Efficient and Scalable Schema Evolution with Column Oriented Databases

Presented by Bin He

Research Scientist
IBM Almaden Research Center
Column oriented databases

- Column oriented databases store a table according to columns instead of rows
- Compressed bitmap indexes are often used for each column

<table>
<thead>
<tr>
<th>Employee</th>
<th>Skill</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jones</td>
<td>Typing</td>
<td>425 Grant Ave</td>
</tr>
<tr>
<td>Jones</td>
<td>Shorthand</td>
<td>425 Grant Ave</td>
</tr>
<tr>
<td>Roberts</td>
<td>Light Cleaning</td>
<td>747 Industrial Way</td>
</tr>
<tr>
<td>Ellis</td>
<td>Alchemy</td>
<td>747 Industrial Way</td>
</tr>
<tr>
<td>Jones</td>
<td>Whittling</td>
<td>425 Grant Ave</td>
</tr>
<tr>
<td>Ellis</td>
<td>Juggling</td>
<td>747 Industrial Way</td>
</tr>
<tr>
<td>Harrison</td>
<td>Light Cleaning</td>
<td>425 Grant Ave</td>
</tr>
</tbody>
</table>

A logical view of a table

The column oriented storage of a table
Schema evolution

- Why schema evolution
  - data changes
  - query workload changes
  - application changes
  - initial schema is not well designed

- Two steps to complete a schema evolution in practice
  - Schema update
  - Data migration

- Schema evolution is frequent
  - Fast changing world → fast changing database
  - Annotations on unstructured data/semi-structured data

- Schema evolution is expensive
  - Data migration can be a time-consuming step, which may block users from making frequent schema adjustments to achieve optimal performance/usability
Current data migration on row oriented DBMSs: A query-level framework (e.g., Clio)

- Problems of the query-level approach
  - Unnecessary attribute accessing
  - Index rebuilding

![Diagram showing data migration process with query operations: Select and Insert, and schema revision.](image)
An example of the query-level migration for row oriented databases

- **The schema evolution action**
  - Split Employee and Address as a separate table T
  - Leave Employee and Skill as it is with a new table name S

- **The queries**
  - Q1: insert into T select distinct Employee, Address from R
  - Q2: insert into S select Employee, Skill from R

- **The inefficiency of such a query-level approach**
  - Skill is not affected, and thus should not be accessed. But both Q1 and Q2 access it
  - Indexed may have to be rebuilt on S and T, even S is the same as R on Employee and Skill
Are column oriented databases a better choice on schema evolution?

- Obvious advantages on schema evolution
  - Unnecessary attribute accessing can be avoided
  - Existing storage can be reused
  - Some index rebuilding is not necessary

- Existing works on column oriented databases focus on query processing, not on schema evolution
Query-level data migration for column oriented databases

<table>
<thead>
<tr>
<th>Employee</th>
<th>Skill</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jones</td>
<td>Typing</td>
<td>425 Grant Ave</td>
</tr>
<tr>
<td>Jones</td>
<td>Shorthand</td>
<td>425 Grant Ave</td>
</tr>
<tr>
<td>Roberts</td>
<td>Light Cleaning</td>
<td>747 Industrial Way</td>
</tr>
<tr>
<td>Ellis</td>
<td>Alchemy</td>
<td>747 Industrial Way</td>
</tr>
<tr>
<td>Jones</td>
<td>Whittling</td>
<td>425 Grant Ave</td>
</tr>
<tr>
<td>Ellis</td>
<td>Juggling</td>
<td>747 Industrial Way</td>
</tr>
<tr>
<td>Harrison</td>
<td>Light Cleaning</td>
<td>425 Grant Ave</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Employee</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jones</td>
<td>425 Grant Ave</td>
</tr>
<tr>
<td>Roberts</td>
<td>747 Industrial Way</td>
</tr>
<tr>
<td>Ellis</td>
<td>747 Industrial Way</td>
</tr>
<tr>
<td>Harrison</td>
<td>425 Grant Ave</td>
</tr>
</tbody>
</table>

- **The query-level migration**
  - Q1: insert into T select distinct Employee, Address from R
  - Q2: insert into S select Employee, Skill from R

- With column oriented databases, Q2 can be done without a real query. Q1 can avoid accessing Skill. No index rebuilding in necessary

- However, new problems arise…
The problems of query-level data migration on column oriented databases

- With the query-level approach, the query Q1 is processed as
  - First, the query “select distinct Employee, Address from R” is processed. The query result is materialized
    - Columns are aligned into tuples (rows)
    - Compressed values are organized into uncompressed ones
  - Second, the “insert into T” part is processed
    - Tuples are again broken into columns
    - Values are compressed again

- The column→tuple→column transition is unnecessary
- The compressed→decompressed→compressed transition is unnecessary

- Can we do better?
Our proposal: A data-level migration framework

- The data-level migration framework takes all the advantages of the column oriented databases
- Plus, it has the following advantages over the query-level framework
  - It directly generates the target data storage (compressed bitmaps) from the source data storage (compressed bitmaps)
  - No tuple materialization
  - No bitmap decompression
Data-level vs. Query-level

The Query-Level Framework

Query: Select

Query: Insert

The Data-Level Framework

A

Schema Revision

B

Data Migration

A

Schema Revision

B

Data Migration
### Schema modification operations

#### Easy operations
- CREATE TABLE
- DROP TABLE
- RENAME TABLE
- COPY TABLE
- COMBINE TABLES
- PARTITION TABLE
- RENAME COLUMN
- ADD COLUMN
- DROP COLUMN

#### Difficult ones
- DECOMPOSE TABLE
- MERGE TABLES

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

- **Step 1**: Find distinct bitmap vectors to a set of distinct employees
  - For each distinct employee, find the first location to contain it in R
    - {'Jones', 1}, {'Roberts', 3}, {'Ellis', 4}, {'Harrison', 7}
    - The position list is thus {1,3,4,7}
  - Employee is the primary key attribute, and we directly store the values

- **Step 2**: Generate new bitmap indexes from old ones using the filter vector generated in step 1
  - The vector of address “425 Grant Ave” in R is 1100101. Use {1,3,4,7} as the filter, the new vector in T is 1001
  - The vector of address “747 Industrial Way” in R is 0011010. Use {1,3,4,7} as the filter, the new vector in T is 0110

- **Step 2** can be repeated on multiple non-key attributes, one at a time
  - Very scalable in terms of number of attributes to be processed
Merge tables

Given a table \( T(A_1, \ldots, A_k, A_{k+1}, \ldots, A_p) \) and \( S(A_1, \ldots, A_k, A_{p+1}, \ldots, A_n) \), we want to get their merge result \( R(A_1, \ldots, A_k, A_{k+1}, \ldots, A_p, A_{p+1}, \ldots, A_n) \) (i.e., a join + insert into)

Three cases
- Trivial case: \( (A_1, \ldots, A_k) \) is primary key of both \( S \) and \( T \). No real data migration is necessary for the mergence
- Key-Foreign Key case: \( (A_1, \ldots, A_k) \) is primary key of only one table (say \( T \))
  - Column storage on \( S \) can be reused in \( R \). No real data migration is necessary on \( A_1, \ldots, A_k \) and \( A_{p+1}, \ldots, A_n \)
  - Our algorithm focuses on efficiently generation of bitmaps for \( A_{k+1}, \ldots A_p \) in \( R \)
- Non-reusable case: \( (A_1, \ldots, A_k) \) is not primary key of either \( S \) or \( T \)
  - Bitmaps of all attributes in \( R \) need to be regenerated
Employee and Skill in S can be reused in R

To generate bitmap vector of address “425 Grant Ave” in R
- the vector of “425 Grant Ave” in T is 1001
- using the position→value offset index of T’s employee, we know Jones and Harrison have this address
  - Jone’s vector in S is 1100100, Harrison’s is 0000001
  - “425 Grant Ave”’s vector in R is thus 1100100 OR 0000001 = 1100101

Each non-key attribute needs to access all the key attributes using the offset index. When there are many non-key attributes, offset index is access multiple times, which is not efficient.
An example of non-reusable mergence

<table>
<thead>
<tr>
<th>Employee</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jones</td>
<td>425 Grant Ave</td>
</tr>
<tr>
<td>Ellis</td>
<td>747 Industrial Way</td>
</tr>
<tr>
<td>Jones</td>
<td>747 Industrial Way</td>
</tr>
<tr>
<td>Jones</td>
<td>60 Aubin St</td>
</tr>
<tr>
<td>Ellis</td>
<td>501 Oakman Blvd</td>
</tr>
</tbody>
</table>

The difficulty

- How to compute the vector positions without doing a join of S* and T*?
  - Jone’s positions in R* is not only affected by Jone’s vectors in S* and T*, but also other employees
A re-organized table R

<table>
<thead>
<tr>
<th>Employee</th>
<th>Address</th>
<th>Skill</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jones</td>
<td>425 Grant Ave</td>
<td>Typing</td>
</tr>
<tr>
<td>Ellis</td>
<td>747 Industrial Way</td>
<td>Alchemy</td>
</tr>
<tr>
<td>Jones</td>
<td>747 Industrial Way</td>
<td>Typing</td>
</tr>
<tr>
<td>Jones</td>
<td>60 Aubin St</td>
<td>Typing</td>
</tr>
<tr>
<td>Ellis</td>
<td>501 Oakman Blvd</td>
<td>Alchemy</td>
</tr>
</tbody>
</table>

- **R* is re-organized in a way that**
  - first clustered by the key attributes (i.e., Employee)
  - then clustered by non-key attributes in T (i.e., Address)
  - finally clustered by non-key attributes in S (i.e., Skill)

- **The advantages of re-organization**
  - It is now much easier to calculate the positions of an attribute value in R*  
  - The positions of the values of generated attributes are predictable now
Experiments

- Comparison between
  - Data-level decomposition and mergence on column oriented databases
    - We implement our algorithms in C++
    - We use file system and thus do not take the advantages of buffer and disk management in DBMSs
  - Query-level decomposition and mergence on row oriented databases
    - DB2 is used
    - For fair comparison, direct import is used with minimal logging
    - Two situations are experimented, with and without indexes
  - We are doing other experiments as well
    - Comparing to query-level framework on column databases
    - Comparing to baseline algorithms, not full-fledged systems

- Experimental Data
  - Synthetic data
    - Parameters: number of tuples, number of distinct values in a column, number of affected columns in a decomposition/mergence, number of unaffected columns in a decomposition/mergence, and the length of data values
  - Real Data: Patent DB

- Result highlights
  - Data-level schema evolution on column oriented databases is always faster, and can achieve efficiency improvement of 1-2 orders of magnitude in most cases
  - Data-level schema evolution on column oriented databases is much more scalable in terms of number of tuples, number of columns, number of distinct values and number of columns, the length of data values
Performance of Decomposition on Synthetic Data—Number of tuples

- **R→S and T. Settings**
  - R is \((A_1, A_2, A_3)\): \(A_3\) functionally depends on \(A_2\)
    \[ R(A_1, A_2, A_3) \rightarrow S(A_1, A_2), T(A_2, A_3) \]
  - The average length of the values in R is 5 bytes
  - Number of tuples in R varies from 2M to 10M. Let the number of tuples be \(NT\)
  - \(A_2\) has \(NT/10\) distinct values and \(A_3\) has \(NT/100\) distinct values

![Graph showing performance of decomposition on synthetic data with data-level, DB2, and DB2+indexes]
Conclusion

- Observations from analysis and experiments
  - Column oriented database is a better choice to support efficient data migration in scheme evolution
  - Data-level migration is better than query-level migration

- The implications of our work
  - If schema evolution becomes quick and easy, database designer does not have to be so cautious in the design phase any more
  - Frequent schema optimization according to query workload becomes possible
  - There are other potential applications of the data-level framework besides schema evolution, e.g., generating materialized views
Thanks!