CODS: Evolving Data Efficiently and Scalably in Column Oriented Databases *

Ziyang Liu[1], Sivaramakrishnan Natarajan[1], Bin He[2], Hui-I Hsiao[2], Yi Chen[1]
Arizona State University[1], IBM Almaden Research Center[2]
{ziyang.liu,snatara5,yi}@asu.edu[1]
binhe@us.ibm.com[2]
hhsiao@almaden.ibm.com[2]

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). Database evolution is often necessitated in relational databases due to the changes of data or workload, the suboptimal initial schema design, or the availability of new knowledge of the database. It involves two steps: updating the database schema, and evolving the data to the new schema. Despite the capability of commercial RDBMSs to well optimize query processing, evolving the data during a database evolution through SQL queries is shown to be prohibitively costly. We designed and developed CODS, a platform for efficient *data level* data evolution in column oriented databases, which evolves the data to the new schema without materializing query results or unnecessary compression/decompression as occurred in traditional *query level* approaches. CODS ameliorates the efficiency of data evolution by orders of magnitude compared with commercial or open source RDBMSs.

1. INTRODUCTION

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). Database evolution occurs in relational databases and data warehouses with a good deal of regularity [5, 7], aiming at adapting the database for optimal performance up against the emergence and/or change to the data or workload. The Wikipedia database, for example, has had more than 170 versions in the past 5 years [4]. The scenarios where a database evolution is necessitated or highly desirable include but are not limited to:

1. **New Information about the Data.** Consider table $\mathbb{R}$ in Figure 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 $\mathbb{R}$. Besides, 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 $\mathbb{R}$ to two tables $\mathbb{S}$ and $\mathbb{T}$, as shown in schema 2, in order to prevent data redundancy and update anomaly.
2. **New Information about the Workload.** Assume that the original workload on Figure 1 is update intensive, for which Schema 2 is desirable. Later workload characteristics change to become query intensive, and most queries look for addresses given skills. Now Schema 1 becomes more suitable, as it avoids joins of two tables. One typical example is in data warehouse applications, when workload evolves, we choose to change star schema to snowflake schema, or vice versa.

schema 1

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

$\mathbb{R}$

schema 2

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

$\mathbb{S}$

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

$\mathbb{T}$

Figure 1: Sample Database Evolution

Database evolution consists of two steps: schema update and data evolution. Existing work on database evolution mainly studies the first step, specifically, the interfaces and operators needed for schema update and the maintenance of associated views/applications. A system for supporting automatic schema evolution, the PRISM workbench [5], provides support for predicting the effect of schema update, implementing logical independence, improving audibility, rewriting queries, etc. However, efficient algorithms for evolving the data from the original schema to the new schema is yet to be investigated. Currently, data evolution is expressed and executed at *query level*, i.e., via SQL queries. As an example, the following

*This material is based on work partially supported by NSF CAREER award IIS-0845647, IIS-0740129, IIS-0915438.

Proceedings of the VLDB Endowment, Vol. 3, No. 2

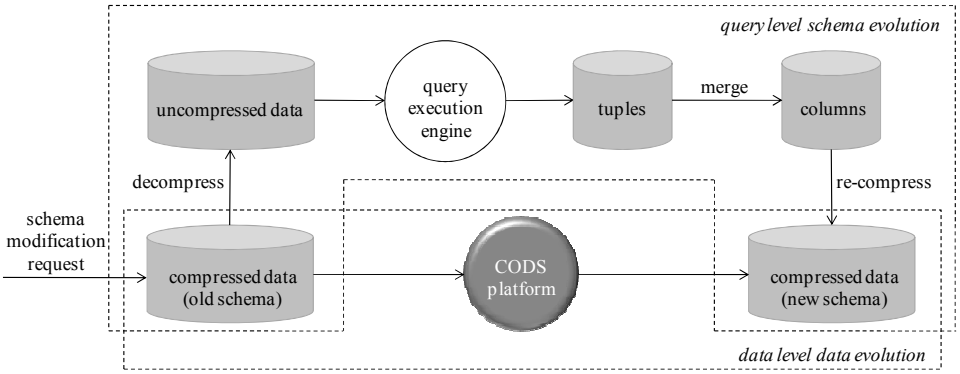

Figure 2: Architecture of Data Level vs. Query Level Data Evolution on Column Oriented Databases

SQL queries are executed to evolve the data from $\mathbb{R}$ to $\mathbb{S}$ and $\mathbb{T}$ in Figure 1:

1. INSERT INTO $\mathbb{S}$ SELECT EMPLOYEE, SKILL FROM $\mathbb{R}$
2. INSERT INTO $\mathbb{T}$ SELECT DISTINCT EMPLOYEE, ADDRESS FROM $\mathbb{R}$

Such a query level approach is notoriously costly 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 [5]. 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.

To tackle this challenge, we observe that, when evolving the data to the new schema, usually many columns in a table remain unchanged. Therefore, data evolution can potentially be much more efficient on column oriented databases, where each column is stored separately, and thus unchanged columns can be reused.[1] In the simplest case, to add (or drop) a column in a column store, other columns do not need to be affected at all. However, to fully optimize data evolution on column stores in supporting of general schema updates, two technical challenges need to be addressed. First, is it possible to directly evolve the data from column oriented storage to column oriented storage? In other words, can we do better than query-level data evolution that requires to generate the query results and re-build the indexes on the results? Second, values in a column store are typically compressed; is it possible to evolve the data without decompression and re-compression?

To address these challenges, we present CODS (Column Oriented Database Schema update) in this demo, a platform that handles data evolution on column oriented databases efficiently. It allows users to specify a schema update using one of the Schema Modification Operators [5], and evolves the data to the new schema with orders of magnitude of improved efficiency compared with query level approaches on commercial and open source RDBMSs.

By enabling efficient data evolution, CODS makes databases more flexible and agile to handle evolving data and workload. Besides, CODS also guides the choice between row oriented databases and column oriented databases when schema changes are potentially wanted, and brings opportunities for research on further utilization of efficient data evolution.

[1]"column oriented database" and "column store" are used interchangeably in this paper.

Table 1: Operations for Schema Update

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. SYSTEM OVERVIEW

2.1 Architecture of CODS

The architecture of data level data evolution adopted in CODS and that of traditional query level data evolution on column oriented databases are shown in Figure 2. As we can see, query-level data evolution in a column oriented database is very expensive. First, we need to generate query results, during which the columns need to be materialized into tuples. Then, the tuples have to be broken up into columns, and finally, each column of the results needs to be compressed and stored, which incurs high costs. On the other hand, CODS directly operates on the compressed columns (usually encoded as compressed bitmaps [8, 9]) which are affected by the schema change, and generates the compressed columns corresponding to the new schema without the need of decompression or re-compression.

2.2 Data Storage

In column stores, the data in each column are stored contiguously on the storage. 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. Bitmap is the most common encoding scheme used in column stores [3, 9]. A

bitmap for a column can be viewed as a $v \times r$ matrix, where v is the number of distinct values and r is the number of rows. Each value in the column corresponds to a vector of length r in the bitmap, in which the kth position is 1 if this value appears in the kth row, and 0 otherwise. To reduce the storage size and improve performance, a bitmap is typically compressed when a column has a reasonable number of duplicate values. Among existing compression schemes for bitmaps, WAH [9] is the state-of-the-art which support query processing on compressed data directly, and is adopted in our implementation. Other compression schemes are sometimes used for special columns, such as run length encoding for sorted columns. Supporting other compression/encoding schemes is one of our future works.

2.3 Operations in Schema Update

We support all the Schema Modification Operators (SMO) as introduced in [5], which are listed in Table 1. Among these operations, *Decompose Table* and *Merge Tables* are the most challenging ones, as they involve major change to the underlying data. *Create, Drop,* and *Rename Table* operations incur mainly schema level operations while *Copy, Union and Partition Table* operations require data movement, but no data change. Thus, they are relatively straightforward. For column-level SMO, *Add Column* and *Drop Column* can also be supported easily. Due to space reasons, in the following we focus the discussion on *Decompose* and *Merge* operations.

2.4 Decomposition Operation

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, 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.

We observe two properties of a lossless-join decomposition, which are exploited for efficient decompositions.

1. In a lossless-join decomposition which decomposes a table into two, at least one of the two output tables is unchanged from the original one.
2. For a changed output table, its non-key attributes have functional dependency on its key attributes in the original table.

The proof is omitted and can be found in [6].

Property 1 can be effectively utilized by column stores. 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 a data evolution, and thus save computation time.

Without loss of generality, 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_m), and (A_1, $\cdots$, A_k, A_{m+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. Then, 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. This step is referred to as "*distinction*".

2) Given the list of positions of the key attribute values of $\mathbb{T}$ in $\mathbb{R}$, we can directly generate new bitmaps of $\mathbb{T}$'s attributes from their corresponding bitmaps in $\mathbb{R}$. Each v presents a list of k values from $\mathbb{T}$'s key attributes. Using Property 2, all tuples in $\mathbb{R}$ with the same v have the same values on $A_{m+1}, \cdots, A_n$. We thus can choose any one of these tuples and do not need to access all of them. For each attribute, we shrink their bitmap in $\mathbb{R}$ by only taking the bits specified in the position list. We name such an operation as "bitmap filtering".

2.5 Mergence Operation

The mergence operation joins two tables to form a new table, such as joining $\mathbb{S}$ and $\mathbb{T}$ in Figure 1 into $\mathbb{R}$. There are two scenarios of mergence: 1) At least one of the input tables can be reused in the mergence (i.e., the columns in this table are the same as the corresponding ones in the output table), and 2) Neither of the input tables can be reused. For scenario 1, 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, such as the mergence of tables $\mathbb{S}$ and $\mathbb{T}$ in Figure 1 into table $\mathbb{R}$. This type of mergence is referred to as *key-foreign key based mergence*. Scenario 2 involves other types of equi-joins, named as *general mergence*.

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 $\mathbb{R}$ ($A_1, \cdots, A_n$).

2.5.1 Key-Foreign Key Based Mergence

Suppose the common attributes of $\mathbb{S}$ and $\mathbb{T}$ comprise the key of $\mathbb{T}$. Thus instead of generating all columns in $\mathbb{R}$, we can reuse the columns in $\mathbb{S}$ (i.e., $A_1, \cdots, A_k, A_{k+1}, \cdots, A_m$) and only generate columns $A_{m+1}, \cdots, A_n$ for $\mathbb{R}$. The new bitmap of each A_i (m+1 $\leq i \leq n$) in $\mathbb{R}$ can be obtained by deploying the original bitmap of A_i in $\mathbb{S}$ and the bitmap of the key attributes in $\mathbb{T}$. To generate a new bitmap vector of a value u in $\mathbb{R}$, we can 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 OR operations.

However, such an approach needs to find the key value corresponding to each value of attribute A_i (m+1 $\leq i \leq n$). Doing so for each value requires the key values in $\mathbb{S}$ 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 $\mathbb{S}$, which can generate the same result. Specifically, we perform a sequential scan of $\mathbb{S}$, and for each attribute value of A_i (m+1 $\leq i \leq n$) in row j of $\mathbb{S}$, we take the bitmap vector of row j's key value in $\mathbb{T}$ and do an OR operation with the existing new vector of that value.

2.5.2 General Mergence

This scenario is more challenging. We are not only unable to reuse existing tables, but face difficulties to efficiently determining the positions of attribute values in $\mathbb{R}$, as we cannot compute the positions of a value in $\mathbb{R}$ using only the vectors of the value in $\mathbb{S}$ and $\mathbb{T}$. We design a two-pass algorithm to quickly generate the target table for this scenario.

The first pass is performed 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, we can cluster $\mathbb{R}$ by the join attributes, thus the bitmap vector of each value can be directly derived from the occurrence counts.

In the second pass, for each distinct join value, we find the match-

ing non-key attributes in $\mathbb{S}$ and $\mathbb{T}$. For values of non-key attributes in $\mathbb{S}$, we put them in a consecutive way and thus can correspondingly compute the positions for each value. For values of non-key attributes in $\mathbb{T}$, we put them in a non-consecutive way but with the same distance and thus can also correctly compute the positions for each value. In this way, we generate the output table $\mathbb{R}$ efficiently. Due to space constraints, we omit the details of our algorithms, which are presented in [6].

2.6 Empirical Evaluation

We have empirically evaluated CODS in comparison with a commercial row-oriented RDBMS product, SQLite [2] (an open source row oriented RDBMS), and MonetDB [1] (an open source column-oriented RDBMS). The test results of decomposing a table $\mathbb{R}$ with 10 million tuples into $\mathbb{S}$ and $\mathbb{T}$, and merging them back to $\mathbb{R}$, are presented in Figure 3. Although full-fledged RDBMSs offer a much large number of functionalities which will likely affect their performance of data evolution, these experiments can serve as an initial empirical study on the data-level approach for data evolution. As can be seen, CODS has a significantly better efficiency and scalability than query-level data evolution on row-oriented databases and column-oriented databases. More experimental evaluations are reported in [6].

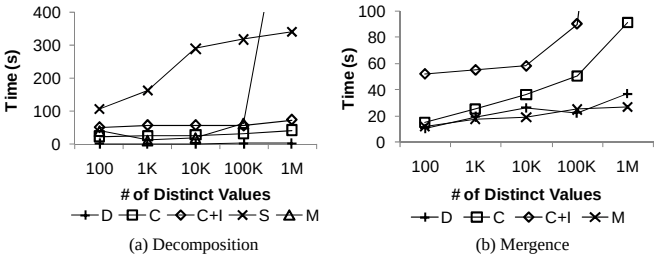

Figure 3: Evaluation. D = Data-Level Approach (CODS), C = Commercial RDBMS, C+I = Commercial RDBMS with Indexes, S = SQLite, M = MonetDB.

3. DEMONSTRATION OUTLINE

In our demonstration we present CODS, the first platform that supports efficient data-level data evolution. The goal of our demonstration is to showcase how CODS can be used to perform data evolutions upon schema changes.

A snapshot of CODS is shown in Figure 4, where a set of operations are supported.

Specifying and Executing Operators for Data Evolution. First, tables need to be specified by clicking the "*create/drop table*" button. Users will specify the attributes of a table, in the same way they create tables using SQL. In Figure 4, for example, three tables are already created. Then, the users can load data into the tables from data files. When clicking "*load data*", users will choose the file location and the table to which the data are loaded. CODS will build a bitmap index for each non-key attribute of a table. The tuples in the tables can be shown by clicking "*display table*".

Then, users need to specify one or more schema modification operators by clicking the "*add*" button. CODS supports the operators for schema update listed in Table 1. Each operator takes some parameters, e.g., if the user chooses "decompose", then s/he needs to specify three tables: one input table and two output tables. In Figure 4, one schema modification operator is specified.

Finally, the schema modification operators will be executed by clicking the "execution" button.

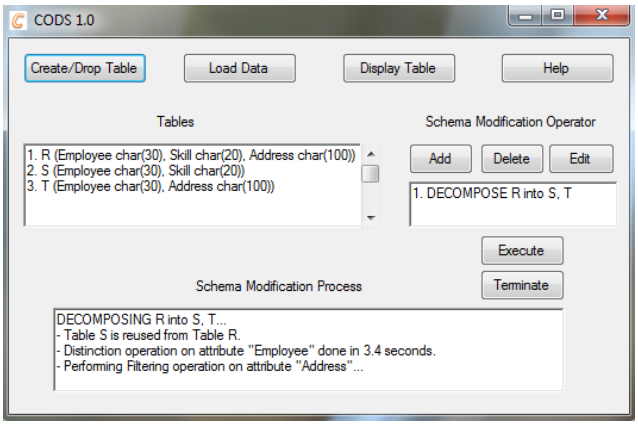

Figure 4: CODS Snapshot

Tracking Data Evolution Status. During the data evolution process, the detailed status will be shown in the "*Data Evolution Status*" part. It shows each step that is taken by CODS during the data evolution, such as "distinction" and "filtering", as introduced in Section 2.4. The users will see that CODS directly generates the output tables from the original ones. On the other hand, the query-level approach first executes queries, then import the results into the new tables, and re-build the indexes if needed.

After a data evolution is finished, the users can choose to display the generated tables (by clicking "*display table*"), or specify further schema modification operators and run them on the new tables.

In summary, since the demands of database evolution occur quite often in databases and data warehouses in order to achieve optimized performance upon change of data or workload, a system for efficiently evolving data to the new schema is remarkably helpful. The buildup of the CODS platform fills a gap in the development of database management systems with high flexibility in terms of schemas. CODS presents the good feasibility of data evolution in column oriented databases, and illustrates how data evolution can be efficiently and scalably achieved. It is of both theoretical and practical use for database researchers, designers, administrators as well as users, guides the choice of row oriented databases versus column oriented databases in applications, and encourages researches on further utilization of efficient data evolution.

4. REFERENCES

[1] MonetDB. http://monetdb.cwi.nl/.

[2] SQLite. http://www.sqlite.org/.

[3] D. J. Abadi, S. R. Madden, and M. C. Ferreira. Integrating Compression and Execution in Column-Oriented Database Systems. In *SIGMOD*, 2006.

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

[5] C. Curino, H. Moon, and C. Zaniolo. Graceful Database Schema Evolution: the PRISM Workbench. In *VLDB*, 2008.

[6] Z. Liu, B. He, H.-I. Hsiao, and Y. Chen. Agile Schema Evolution on Column Oriented Databases. Technical Report TR-09-016, Arizona State University, 2009.

[7] H. Moon, C. Curino, A. Deutsch, C.-Y. Hou, and C. Zaniolo. Managing and Querying Transaction-time Databases under Schema Evolution. In *VLDB*, 2008.

[8] M. Stonebraker, D. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-Store: A Column Oriented DBMS. In *VLDB*, 2005.

[9] K. Wu, E. J. Otoo, and A. Shoshani. Optimizing Bitmap Indices with Efficient Compression. *ACM Trans. Database Syst.*, 31(1):1–38, 2006.