Efficien and Scalable Data Evolution with Column Oriented Databases

Ziyang Liu[1], Bin He[2], Hui-I Hsiao[2], Yi Chen[1]
Arizona State University[1], IBM Almaden Research Center[2]
ziyang.liu@asu.edu, binhe@us.ibm.com, hhsiao@almaden.ibm.com, yi@asu.edu

ABSTRACT

Database evolution is the process of updating the schema of a database or data warehouse (schema evolution) and evolving the data to the updated schema (data evolution). It is often desired or necessitated when changes occur to the data or the query workload, the initial schema was not carefully designed, or more knowledge of the database is known and a better schema is concluded. The Wikipedia database, for example, has had more than 170 versions in the past 5 years [8]. Unfortunately, although much research has been done on the schema evolution part, data evolution has long been a prohibitively expensive process, which essentially evolves the data by executing SQL queries and re-constructing indexes. This prevents databases from being flexibly and frequently changed based on the need and forces schema designers, who cannot afford mistakes, to be highly cautious. Techniques that enable efficient data evolution will undoubtedly make life much easier.

In this paper, we study the efficiency of data evolution, and discuss the techniques for data evolution on column oriented databases, which store each attribute, rather than each tuple, contiguously. We show that column oriented databases have a better potential than traditional row oriented databases for supporting data evolution, and propose a novel data-level data evolution framework on column oriented databases. Our approach, as suggested by experimental evaluations on real and synthetic data, is much more efficient than the query-level data evolution on both row and column oriented databases, which involves unnecessary access of irrelevant data, materializing intermediate results and re-constructing indexes.

Categories and Subject Descriptors

H.2.4 [**Systems**]: Relational databases

General Terms

Algorithms, Performance

Keywords

column oriented database, bitmap index, schema, data evolution

1. INTRODUCTION

Database evolution is the process of updating the schema of a database/data warehouse (schema evolution) and evolving the data to the new schema (data evolution). The needs of database evolution arise frequently in modern databases [16, 15, 27, 29] and data warehouses [10, 18] in both OLTP and OLAP applications due to emerging of new knowledge about data and/or changes in workload, and from both new application demand perspective and analytical experiments perspective. As observed by Curino et. al [16], "the serious problems encountered by traditional information systems are now further exacerbated in web information systems and cooperative scientific databases where the frequency of schema changes has increased while tolerance for downtimes has nearly disappeared."

For instance, the Ensembl Genetic DB has involved in over 400 schema changes in nine years [16]. Another example is a data warehouse containing raw data and derived data (analytical results on the raw data). Since there are more and more new analytical requirements nowadays, the derived data and their schema often become complex and frequently evolving. Even for a single analytical task, it is also often true that it may take some time to develop and experiment before being stabilized, which will generate many schemas during the process. Consider a patent database as an example. In the first round of building an annotator to extract chemical compounds from patents, we may extract SMILES[1] in the patents, chemical names and the corresponding usages. Later on, we may want to put SMILE and chemical name attributes into a separate table to reduce the redundant information [13].

In general, a database evolution may be desired or necessitated when new information about the data and/or the workload emerges. In the remainder of this paper we use the tables in Figure 1 as a simplified example to illustrate the motivation and the proposed methodologies.

1. **New Information about the Data.** In a dynamic application scenario, new information about the data may become available, which requires to update, add or remove attributes in a database, as well as re-organize the table structure. Consider table $\mathbb{R}$ in Fig-

[1]SMILES are string expressions for chemicals

Employee	Skill	Address
Jones	Typing	425 Grant Ave
Jones	Shorthand	425 Grant Ave
Roberts	Light Cleaning	747 Industrial Way
Ellis	Alchemy	747 Industrial Way
Jones	Whittling	425 Grant Ave
Ellis	Juggling	747 Industrial Way
Harrison	Light Cleaning	425 Grant Ave

ℝ

schema 1

Employee	Skill
Jones	Typing
Jones	Shorthand
Roberts	Light Cleaning
Ellis	Alchemy
Jones	Whittling
Ellis	Juggling
Harrison	Light Cleaning

𝕊

Employee	Address
Jones	425 Grant Ave
Roberts	747 Industrial Way
Ellis	747 Industrial Way
Harrison	425 Grant Ave

𝕋

schema 2

Figure 1: Example of Database Evolution

ure 1. Suppose originally, it only has attributes *Employee* and *Skill*. If later on the address information emerges, we would like to add an *attribute Address* to ℝ. Furthermore, at the schema design time, the designer might have believed that each *Employee* has a single *Skill*. When more data *tuples* are added, it is revealed that each employee may have multiple skills. Thus it is better to decompose ℝ to two tables 𝕊 and 𝕋, as shown in schema 2, in order to prevent data redundancy and update anomaly. As we can see, updating the data content may reveal or invalidate functional dependencies that may not be known when the schema was originally designed, and thus requires schema changes.

2. **New Information about the Workload.** Query and update workload of a database may vary from time to time. Different workloads may have different data access patterns, and demand different schemas for optimized performance. Consider Figure 1 again, assume that the original workload is update intensive, for which Schema 2 is desirable. Later workload characteristics change and become query intensive, and most queries look for addresses given skills. Now Schema 1 becomes more suitable, as it avoids joins of two tables.

A database evolution involves both updating the schema of a database and evolving the data to the new schema. Existing work on database evolution mainly studies schema evolution and its effects [15, 27, 11, 14, 20, 34, 37]. However, efficient algorithms to actually evolve the data to the new schema has yet to be developed. Currently data evolution is performed at *query-level*, expressed and implemented by SQL queries [26]. As an example, the following SQL queries can be executed to evolve the data from ℝ to 𝕊 and 𝕋 in Figure 1:

1. INSERT INTO 𝕊 SELECT EMPLOYEE, SKILL FROM ℝ
2. INSERT INTO 𝕋 SELECT DISTINCT EMPLOYEE, ADDRESS FROM ℝ

Such a query-level data evolution framework is notoriously inefficient in terms of both time and space for two reasons. First, it has excessive data accesses; every (distinct) attribute value in every tuple is accessed. Second, after the query results are loaded into the new tables, indexes have to be built from scratch on the new table. This approach makes data evolution prohibitively expensive and even inapplicable, as to be shown in this paper and is also observed in [15]. The inflexibility of schema change on current database systems not only results in degraded system performance over time, but forces schema designers to be highly cautious and thus severely limits the database usability [21].

In this paper, we study the problem of efficient data evolution in designing an agile database system. Ideally, an agile data evolution should minimize data accesses; and it is also desirable if we can derive the new indexes from the original indexes, rather than constructing the indexes on the new table from scratch.

To minimize data accesses, we observe that when changing the database schema, usually many columns in a table remain unchanged. For example, the columns in table 𝕊 in Figure 1 are exactly the same as the corresponding ones in ℝ, thus these columns are unchanged during the decomposition from ℝ to 𝕊 and 𝕋. As a result, a column-oriented database, or column store, becomes a natural choice for efficient data evolution. In a column-oriented database, each column, rather than each row, is stored continuously on the storage. Column stores usually outperforms row stores for OLAP workloads, and studies have shown that column stores fit OLTP applications as well [6, 33].

However, performing a query-level data evolution on a column oriented database is still very expensive. First, we need to generate query results, during which the columns need to be assembled into tuples; second, result tuples need to be disassembled into columns; at last, each column of the results needs to be re-compressed and stored. These steps incur high costs, as to be shown in Section 5.

To address these issues, we propose a framework of *data-level* data evolution on column oriented databases, which achieves storage to storage data evolution without involving queries. A system of our data level approach has been demonstrated in VLDB '10 [25]. We support various operations of schema changes, among which two important ones, decomposing a table and merging tables, are the focus of this paper. We design algorithms that exploit the characteristics of column oriented storage to optimize the data evolution process for better efficiency and scalability. Our approach only accesses the portion of the data that is affected by the data evolution, and has the unique advantages of avoiding SQL query execution and data decompression and compression. We report a comprehensive study of the proposed data evolution algorithms on column oriented storages, compared with the traditional query-level data evolution. Experiments show that data-level data evolution on column oriented databases can be orders of magnitude faster than query-level data evolution on both row and column oriented storages for the same data and schemas. Besides, it is observed that our approach is more scalable, while the performance of query-level data evolution deteriorates quickly when the data become large. Our results bring a new perspective to help users make choices between column oriented storage and row oriented storage, considering both query processing and database evolution needs. Due to the advantage in data evolution, column stores may get an edge over row stores even for OLTP workloads in certain applications.

Our contributions in this paper are summarized as:

1. This is, to the best of our knowledge, the first study of data evolution on column oriented storages, and the first on designing algorithms for efficient data-level evolution, in contrast to the traditional query-level data evolution. Our techniques make database evolution a more feasible option. Furthermore, our results bring a new perspective to help users make choices between column oriented storage and row oriented storage, considering both query processing and database evolution needs.

2. We propose a novel data-level data evolution framework that exploits column oriented storages. This approach only accesses the portion of the data that is affected by the data evolution, and has the unique advantages of avoiding SQL query execution, indexes reconstruction from the query result, and data decompression and compression.

3. We have performed comprehensive experiments to evaluate our approach, and compared it with traditional query-level data evolution, which show that our approach achieves remarkable improvement in efficiency and scalability.

The following sections of this paper are structured as: Section 2 introduces necessary backgrounds on column oriented databases and the types of schema changes. Then we present the algorithms for two important types of schema change: decomposition (Section 3) and merging (Section 4). We report the results of experimental evaluations to test the efficiency of data evolution in Section 5. Section 6 discusses related works and Section 7 concludes the paper.

2. PRELIMINARIES

2.1 Column Oriented Databases

Unlike traditional DBMSs which store tuples contiguously, column oriented databases store each column together. Column store improves the efficiency of answering queries, as most queries only access a subset of the columns, and therefore storing each column contiguously avoids loading unnecessary data into memory. In addition, as values in a column are often duplicate and/or similar, compression and/or encoding technologies can be applied in column stores to reduce the storage size and I/O time.

Studies show that column store is suitable for read-mostly queries typically seen in OLAP workloads, and it is a good choice for many other application scenarios, e.g., databases with wide tables and sparse tables [3], vertically partitioned RDF data [4], etc. In these applications, schema evolution is common. For instance, in OLAP applications, a warehouse may add/delete dimensions. When new data or workload emerges, a warehouse may want to evolve a dimension from star schema to snowflake schema (or verse visa), which is exactly the decomposition (or merging) operations studied in this paper. This paper proposes novel techniques that efficiently support schema evolution in column stores.

Bitmap is the most common encoding scheme used in column stores [5]. The state-of-the-art bitmap compression methods and encoding strategies have made bitmap indexes widely applicable. Modern bitmap indexes can be applied on all types of attributes (e.g., high-cardinality categorical attributes [35], numeric attributes [28] and text attributes [32]). Studies show that a compressed bitmap index occupies less space than the raw data and provides better query performance for equal queries [36], range queries [35], and keyword queries [32]. Besides, [36] shows that with proper compression, bitmaps perform well for columns with cardinality up to 55% of the number of rows, indicating that they can be widely adopted. Nowadays, bitmap indexing is supported in many commercial database systems (e.g, Oracle, Sybase, Informix), and is often the default (or only) indexing option in column-oriented database systems (e.g., Vertica, CStore, LucidDB), especially for applications with read-mostly or append-only data, such as OLAP applications and data warehouses. In this paper we assume that, unless an attribute (also referred to as a column) is a primary key, it is compressed using bitmap encoding.[2]

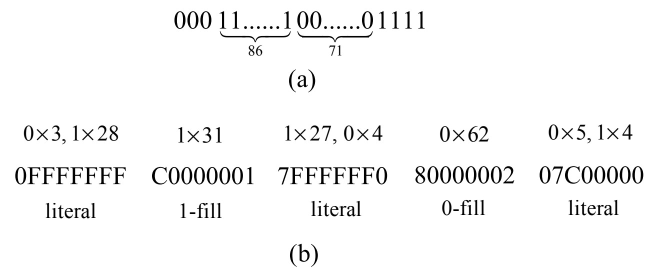


Figure 2: WAH Compression of Bitmap

Various compression methods for bitmaps have been proposed. Among those, Word-Aligned Hybrid (WAH) [36], Position List Word Aligned Hybrid [17] and Byte-aligned Bitmap Code (BBC) [9] can be applied to any column and be used in query processing without decompression. We adopt Word-Aligned Hybrid (WAH) to compress the bitmaps in our implementation. In contrast, run length compression [33] can be used only for sorted columns.

WAH organizes the bits in a vector by words. A word can be either a literal word or a fill word, distinguished by the highest bit: 0 for literal word, 1 for fill word. Let L denote the word length in a machine. A literal word encodes $L-1$ bits, literally. A fill word is either a 0-fill or a 1-fill, depending on the second highest bit, 0 for 0-fill and 1 for 1-fill. Let N be the integer denoted by the remaining $L-2$ bits, then a fill word represents $(L-1) \times N$ consecutive 0s or 1s. For example, the bitmap vector shown in Figure 2(a) is WAH-compressed into the one in Figure 2(b).

In order to find the corresponding vector in a bitmap given a value in a column, we use a hash table for each column to map from each value to the position of the vector. The number of entries in the hash table is the number of distinct values in the column. We also build an offset index similar to the one in [33] for each bitmap to find the value in a column given a position.

[2]Other compression schemes are sometimes used for special columns, such as run length encoding for sorted columns. In this paper we focus on the general case with bitmap encodings for all non-key attributes. Supporting other compression/encoding schemes is one of our future works.

Table 1: Types of Schema Updates

SMO	Description
DECOMPOSE TABLE	Decompose a table into two tables. The union of the attributes in the two output tables equals to the attributes of the input table
MERGE TABLES	Create a new table on storage by joining two tables
CREATE TABLE	Create a new table in the database
DROP TABLE	Delete a table from the database
RENAME TABLE	Rename a table, keeping its data unchanged
COPY TABLE	Create a copy of an existing table
UNION TABLES	Combine the tuples of two tables with the same schema into one table
PARTITION TABLE	Partition the tuples into a table into two tables with the same schema with a condition
ADD COLUMN	Create a new column for a table and load the data from user input or by default
DROP COLUMN	Delete an existing column and its associated data
RENAME COLUMN	Change the name of a column without changing data

2.2 Schema Updates

We support all the Schema Modification Operators (SMO) introduced in [15], which are listed in Table 1. Create, Drop, and Rename Table operations incur mainly schema level operations and are straightforward. Copy, Union and Partition Table operations require data movement, but no data change. Thus existing approaches for data movement can be applied. For column level SMO, especially Add Column and Drop Column, column oriented databases have obvious advantages, as data in other columns do not need to be accessed. In this paper, we discuss Decompose and Merge in details. We will propose algorithms for efficient decomposition and merging, and show empirically that these operations are much more efficient on column stores than on row stores.

Decomposition. This operation decomposes a table into two tables. As an example, see Figure 1. Table $\mathbb{R}$ is decomposed into two tables ($\mathbb{S}$ and $\mathbb{T}$). We assume that a decomposition is lossless [31], since only a lossless-join decomposition ensures that the original table can be reconstructed based on the decomposed tables. In a lossless-join decomposition of table $\mathbb{R}$ into $\mathbb{S}$ and $\mathbb{T}$, the common attributes of $\mathbb{S}$ and $\mathbb{T}$ must include a candidate key of either $\mathbb{S}$ or $\mathbb{T}$, or both. Decomposing a table into multiple tables can be done by recursively executing this operation.

Merging. It joins two tables to form a new table, such as $\mathbb{S}$ and $\mathbb{T}$ in Figure 1 into $\mathbb{R}$. Merging multiple tables can be done by recursively executing this operation.

Next, we will discuss the data-level evolution algorithms for decomposition and merging respectively in Section 3 and Section 4.

3. DECOMPOSING A TABLE

In this section we discuss the details of lossless-join decomposition. We use $\mathbb{R}$ to denote the original table, and $\mathbb{S}$, $\mathbb{T}$ to denote the two output tables.

We start by introducing two critical properties of a lossless-join decomposition which will be exploited by our algorithms:

Property 1: In a lossless-join decomposition which decomposes a table into two, at least one of the two output tables has the same number of rows as the input table.

Property 2: For an output table with a different number of rows as the input table, its non-key attributes have functional dependency on its key attributes in the original table.

Now we prove these two properties. Without loss of generality, let $\mathbb{R}$ denote the input table and $\mathbb{S}$, $\mathbb{T}$ the output tables. By the definition of lossless-join decomposition, the common attributes of $\mathbb{S}$ and $\mathbb{T}$ must include a candidate key of either $\mathbb{S}$ or $\mathbb{T}$. Suppose the common attributes, denoted by A_c, are a candidate key of $\mathbb{T}$. Then if we join $\mathbb{S}$ and $\mathbb{T}$, since they join on A_c and each tuple of $\mathbb{T}$ has a unique value of A_c, each tuple in $\mathbb{S}$ can have at most one matching tuple in $\mathbb{T}$. Therefore, $\mathbb{S}$ has the same number of rows as the input table $\mathbb{R}$, and property 1 holds. Furthermore, since the join is essentially extending each of $\mathbb{T}$'s key value with $\mathbb{S}$'s matching non-key attributes, thus for any of $\mathbb{T}$'s key values in $\mathbb{R}$, the corresponding $\mathbb{T}$'s non-key values should be the same, and thus Property 2 holds.

Property 1 can be effectively utilized by column oriented databases. Since each column is stored separately, the unchanged output table can be created right away using the existing columns in $\mathbb{R}$ without any data operation. Column-store makes it possible to only access the data that is necessary to change in data evolution, and thus save computation time.

3.1 Algorithm Overview

In our algorithm description below, let the set of attributes in $\mathbb{R}, \mathbb{S}, \mathbb{T}$ be $(A_1, \cdots, A_k, A_{k+1}, \cdots, A_n)$, $(A_1, \cdots, A_k, A_{k+1}, \cdots, A_p)$, and $(A_1, \cdots, A_k, A_{p+1}, \cdots, A_n)$ respectively. Assume that the common attributes of $\mathbb{S}$ and $\mathbb{T}$, $A_1, \cdots, A_k$, comprise the key of $\mathbb{T}$, which means $\mathbb{S}$ is the unchanged output table.

Intuitively, to generate $\mathbb{T}$, we can take the following steps:

1) For each distinct value v of $\mathbb{T}$'s key attributes $A_1, \cdots, A_k$, we locate a tuple position in $\mathbb{R}$ that contains v. The result of this step is a list of tuple positions in $\mathbb{R}$, one for each distinct value of $\mathbb{T}$'s key attributes.

Each v presents a list of k values from $\mathbb{T}$'s key attributes. Because of Property 2, all tuples in $\mathbb{R}$ with the same v have the same values on $A_{p+1}, \cdots, A_n$. We thus can choose any one of these tuples and do not need to access all of them.

2) Given the position list, we can directly generate new bitmaps of $\mathbb{T}$'s attributes from their corresponding bitmaps in $\mathbb{R}$. That is, for each attribute, we can shrink their bitmap in $\mathbb{R}$ by only taking the bits specified in the position list. We name such an operation as "bitmap filtering".

During this process, the hash and offset index for each of $\mathbb{T}$'s attributes are also updated. After we shrink each vector using the bitmap filtering operation, we insert an entry to the corresponding hash table for the attribute, recording the position of the shrunk vector. For each position in the position list, we access the offset index of the original attribute (in $\mathbb{R}$) to retrieve the attribute value, and insert an entry accordingly to the corresponding offset index for the attribute in $\mathbb{T}$.

The pseudo-code for the above two steps is presented in Algorithm 1. Step one is realized by calling a **distinction** function, which is to be described in Section 3.2. It returns *keyvalues*, which records the distinct values of $A_1, \cdots, A_k$, and *keypos*, storing one position of each of these values in $\mathbb{R}$. The positions in *keypos* are then sorted to support efficient search in step two.

EXAMPLE 3.1. *In Figure 1, the key of table $\mathbb{T}$ is a single attribute* Employee. *We first compute its distinct values stored in keyvalues, which are* Jones, Roberts, Ellis *and* Harrison. *We also find one position of each of them in $\mathbb{R}$ stored in keypos, which are, for example, rows 1, 3, 4 and 7 respectively.*

In step two (lines 3-9 of Algorithm 1), When $\mathbb{T}$ only has one key attribute A_1 (i.e., $k = 1$), it is unnecessary to store A_1 with a bitmap structure as each A_1's value only occurs once. We thus simply output the distinct values in *keyvalues* (line 4). For other cases and non-key attributes, we conduct the "bitmap filtering" operation for each attribute. The details of the **filtering** operation is described in Section 3.3. It is easy to see that, since we deal with one attribute at a time in step two, the number of attributes to be processed does not bring a memory constraint for our algorithm.

Algorithm 1 Decomposing a Table

DECOMPOSE ($\mathbb{R}(A_1, \cdots, A_n), k, p$)
Output: $\mathbb{T}(A'_1, \cdots, A'_k, A'_{p+1}, \cdots, A'_n)$

1: $keyvalues, keypos =$ **distinction**$(\mathbb{R}, A_1, \cdots, A_k)$
2: sort *keypos*
3: **if** $k = 1$ **then**
4: $A'_1 =$ **output**$(keyvalues)$
5: **else**
6: **for** $i = 1$ to k **do**
7: $A'_i =$ **filtering**$(A_i, keypos)$
8: **for** $i = p + 1$ to n **do**
9: $A'_i =$ **filtering**$(A_i, keypos)$

EXAMPLE 3.2. *In the running example, table $\mathbb{T}$ has a single key attribute:* Employee. *According to line 4 of Algorithm 1, column* Employee *in table $\mathbb{T}$ consists of the values in keyvalues, which are* Jones, Roberts, Ellis *and* Harrison.

Given the list of positions $keypos = \{1, 3, 4, 7\}$, in order to generate column Address *in the output table $\mathbb{T}$, we need to perform a filtering operation on column* Address *in $\mathbb{R}$. There are two vectors for this attribute, corresponding to* 425 Grant Ave *and* 747 Industrial Way. *Their vectors in the bitmap are 1100101 and 0011010 respectively. We retrieve the values of the two vectors at positions 1, 3, 4 and 7. For* 425 Grant Ave, *the values are 1001, and for* 747 Industrial Way *the values are 0110. Therefore, the new bitmap of* Address *consists of two vectors, 1001 and 0110.*

Next we discuss the two procedures involved in a decomposition, **distinction** and **filtering**, in details.

3.2 Distinction

We start with the easier case – distinction on a single attribute, such as attribute *Employee* in Figure 1. Since the attribute on which **distinction** is performed is not the key of table $\mathbb{R}$, it is encoded as a bitmap structure.

Algorithm 2 Distinction on a Single Attribute A

DISTINCTION (Bitmap of Attribute A)
Output: *keyvalues*, *keypos*

1: $keyvalues = \emptyset$
2: $keypos = \emptyset$
3: **for** each distinct value v of A **do**
4: $vec = v$'s vector in A's bitmap
5: $elapsed = 0$
6: **for** $i = 1$ to $vec.size$ **do**
7: **if** $vec[i]$ is a literal word **then**
8: $j =$ position of the first seen 1-value bit in $vec[i]$
9: $keypos = keypos \cup \{elapsed + j\}$
10: $keyvalues = keyvalues \cup \{v\}$
11: break
12: **else if** $vec[i]$ is a 1-fill word **then**
13: $keypos = keypos \cup \{elapsed + 1\}$
14: $keyvalues = keyvalues \cup \{v\}$
15: break
16: **else**
17: $N =$ the number of 0s encoded in $vec[i]$
18: $elapsed \mathrel{+}= N$
19: **return** *keyvalues*, *keypos*

Algorithm 2 presents the steps for distinction on a single column, denoted as A. Recall that A's bitmap has a vector for each of A's distinct values, and therefore the bitmap directly gives the distinct values of A. To find the position of each distinct value, we traverse A's value $\rightarrow$ vector position hash; for each of A's distinct value v, we locate the position of v's vector (line 4), and then find the position of the first "1" in the vector (lines 6-18), which is the position of the first occurrence of v in column A (recall that actually the position of any other "1" can be used). The position of the first "1" in each vector is stored in array *keypos* and the distinct values in *keyvalues*.

To find the position of the first "1" for each distinct value, we can simply scan each vector. We notice that in a bitmap compressed using WAH, the first "1" of any vector always appears in the first two words, unless the table has more than 31×2^{30} ($\approx$ 30 billion) rows (assuming the machine word length is 32), which is rarely seen. Therefore in the majority of cases, we do not need to access the entire bitmap of a column at all. Instead, we can additionally record the first two words of each vector in the bitmap in a separate file, called the *snapshot* of the bitmap. When we perform distinction on this column, this file can be used instead of the bitmap itself to find the position of the first "1" in each vector. The snapshot file is tiny compared to the bitmap, as regardless of the length of the bitmap vector, in a snapshot each vector has only two words.

When there are more than one key attribute, we cannot directly get distinct values from the bitmap, as the bitmap is constructed on a per attribute basis. There are two ways to deal with this case. The first method is to combine bitmaps of multiple attributes. Consider two attributes A and B, each encoded as a bitmap. For each value v_A of A and v_B of B, if we perform an AND operation on their vectors, the resulting vector indicates the position where v_A and v_B occur together. Therefore, if we perform AND on every vector of A and B and keep the non-zero vectors, we can get the distinct values of (A, B) and their positions with similar process of Algorithm 2. It is easy to generalize this method on three or more attributes. This approach works well if A and B have a small number of distinct values, as its processing time is proportional to the product of the number of A's distinct values and the number of B's distinct values. As an alternative, we can also scan the offset index of the involved columns to access each row of these columns, and use hashing to recognize each distinct values during the scan.

3.3 Filtering

The filtering operation generates a new bitmap of an attribute given its original bitmap and a list of positions, *keypos*. Each vector in the new bitmap only contains rows specified by *keypos* in the original bitmap, and thus this operation can be considered as "filtering" the original bitmap.

Algorithm 3 Filtering a Bitmap Based on a List of Positions

```
FILTERING (Bitmap and offset index for Attribute A, keypos)
Output: the filtered bitmap tgtBitmap, the new hash tgtHash
and offset index tgtOffset
1: for each vector vec of A's bitmap do
2:   vec' = an empty vector
3:   elapsed = 0
4:   currpos = 1
5:   for i = 1 to vec.size do
6:     if vec[i] is a literal word then
7:       elapsed += 31
8:       while currpos ≤ keypos.size and keypos[currpos] ≤
         elapsed do
9:         add the (keypos[currpos] − elapsed)th bit of vec[i]
           to vec'
10:        encode the new added bits in the vector vec' if
           necessary using WAH encoding
11:        currpos++
12:    else
13:      elapsed += the number of 0s (1s) encoded in vec[i]
14:      endpos = use a binary search to find the largest po-
         sition in keypos satisfying keypos[endpos] ≤ elapsed
15:      add (endpos − currpos + 1) 0s (1s) to the vector vec'
16:      encode the new added bits in the vector vec' if nec-
         essary using WAH encoding
17:      currpos = endpos + 1
18:  add the vector vec' to tgtBitmap
19:  add the position of tgtBitmap to tgtHash
20: for i = 1 to keypos.size do
21:   value = the value of A at row keypos[i]
22:   add entry (i, value) into tgtOffset
23: return tgtBitmap
```

The algorithm for the filtering operation is shown in Algorithm 3. For each vector *vec* of attribute A, we traverse the list of positions *keypos*. Specifically, for each word $vec[i]$ in *vec*, if it is a 0-fill or a 1-fill word, we directly jump from the current position in *keypos* to the largest position that is still in the range of $vec[i]$. This can be done using a binary search since *keypos* is sorted. All positions in between are either 0 if $vec[i]$ is a 0-fill, and 1 otherwise. Suppose we jumped p positions, then we can directly insert p 0s (or 1s) into the corresponding vector vec' in the target bitmap (lines 13-15). If $vec[i]$ is a literal word, we traverse *keypos*, and for each position in the range of $vec[i]$, we test whether the value is 0 or 1 in that position in $vec[i]$, and insert one bit of 0 or 1 into the target vector vec' (lines 6-11). In both cases, we check the recently added bits and compress them using WAH encoding if possible (line 10 and line 16). In this way, a target bitmap is efficiently built. The hash table for the target bitmap is constructed by inserting an entry after a vector has been built (line 19). The offset index for the target bitmap is constructed by inserting an entry for each position in *keypos* (lines 20-22).

4. MERGING TWO TABLES

In this section we discuss efficient evolution techniques for another important type of data evolution – merging. Since merging more than two tables can be done by recursively merging two tables at a time, we focus on the discussion of merging two tables.

Similar as a decomposition, a merging operation can be performed on a row oriented database by running SQL joins. However, for the same reason as decomposition, the processing time of such a query level data evolution can be huge and prohibitive for big tables. This is true for both row and column stores. In this section, we present techniques for utilizing the storage scheme of column stores to perform data-level evolution which avoids materializing query results and decompressing the bitmap encodings. Without loss of generality, we use $\mathbb{S}$ $(A_1, \cdots, A_k, A_{k+1}, \cdots, A_m)$, $\mathbb{T}$ $(A_1, \cdots, A_k, A_{m+1}, \cdots, A_n)$ to denote the two input tables, and our goal is to merge $\mathbb{S}$ and $\mathbb{T}$ into table $\mathbb{R}$ $(A_1, \cdots, A_n)$.

Note that a merging essentially creates a new table by joining two tables. From the perspective of data evolution, we can classify mergings into three scenarios: 1) Both input tables can be reused for merging, 2) Only one input table can be reused for merging, and 3) Neither input table can be reused for merging.

In scenario 1, the two input tables have the same primary key attributes and their join attributes contain these key attributes. This is a trivial situation for column-store, as the generated table $\mathbb{R}$ can simply reuse the existing column storages of $\mathbb{S}$ and $\mathbb{T}$.

Our focus in this section is thus to discuss algorithms for the other two scenarios. For scenario 2, the join attributes of the two input tables comprise the key of one input table, and thus the other input table's columns are reusable. For example, tables $\mathbb{S}$ and $\mathbb{T}$ in Figure 1 merge into table $\mathbb{R}$. *Employee* is the only common attribute of $\mathbb{S}$ and $\mathbb{T}$, which is the key of $\mathbb{T}$. As we can see, the columns in $\mathbb{S}$ can be directly used as the corresponding columns in $\mathbb{R}$. We refer to this case as *key-foreign key based merging* and present the algorithm to specifically deal with this scenario in Section 4.1. Section 4.2 presents a general algorithm to cope with any scenarios and is mainly used for scenario 3.

4.1 Key-Foreign Key Based Merging

In the key-foreign key based merging, instead of generating all columns in $\mathbb{R}$, we can reuse the columns in $\mathbb{S}$ (i.e., $A_1, \cdots, A_m$) and only generate $n - m$ columns $A_{m+1}, \cdots, A_n$ for $\mathbb{R}$. The new bitmap of each A_i ($m+1 \le i \le n$) in $\mathbb{R}$ can be obtained by deploying the bitmap of A_i in $\mathbb{T}$ and the bitmap of the key attributes in $\mathbb{S}$. The example below shows how the new bitmap generation works.

EXAMPLE 4.1. *We again use Figure 1 as our running example, in which we merge $\mathbb{S}$ and $\mathbb{T}$ into $\mathbb{R}$. In this case, $\mathbb{S}$ and $\mathbb{T}$ have a single join attribute* Employee *(i.e., k=1), which is also the key of $\mathbb{S}$. When we build $\mathbb{R}$, the bitmaps, hash tables and offset indexes of attributes* Employee *and*

Skill *can reuse the ones of* $\mathbb{S}$*, and only those of* Address *need to be generated.*

The bitmap vector of 425 Grant Ave *in* $\mathbb{T}$ *is 1001, which means this address should appear in rows of* $\mathbb{R}$ *with employee name* Jones *or* Harrison*. Since the vectors of* Jones *and* Harrison *are 1100100 and 0000001 (in both* $\mathbb{S}$ *and* $\mathbb{R}$*) respectively, the new bitmap vector of* 425 Grant Ave *is thus 1100100 OR 0000001 = 1100101. Similarly, the new bitmap vector of address* 747 Industrial Way *is the vector of* Roberts *OR the vector of* Ellis *= 0010000 OR 0001010 = 0011010.*

As the example above shows, to generate a new bitmap vector of a value u in $\mathbb{R}$, intuitively, we should examine u's bitmap vector in $\mathbb{T}$ to find the key values co-occurred with u, and then combine the bitmap vectors of these key values in $\mathbb{S}$ with an OR operation.

However, such an approach needs to find the key values for each vector of each attribute A_i $(k+1 \le i \le m)$. Consequently, doing so for each value u requires the key values in $\mathbb{T}$ to be randomly accessed, which is not efficient. Therefore, instead of scanning each vector of each attribute, we can sequentially scan each attribute value of each tuple in $\mathbb{T}$, which can generate the same result.

Algorithm 4 Key-Foreign Key Based Merging

MERGING $(\mathbb{S}(A_1, \cdots, A_k, A_{k+1}, \cdots, A_m),$
$\mathbb{T}(A_1, \cdots, A_k, A_{m+1}, \cdots, A_n))$
Output: The bitmaps, hash tables and offset indexes of $A_{m+1}, \cdots, A_n$ in $\mathbb{R}$

1: $R_{\mathbb{S}}$ = number of rows in $\mathbb{S}$
2: $R_{\mathbb{T}}$ = number of rows in $\mathbb{T}$
3: **for** $i = 1$ to $R_{\mathbb{T}}$ **do**
4: $v_1, \cdots, v_k$ = the values of key attributes $A_1, \cdots, A_k$ of $\mathbb{T}$ in row i
5: $u_1, \cdots, u_{n-m}$ = the values of non-key attributes $A_{m+1}, \cdots, A_n$ of $\mathbb{T}$ in row i
6: vec = **findvector**($v_1, \cdots, v_k, \mathbb{S}$)
7: **for** $j = 1$ to $n - m$ **do**
8: **if** u_j has not been seen before in the bitmap of A_{j+m} **then**
9: set vec as the bitmap vector of A_{j+m} for value u_j
10: insert an entry for vec into the hash table of A_{j+m}
11: insert an entry for each position of "1" in vec into the offset index of A_{j+m}
12: **else**
13: vec' = the current vector of A_{j+m} for value u_j
14: $vec = vec'$ OR vec
15: insert an entry for each position of "0" in vec and "1" in vec' into the offset index of A_{j+m}
16: replace the bitmap vector of A_{j+m} for value u_j with vec

Algorithm 4 presents our merging algorithm by realizing the idea just discussed. Line 3 shows the sequential scan of $\mathbb{T}$. For each row during the scan, procedure **findvector** is called to find the vector of the key value of this row in table $\mathbb{S}$ (line 6). The details of the **findvector** procedure will be discussed later. After calling **findvector**, we generate the bitmap vectors for values of non-key attributes (lines 7-16). We check, for each value, whether its bitmap vector exists (line 8). If not, we can directly output the bitmap vector generated in this iteration (line 9). We also insert an entry for this new vector into the hash table of the attribute (line 10). Besides, for each "1" appearing in vec, we insert an entry accordingly to the offset index (line 11). Otherwise, we combine the vector with the previously generated one with OR operation (lines 13-14). During the OR operation, for each position such that the value in vec' is 1 and the value in vec is 0, we insert an entry accordingly into the offset index (line 15).

Employee	Skill
Jones	Typing
Ellis	Alchemy
Ellis	Typing
Jones	Shorthand

$\mathbb{S}$

Employee	Address
Jones	425 Grant Ave
Ellis	747 Industrial Way
Jones	747 Industrial Way
Jones	60 Aubin St
Ellis	501 Oakman Blvd

$\mathbb{T}$

Employee	Address	Skill
Jones	425 Grant Ave	Typing
Jones	747 Industrial Way	Typing
Jones	60 Aubin St	Typing
Ellis	747 Industrial Way	Alchemy
Ellis	501 Oakman Blvd	Alchemy
Ellis	747 Industrial Way	Typing
Ellis	501 Oakman Blvd	Typing
Jones	425 Grant Ave	Shorthand
Jones	747 Industrial Way	Shorthand
Jones	60 Aubin St	Shorthand

$\mathbb{R}$

Figure 3: A Non-Reusable Merging Example

EXAMPLE 4.2. *In our running example, we sequentially scan table* $\mathbb{T}$*. The first tuple of* $\mathbb{T}$ *contains employee* Jones*.* Jones*'s vector in* $\mathbb{S}$ *is 1100100, and thus the vector of the corresponding non-key attribute address* 425 Grant Ave *in* $\mathbb{R}$ *is 1100100. Meanwhile, we insert an entry for this vector into the hash table of* address*, and insert three entries (1,* 747 Industrial Way*), (2,* 747 Industrial Way*) and (5,* 747 Industrial Way*) into the offset index of* address*. The employee of* $\mathbb{T}$*'s second tuple* Roberts *has vector 0010000 in* $\mathbb{S}$*, so we set the vector of the corresponding address* 747 Industrial Way *in* $\mathbb{R}$ *as 0010000, and update the hash table and offset index accordingly. The employee of the third tuple in* $\mathbb{T}$ *is* Ellis*, which has vector 0001010 in* $\mathbb{S}$*. Since the corresponding address,* 747 Industrial Way*, has occurred before, we perform an OR operation on its existing vector and get 0010000 OR 0001010 = 0011010. We also insert two entries, (4,* 747 Industrial Way*) and (6,* 747 Industrial Way*) into the offset index of* address*. Similarly, the last tuple in* $\mathbb{T}$ *has address* 425 Grant Ave*, and we thus preform an OR operation with its existing vector and get 1100101.*

The **findvector** function takes two parameters. The first parameter v is a list of values from key attributes, and the second is a table $\mathbb{X}$. It returns the vector of occurrences of v in $\mathbb{X}$. If v has only a single attribute A, we simply return the vector corresponding to v in A's bitmap. If v has multiple attributes, we perform an AND operation on the vector of each individual value in v, and return the result vector.

4.2 General Merging

When neither input table can be reused, the merging operation has to generate bitmaps for all the attributes in $\mathbb{S}$ and $\mathbb{T}$. This is the most complicated scenario, which happens when the common attributes of the two input tables are not the key of either of them.

As an example, consider Figure 3. It is a variation of Figure 1 such that each employee may have multiple addresses. This means that the join attribute *Employee* of $\mathbb{S}$ and $\mathbb{T}$ are the key of neither $\mathbb{S}$ nor $\mathbb{T}$. As we can see, during the merging

Employee	Skill
Jones	Typing
Ellis	Alchemy
Ellis	Typing
Jones	Shorthand

$\mathbb{S}$

Employee	Address
Jones	425 Grant Ave
Ellis	747 Industrial Way
Jones	747 Industrial Way
Jones	60 Aubin St
Ellis	501 Oakman Blvd

$\mathbb{T}$

Employee	Address	Skill
Jones	425 Grant Ave	Typing
Jones	425 Grant Ave	Shorthand
Jones	747 Industrial Way	Typing
Jones	747 Industrial Way	Shorthand
Jones	60 Aubin St	Typing
Jones	60 Aubin St	Shorthand
Ellis	747 Industrial Way	Alchemy
Ellis	747 Industrial Way	Typing
Ellis	501 Oakman Blvd	Alchemy
Ellis	501 Oakman Blvd	Typing

$\mathbb{R}$

Figure 4: Non-Reusable Merging with Re-Organization.

operation, the columns of $\mathbb{R}$ cannot reuse existing columns of either $\mathbb{S}$ or $\mathbb{T}$.

Intuitively, this case demands a more complex algorithm, as we are not only unable to reuse existing tables, but face difficulties to efficiently determine the positions of attribute values in $\mathbb{R}$. For example, in Figure 3, we cannot compute the positions of the employee *Jones* in $\mathbb{R}$ only from *Jones*'s bitmaps in $\mathbb{S}$ and $\mathbb{T}$. Instead, we have to consider the join results of all employee values to accurately compute the positions of every employee. Conducting joins and holding the join results of the key attribute in memory are neither time nor space efficient.

We solve the general merging problem by developing an efficient merging algorithm to output $\mathbb{R}$ in a specific organization. Figure 4 shows the re-organized $\mathbb{R}$. We can see that tuples in $\mathbb{R}$ are first clustered by the join attributes (i.e., *Employee*), then by attributes in $\mathbb{T}$ (i.e., *Address*), and finally by attributes in $\mathbb{S}$ (i.e., *Skill*). This re-organized $\mathbb{R}$ is the same as the one in Figure 3 from relational perspective. But it can be generated more quickly, since we can derive the positions of attribute values in a much easier way.

In particular, we design a two-pass algorithm to quickly generate the target table for the non-reusable merging scenario. Since other scenarios can be viewed as simpler cases of the non-reusable scenario, this two-pass algorithm is actually a general merging algorithm to handle any merging situations, although in practice, it is mainly used for the non-reusable merging scenario.

The first pass is only on the join attributes of $\mathbb{S}$ and $\mathbb{T}$. In this pass, we compute the number of occurrences of each distinct join value in $\mathbb{S}$ and $\mathbb{T}$. After this pass, we are able to easily generate the bitmaps for the join attributes: If a join value v has n_1 occurrences in $\mathbb{S}$ and n_2 occurrences in $\mathbb{T}$, then it has $n_1 \times n_2$ occurrences in $\mathbb{R}$. Further, since $\mathbb{R}$ is clustered by the join attributes, the bitmap vector of each value, as well as the hash table and offset index of each attribute, can be directly derived from the occurrence counts.

EXAMPLE 4.3. *Consider Figure 4. We scan the distinct values of the join attribute* Employee. *For each distinct* Employee *value, we count how many times it occurs in $\mathbb{S}$ and $\mathbb{T}$, and store the counting results in a hash structure. As we can see,* Jones *appears three times in $\mathbb{S}$ and twice in $\mathbb{T}$;* Ellis *appears twice in both $\mathbb{S}$ and $\mathbb{T}$.*

Based on this result, we can easily generate the bitmap of Employee *in $\mathbb{R}$:* Jones *occurs $2 \times 3 = 6$ times and thus its vector is 1111110000.* Ellis *occurs $2 \times 2 = 4$ times and thus its vector is 0000001111. The hash table and offset index for* Employee *are also constructed accordingly.*

Algorithm 5 General Merging

MERGING ($\mathbb{S}(A_1, \cdots, A_k, A_{k+1}, \cdots, A_m)$, $\mathbb{T}(A_1, \cdots, A_k, A_{m+1}, \cdots, A_n)$)
Output: $\mathbb{R}(A_1, \cdots, A_n)$

```
keyocc = empty hash table
for each distinct value v of the join attributes A_1, ..., A_k do
    occ_S = number of occurrences of v in S
    occ_T = number of occurrences of v in T
    keyocc{v, S} = occ_S
    keyocc{v, T} = occ_T
generate bitmaps, hash tables and offset indexes for join attributes
processTable (S, k, keyocc, consecutive)
processTable (T, k, keyocc, non-consecutive)
```

PROCESSTABLE ($\mathbb{X}, k, keyocc, type$)

```
keyelapsed = empty hash table
rowelapsed = 0
for each distinct value v of join attribute in X do
    occ = keyocc{v, X}
    occ' = keyocc{v, Y}, where Y is the other input table
    vec = findvector (v, X)
    keypos = the positions of "1" in vec
    for i = 1 to keypos.size do
        u_1, ..., u_t = the values of non key attributes of X in row
          keypos[i]
        pos = keyelapsed{u_1, ..., u_t}, set to 0 if not exists
        for j = 1 to t do
            if type = consecutive then
                set occ' bits of the vector of u_j to be 1, from the
                  (rowelapsed+occ'*pos)th bit to the (rowelapsed+
                  occ'*(pos+1)-1)th bit
                insert the corresponding entries into the corre-
                  sponding offset index
            else
                for k = 0 to occ'-1 do
                    set the (rowelapsed+k*occ+pos)th bit of the
                      vector of u_j to be 1
                    insert the corresponding entries into the corre-
                      sponding offset index
            encode the new added bits in the vector of u_j if nec-
              essary using WAH encoding
        keyelapsed{u_1, ..., u_t} = pos + 1
    rowelapsed += occ × occ'
    insert an entry into for each vector of non-key attribute of
      X into the corresponding hash table
```

In the second pass, for each distinct join value, we find the value of other attributes in $\mathbb{S}$ and $\mathbb{T}$. For values of non-key attributes in $\mathbb{T}$ (or $\mathbb{S}$), we put them in a consecutive way and thus can correspondingly compute the positions for each value. For values of non-key attributes in $\mathbb{S}$ (or $\mathbb{T}$), we put them in a non-consecutive way but with the same distance and thus we can also correctly compute the positions for each value.

EXAMPLE 4.4. *In Figure 4, we first process the attributes in table $\mathbb{T}$. For employee* Jones, *we have three matching addresses:* 425 Grant Ave, 747 Industrial Way *and* 60 Aubin St. *Since the number of occurrences of* Jones *in $\mathbb{S}$ is 2, we know that each of these addresses will appear twice in $\mathbb{R}$. As we organize the values in a consecutive way, the address* 425

Grant Ave *is put in the first two positions and thus has a bitmap vector 1100000000 in* $\mathbb{R}$. *Similarly,* 747 Industrial Way *has vector 0011000000, and* 60 Aubin St *0000110000.*

We then process the attributes in $\mathbb{S}$. *For employee* Jones, *we have two matching skills:* Typing *and* Shorthand. *Since the number of occurrences of* Jones *in* $\mathbb{T}$ *is 3, we know that each of these skills will appear three times in* $\mathbb{R}$. *As we organize the values in a non-consecutive way, skill* Typing *is put in the 1st, 3rd and 5th positions and thus has a bitmap vector 1010100000 in* $\mathbb{R}$. *Similarly,* Shorthand *has a vector 0101010000.*

Addresses and skills of employee Ellis *are processed in a similar way. The offset index and hash table of each attribute is constructed according to the bit vector.*

Algorithm 5 shows the general two-pass merging algorithm. Lines 1-7 realize the first pass, and lines 8-9 the second pass by calling function *processTable* twice to process table $\mathbb{T}$ and $\mathbb{S}$. We use $\mathbb{T}$ to illustrate the *processTable* procedure below, and the process on $\mathbb{S}$ is similar. For each distinct join attribute value v in $\mathbb{T}$, we first get the occurrences of v in $\mathbb{T}$ as *occ* (line 4) and its occurrences in $\mathbb{S}$ as occ' (line 5). We then find the positions of v in $\mathbb{T}$, denoted as *keypos* (lines 6-7). Function **findvector** has been discussed in Section 4.1. Each of these positions corresponds to a list of attribute values in $\mathbb{T}$, $A_{m+1}, \cdots, A_n$ (line 9). For each attribute value $A_i (m+1 \leq i \leq n)$, we construct its corresponding vector in $\mathbb{R}$ in either consecutive or non-consecutive way (lines 10-20).

5. EXPERIMENTS

In this section, we report the results of a set of experiments designed to evaluate our proposed data-level data evolution on column oriented databases. We test the efficiency and scalability in terms of time consumed for data evolution, on both real and synthetic data with large sizes.

5.1 Experimental Setup

Environment. The experiments were conducted on a machine with an AMD Athlon 64 X2 Dual Core processor of 3.0GHz and 4.0GB main memory. In our implementation we used the Windows file system, i.e., the tables in the column store are simulated by a set of files storing the columns, and did not take advantage of the buffer and disk management which is usually used in DBMSs.

Comparison System. We tested two baseline approaches for query-level data evolution, one for row store and one for column store. For decomposition, the baseline approach for row stores scans the input table twice, as it is the minimal cost involved in the decomposition (one scan for each query), and writes the output tables; the baseline approach for column stores scans the affected columns and output the corresponding columns in the output table. For merging, both baseline approaches scan the input tables twice (assuming hash join is used), and writes the output table. The baseline approaches for row stores and column stores are denoted as "BR" and "BC" in the following figures, respectively.

We also tested two RDMBSs: a commercial row-oriented RDBMS (denoted as "C" in the figures), and an open source column oriented RDBMS (LucidDB [1], denoted as "L"). We disabled logging in C for fair comparison. We use "D" to refer to our data-level approach.

Data Set. Both synthetic data and real data are tested. We generated synthetic tables based on five parameters: number of tuples, number of distinct values in a column, number of affected columns in a decomposition/merging, number of unaffected columns in a decomposition/merging, and the width of data values. The tuples in the synthetic data are randomly generated. In each set of experiments, we varied one parameter and fixed the others.

When testing decomposition and key-foreign key based merging, we generated a table $\mathbb{R}$ with a functional dependency, then we decomposed it into two tables $\mathbb{S}$ and $\mathbb{T}$ to test the decomposition and merge $\mathbb{S}$ and $\mathbb{T}$ back to test the merging. Unless otherwise stated, table $\mathbb{R}$ has three columns (A_1, A_2, A_3); A_1 is the primary key and A_3 functionally depends on A_2. $\mathbb{R}$ has 10 million tuples, A_2 has 100 thousand distinct values and A_3 has 10 thousand distinct values. The average width of the values in $\mathbb{R}$ is 5 bytes. When testing general merging, we generated two joinable tables $\mathbb{S}$ and $\mathbb{T}$. Unless otherwise stated, $\mathbb{S}$ and $\mathbb{T}$ both have two columns (A_1, A_2) and (A_2, A_3), and 1 million tuples. Merging $\mathbb{S}$ and $\mathbb{T}$ can result in as many as 2 billion tuples in the output table $\mathbb{R}$. A_2 has 40 thousand distinct values. The number of distinct values of A_1 and A_3 are set to be half and one third of the number of distinct values of A_2, respectively.

For real data, we used a table in the patent database, which can be purchased from the United States Patent and Trademark Office. The table has 25 million tuples and nine attributes: patent number (key), name, first name, last name, address, country, state, city, zip. The total size of this table is roughly 30GB. The compressed bitmaps of all data sets in the experiment fit into main memory.

In the experiment we measure the time taken by each approach for data evolution.

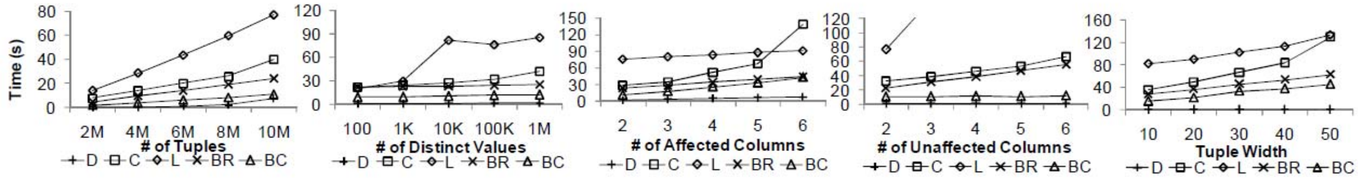
5.2 Performance of Decomposition on Synthetic Data

In this set of experiments we test the efficiency of decomposition for our approach, the two baseline approaches and query-level data evolution on the two RDBMSs. The test results are shown in Figure 5.[3]

Figure 5 shows the processing time of decomposition for all approaches with respect to different parameters. The horizontal axis in Figure 5(b) corresponds to the number of distinct values of A_2. The number of distinct value of A_3 is set to be 1/10 of that of A_2.

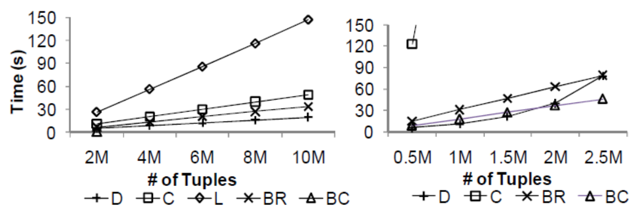
As we can see, the processing time of decomposition of each approach generally increases with the five parameters, and our data-level approach significantly outperforms the other approaches by up to several degrees of magnitude. Our approach is much faster even compared with the baseline approaches, which only perform scans of the tables / columns, and writes the output tables. The processing time of our approach is mainly due to the process of building the new bitmap of A_3 according to the occurrences of distinct values of A_2 in $\mathbb{R}$, which is proportional to the number of distinct values of A_2. Since the bitmap of each column is compressed, the size of the bitmap increases much more slowly than the number of tuples. On the other hand, table scans and writing results onto disks can take much longer time, as a column is usually much larger than its compressed bitmap. Note that BC is generally faster than BR as it does not need

[3]Some data points are missing as the corresponding processing time exceeds the scope of the vertical axes.



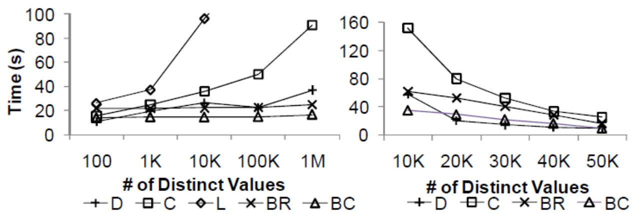
(a) (b) (c) (d) (e)

Figure 5: Processing Time of Decomposition



(a) Key-Foreign Key Based Merging (b) General Merging

Figure 6: Processing Time of Merging, Varying Number of Tuples



(a) Key-Foreign Key Based Merging (b) General Merging

Figure 7: Processing Time of Merging, Varying Number of Distinct Values

to scan or output the unaffected columns.

The processing time of decomposition of the two RDBMSs involves a projection query and a distinction query. C takes a relatively long time for the projection query, as the number of disk I/Os is large. Since L uses bitmap indexes, it is efficient for distinction queries. However, L is generally inefficient as it takes a long time to rebuild the bitmap index after the new tables are generated.

Besides, Figure 5(d) and (e) suggest that our approach is almost not affected by increasing the number of unaffected columns, as they can be reused, and by increasing the tuple width, as the majority of our operations are on bitmaps.

5.3 Performance of Merging on Synthetic Data

In this section we report the merging processing time (including key-foreign key based merging and general merging) for these approaches. Note that we use smaller input tables for general merging, but the size of the output table of general merging is comparable with the output table of key-foreign key based merging. Since L is extremely slow for both key-foreign key based merging and general merging (taking more than 12 hours for a single join query), we do not include L in the corresponding merging test.

The processing time of key-foreign key based merging and general merging of each approach with respect to different parameters is shown in Figure 6 - Figure 8. As we can see, our approach usually outperforms the baseline approaches (which merely scans the tables and outputs the results, but do not perform any join), and is slower but comparable with the BC approach in a few cases. In fact, performing joins take a long time, indicated by the fact that our approach is far more efficient than C and L. For key-foreign key based merging, our approach scans the key attributes of the input table $\mathbb{S}$ once, which is the smaller of the two input tables. There are some additional bitmap "OR" operations, which are efficient for compressed bitmaps. For general merging, the processing time of the baseline approaches is dominated by the size of the output table, as it is much bigger than both input tables. On the other hand, our approach do not outputs this table, but its bitmaps, which is usually several hundred times smaller.

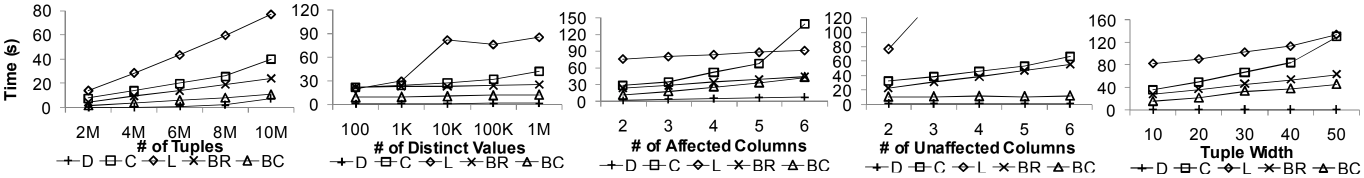
Our approach is far more efficient and scalable than C and L. As an example, when the number of tuples is 2.5 million, C takes almost an hour for general merging while our approach consumes only 78 seconds; when the number of affected columns is 6, C and L take 68 and 433 seconds for key-foreign key based merging, while our approach takes 33 seconds.

Note that, as can be seen from Figure 7(b), the processing time of general merging of all approaches decreases when the number of distinct values increase, as the number of tuples in the output table decreases with increasing number of distinct values. Besides, similar as decomposition, the processing time of our approach is almost not affected by the number of unaffected columns.

5.4 Performance of Decomposition and Merging on Real Data

In the patent table, attributes "first name" and "last name" functionally depend on "name". Therefore we test a decomposition which separates these three attributes into a new table. A merging is also tested as the reverse of the decomposition. LucidDB is not included in this test due to its extremely slow speed for processing this large table.

The test shows that our approach is significantly faster than the commercial database on this data set. We take 71 and 133 seconds to perform the decomposition and merging. Contrarily, C takes 308 seconds to perform the decomposition. For the merging, C does not finish in one hour, largely due to the poor scalability of C. Such a performance makes data evolution prohibitive and prevents it from being used.



(a) Key-Foreign Key (b) General (c) Key-Foreign Key (d) Key-Foreign Key (e) General

Figure 8: Processing Time of Merging, Varying Width of Values and Number of Columns

6. RELATED WORKS

Column Oriented Databases. Notable database management systems that store data via columns include MonetDB [12], LucidDB [1] and C-Store [33]. They all support relational databases and data warehouses, on which SQL queries can be executed. C-Store additionally supports hybrid structures of both row oriented and column oriented storages, as well as overlapping columns to speed up query processing. A detailed investigation of compression schemes on column oriented databases and guidance for choosing these schemes are presented in [5]. [19] compares the performance of column oriented databases and row oriented databases with respect to various factors, such as the number of columns accessed by a query, L2 cache prefetching, selectivity, tuple width, etc., and demonstrates that column oriented databases are in general more efficient than row oriented databases for answering queries that do not access many attributes. [3] studies how column stores handle wide tables and sparse data. An optimization strategy that joins columns into rows as late as possible when answering queries is presented in [7]. [4] shows that column oriented databases are well suitable for handling vertically partitioned RDF data, which achieves an order of magnitude improvement in efficiency compared with row oriented databases. [6] conducts a comprehensive study of the fundamental differences between row stores and column stores during query processing, investigates whether the performance advantage of column stores can be achieved by row stores using vertical partitioning, as well as how much each optimization strategy affects the performance of column stores. All these works on column stores study the query processing efficiency, while data evolution on column stores has not been addressed.

Database Evolution and Related Topics. A database evolution generally involves updating the schema of the database, and evolving the data to the new schema. Existing research on database evolution studies procedures and effects of database schema updates. Clio [26] is a system that enables users to express schema mappings in terms of queries, and then uses query-level data evolution to evolve the data between schemas on XML or relational databases. Other works include the impact-minimizing methodology [30], propagations of the changes of applications to databases [20], consistency preservation upon schema changes [14] and frameworks of metadata model management [11, 34, 37]. Recently, [15] presents the requirements of an operational tool for schema update, e.g., interface, monitoring of redundancy, query rewriting, documentation, etc., but it lacks algorithms that actually evolve the data from the original schema to the new one.

Note that the data evolution problem studied in this paper is quite different from the data migration techniques studied and applied in both industry and academia. Commercial DBMSs, such as DB2, Oracle and SQL Server, provide support for data migration, which is the process of migrating a database between different DBMSs. However, the database schema generally does not change during this process. Data migration is also regarded as a disk scheduling problem [22, 24, 23], which focuses on scheduling disk communications such that a set of data items are transferred from the source disks to the target disks in minimal time. Various approximation algorithms are proposed to tackle this NP-hard problem.

Other related features supported in commercial DBMS products include the Oracle Change Management Pack [2], which provides useful operations to manage change to databases, e.g., it allows developers to track changes to objects as the database schema changes, and to quickly assess the impact of a database evolution and modify the relevant applications accordingly. Besides, online data reorganization in commercial DBMSs is a technique that enables the database to be usable when it is being maintained.

7. CONCLUSIONS AND FUTURE WORK

In this paper we present a framework for efficient data-level data evolution on column oriented databases, based on the observation that column oriented databases have a storage scheme better suitable for database evolution. Traditionally, data evolutions are conducted by executing SQL queries, which we name as query level data evolution. Such a time consuming process severely limits the usability and convenience of the databases.

We exploit the characteristics of column stores and propose a data-level data evolution which, instead of joining columns to form results of SQL queries and then separately compressing and storing each column, evolves the data directly from compressed storage to compressed storage. Both analysis and experiments verify the plausibleness of our approach and show that it consumes much less time and scales better than query level data evolution. Our approach makes databases more usable and flexible, relieves the burden of users and system administrators, and encourages research on further utilization of database evolution. This framework also guides the choice between row oriented databases and column oriented databases when schema changes are potentially wanted. In the future, we would like to study whether the algorithm of merging tables can be applied to optimize join queries, and the applications with excessively large tables such that the bitmap vector of a single attribute does not fit in main memory, as well as the cases in which columns are not compressed using bitmap.

8. ACKNOWLEDGEMENT

This material is based on work partially supported by NSF CAREER Award IIS-0845647, IIS-0915438 and IBM Faculty Award.

9. REFERENCES

[1] LucidDB. http://www.luciddb.org/.

[2] Oracle Enterprise Manager 10g Change Management Pack, 2005. Oracle Data Sheet.

[3] D. J. Abadi. Column Stores for Wide and Sparse Data. In *CIDR*, pages 292–297, 2007.

[4] D. J. Abadi, A. M. 0002, S. Madden, and K. J. Hollenbach. Scalable Semantic Web Data Management Using Vertical Partitioning. In *VLDB*, pages 411–422, 2007.

[5] D. J. Abadi, S. Madden, and M. Ferreira. Integrating Compression and Execution in Column-Oriented Database Systems. In *SIGMOD Conference*, pages 671–682, 2006.

[6] D. J. Abadi, S. Madden, and N. Hachem. Column-Stores vs. Row-Stores: How Different Are They Really? In *SIGMOD Conference*, pages 967–980, 2008.

[7] D. J. Abadi, D. S. Myers, D. J. DeWitt, and S. Madden. Materialization Strategies in a Column-Oriented DBMS. In *ICDE*, pages 466–475, 2007.

[8] R. B. Almeida, B. Mozafari, and J. Cho. On the Evolution of Wikipedia. In *ICWSM*, 2007.

[9] G. Antoshenkov. Byte-Aligned Bitmap Compression. In *DCC*, 1995.

[10] Z. Bellahsene. Schema Evolution in Data Warehouses. *Knowl. Inf. Syst.*, 4(3):283–304, 2002.

[11] P. A. Bernstein. Applying Model Management to Classical Meta Data Problems. In *CIDR*, 2003.

[12] P. A. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*, pages 225–237, 2005.

[13] Y. Chen, W. S. Spangler, J. T. Kreulen, S. Boyer, T. D. Griffin, A. Alba, A. Behal, B. He, L. Kato, A. Lelescu, C. A. Kieliszewski, X. Wu, and L. Zhang. SIMPLE: A Strategic Information Mining Platform for Licensing and Execution. In *ICDM Workshops*, pages 270–275, 2009.

[14] A. Cleve and J.-L. Hainaut. Co-transformations in Database Applications Evolution. In *GTTSE*, pages 409–421, 2006.

[15] C. Curino, H. J. Moon, and C. Zaniolo. Graceful Database Schema Evolution: the PRISM Workbench. *PVLDB*, 1(1):761–772, 2008.

[16] C. Curino, H. J. Moon, and C. Zaniolo. Automating Database Schema Evolution in Information System Upgrades. In *HotSWUp*, 2009.

[17] F. Deliège and T. B. Pedersen. Position List Word Aligned Hybrid: Optimizing Space and Performance for Compressed Bitmaps. In *EDBT*, pages 228–239, 2010.

[18] H. Fan and A. Poulovassilis. Schema Evolution in Data Warehousing Environments - A Schema Transformation-Based Approach. In *ER*, pages 639–653, 2004.

[19] S. Harizopoulos, V. Liang, D. J. Abadi, and S. Madden. Performance Tradeoffs in Read-Optimized Databases. In *VLDB*, pages 487–498, 2006.

[20] J.-M. Hick and J.-L. Hainaut. Database Application Evolution: A Transformational Approach. *Data Knowl. Eng.*, 59(3):534–558, 2006.

[21] H. V. Jagadish, A. Chapman, A. Elkiss, M. Jayapandian, Y. Li, A. Nandi, and C. Yu. Making database systems usable. In *SIGMOD*, 2007.

[22] S. Khuller, Y. A. Kim, and A. Malekian. Improved Algorithms for Data Migration. In *APPROX-RANDOM*, pages 164–175, 2006.

[23] S. Khuller, Y. A. Kim, and Y.-C. J. Wan. Algorithms for Data Migration with Cloning. In *PODS*, pages 27–36, 2003.

[24] Y. A. Kim. Data Migration to Minimize the Total Completion Time. *J. Algorithms*, 55(1):42–57, 2005.

[25] Z. Liu, S. Natarajan, B. He, H.-I. Hsiao, and Y. Chen. CODS: Evolving Data Efficiently and Scalably in Column Oriented Databases. *PVLDB*, 3(2):1521–1524, 2010.

[26] R. J. Miller, M. A. Hernández, L. M. Haas, L.-L. Yan, C. T. H. Ho, R. Fagin, and L. Popa. The Clio Project: Managing Heterogeneity. *SIGMOD Record*, 30(1):78–83, 2001.

[27] H. J. Moon, C. Curino, A. Deutsch, C.-Y. Hou, and C. Zaniolo. Managing and Querying Transaction-time Databases under Schema Evolution. *PVLDB*, 1(1):882–895, 2008.

[28] P. E. O'Neil and D. Quass. Improved Query Performance with Variant Indexes. In *SIGMOD Conference*, pages 38–49, 1997.

[29] G. Özsoyoglu and R. T. Snodgrass. Temporal and Real-Time Databases: A Survey. *IEEE Trans. Knowl. Data Eng.*, 7(4):513–532, 1995.

[30] Y.-G. Ra. Relational Schema Evolution for Program Independency. In *CIT*, pages 273–281, 2004.

[31] R. Ramakrishnan and J. Gehrke. *Database Management Systems, 3rd Edition.* McGraw Hill Higher Education, 2002.

[32] K. Stockinger, J. Cieslewicz, K. Wu, D. Rotem, and A. Shoshani. Using Bitmap Index for Joint Queries on Structured and Text Data, 2008.

[33] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. J. O'Neil, P. E. O'Neil, A. Rasin, N. Tran, and S. B. Zdonik. C-Store: A Column-oriented DBMS. In *VLDB*, pages 553–564, 2005.

[34] Y. Velegrakis, R. J. Miller, and L. Popa. Mapping Adaptation under Evolving Schemas. In *VLDB*, pages 584–595, 2003.

[35] K. Wu, E. J. Otoo, and A. Shoshani. On the Performance of Bitmap Indices for High Cardinality Attributes. In *VLDB*, pages 24–35, 2004.

[36] K. Wu, E. J. Otoo, and A. Shoshani. Optimizing bitmap indices with efficient compression. *TODS*, 31(1), 2006.

[37] C. Yu and L. Popa. Semantic Adaptation of Schema Mappings when Schemas Evolve. In *VLDB*, pages 1006–1017, 2005.