1
6:   while update_list.hasNext() do
7:     (oid, offset) ← update_list.getNext()
8:     row_num1 ← row_num1 + offset
9:     row_num1 ← tbat.body.binarySearch(oid, row_num1, n_body) ▷ binary search starting from row_num1 and return found location
10:    row_num1 ← tbat.body.seekRelative(row_num1, row_num1) ▷ use last found locatoin as the next start
11:   end while
12:   tbat.appendix.destroy() ▷ destroy appendix after cleaning
13:   tbat.close() ▷ close TBAT file
14: return SUCCESS
15: end procedure
```

### Algorithm 2 OB-Tree Progressive Cleaning on Cold Data

**Input:** tbat: TBAT file; update_slip_queue: the queue of updating slips from OB-tree

**Output:** tbat: TBAT file after data cleaning

```
1: procedure OB-Tree-Clean-PROGRESSIVE(tbat, update_slip_queue)
2:   while update_slip_queue.hasNext() do
3:     update_slip ← update_slip_queue.dequeue()
4:     OB-Tree-Clean-EAGER(tbat, update_slip)
5:   end while
6: end procedure
```
6.1 Eager Data Cleaning

If the OB-tree and the update list of \( \langle \text{oid}, \text{offset} \rangle \) can be fit into memory, an eager data cleaning approach is preferred. In addition, we assume the \text{oid}'s in the TBAT body may not necessarily be consecutive. The algorithm of eager cleaning is illustrated in Algorithm 1.

In the first phase, we generate the update list of \( \langle \text{oid}, \text{offset} \rangle \) that need to be cleaned from the OB-tree. Since the OB-tree is a B^+-tree variant, one can first search for the left-most leaf node, and then start to find its sibling nodes until all leaf nodes are exported. The reason that we export all updating \text{oid}'s and their offsets at once, instead of one-by-one, is to avoid multiple searches in the OB-tree which could be time-consuming. The update list is currently sorted by \text{oid} in ascending order.

In the second phase, we retrieve from the update list the records which need to be merged into the body. Each time, we first retrieve a pair of \( \langle \text{oid}, \text{offset} \rangle \) from the update list. The appended record with the latest value is located at the line (\text{bat.body_length} + \text{offset}). The target record, which needs to be updated in the body of the TBAT, can be searched using a special deductive binary searching method. It can be located by a binary search with the row number range from the previous target row number, or \text{row_num}, to the maximum row number in the body, or \text{n_body}. Especially, when searching for the first \text{oid} in the update list, we let \text{row_num} equal to 1. In such a manner, the next binary search can use the previous binary search result to lower the time cost. This deductive searching approach ends when the pairs in the update list are all merged into the TBAT body. After all of this is completed, the appendix of the TBAT file can be destroyed.

6.2 Progressive Data Cleaning

If the OB-tree or the update list of \( \langle \text{oid}, \text{offset} \rangle \) is too large to be fit into memory, a progressive data cleaning approach is preferred. In this scenario, we assume the OB-tree is serialized into secondary storage and read into the memory by segmentations. The algorithm of progressive data cleaning is illustrated in Algorithm 2.

The key difference of the progressive cleaning approach is to organize all updating pairs of \( \langle \text{oid}, \text{offset} \rangle \) into slips, called update slips. The size of the update slips can be manually determined by the administrator according to the hardware and operating system configurations. For each update slip read into memory, we can use the eager approach to merge the pairs of \( \langle \text{oid}, \text{offset} \rangle \) into the TBAT body. Please note that, when merging each update slip, there is no need to read the OB-tree because the offset information is included into the update slip. Therefore, the data cleaning speed is guaranteed.

7. Data Cleaning Experiments

In these experiments, we perform comprehensive comparison of data cleaning with OB-tree (the index-based approach) and without any help of index (the traditional approach). Since our goal is to test the performance improvement using the OB-tree index, we assume all the updated data is loaded into the memory at once. In the traditional approach, the updated records are first sorted in-memory, and then merged into the OB-tree body by using binary searching. The only memory usages for all methods include the storage for OB-tree and the sorting for updated records.

7.1 Experiment Design

We focus on synthetic TBAT datasets of 64MB to mimic a data block in HDFS (Hadoop Distributed File Systems) [20]. We generate random queries to update the datasets with 5 updating ratios from 1% to 3% with the increment of 5%. After that, both data cleaning approaches are performed on same updated datasets to compare their performances in speed and space costs.

7.2 Result Analysis

Three key measurements are recorded during the tests including execution time, disk access count, and memory cost of OB-tree.

First of all, Figure 4 depicts the results of data cleaning execution time on a 64MB dataset using OB-tree and traditional methods. It is obvious that on each stage of update ratio the index-based approach is faster than the traditional approach. The speed difference between the two methods become more obvious with the increment of the updating ratio and the size of data. This can also be observed in Figure 5, where the improvement is measured by relative overhead between two methods defined as

\[
\frac{(\text{time(traditional)} - \text{time(OB-Tree)})}{\text{time(OB-Tree)}} \times 100\%
\]

The borderline lies on the updating ratio of 20% where their differences exceed 10%. In addition, the performance difference is increasing steadily with the updating ratio.

In general, the OB-tree index prominently improves the data cleaning process in all tests compared with the traditional method without using any index. The memory cost associated with OB-tree is minor and increases slowly with the updating ratio. This memory cost can be further mitigated by properly adjusting the parameters of the OB-tree.

8. Conclusion and Future Works

In this research, we introduce data cleaning methods on the TBAT using the OB-tree index. We introduce...
two data cleaning approaches, namely the eager cleaning approach and the progressive cleaning approach. The eager cleaning approach assumes the data size can be fit into memory. On the other hand, the progressive cleaning approach is designed for a large dataset scenario. Preliminary experiments have shown that the proposed data cleaning methods utilizing OB-tree is much faster than traditional methods.

For future work, we will continue to perform comparison experiments on big data sets. More extensive experiments will be designed to compare their speed performance and space costs.

References


