Efficient Aggregation Algorithms for Compressed Data Warehouses

Yingshu Li
Dept. of Computer Science and Engineering
University of Minnesota

Outline
- Motivation
- Preliminaries
- Compression of DW
- Aggregation Algorithms
- Selection of Algorithms
- Experiment Results
- Conclusion & Future Research

Motivation

- More and More Massive Datasets Are Coming
- VLDB2002 shown
- Many large enterprises have datasets with size of \(10^{15}\) bytes, and is growing with rate of 20%.
- Data warehouse is much more larger, and is growing with rate of 40%.
- Walmart is building data warehouse of size \(P\) byte
- Satellites of NASA return more than \(10^{15}\) byte data every year.
- One year's data generated by particle collision experiments in LBL is more than \(3 \times 10^{14}\) bytes.

Challenges of Massive Data

- How to store the massive data?
- How to process queries on massive data?

Experiments

- E1. Execute Cube algorithm Fastcube
  - On 8-dimensional Dataset with 10^6 Tuples, Executing time is 1.67 hours.
  - On 8-dimensional Dataset with 10^8 Tuples, Executing time is more than 160 hours.
- E2. Using PC method to process range queries on 5-dimensional datasets with different size.

Traditional techniques can not efficiently support massive data warehouses!!
Methods to the challenge problems
- Use Parallel computing technique
  - Parallel data management systems
- Use Compression technique
  - Compression data management systems (CDMS)
- Use on line Tertiary storage technique
  - Tertiary storage based data management systems

Techniques of CDMS
- Database compression methods
  - Suitable for random access mode
- Data operation algorithms on Compressed databases
  - Without decompression
- Query optimization and procession
  - Without decompression

Focus of this Paper
- Data operation algorithms on Compressed databases
  - Without decompression
- Four aggregation algorithms on Compressed data warehouses
  - G-Aggregation Algorithm
  - Algorithm M-Aggregation
  - Algorithm Prefix-Aggregation
  - Algorithm Infix-Aggregation
- Algorithm Selection Procedure

Related Work
- J. Li published “Transposition Algorithms on Very Large Compressed Databases” in VLDB’86.
- Y. Zhao published “An Array-Based Algorithm for Simultaneous Multidimensional Aggregations” in SIGMOD’97

Data Warehouses
- Relational DB based data warehouses (RDW)
  - Data is stored in relational databases
  - OLAP is on relational databases
- Multidimensional data warehouse (MDW)
  - A MDW is a set of multidimensional datasets
  - A multidimensional dataset consists of dimensions and measures, represented by R(Dp, ..., Dq; M1, ..., Mk), where Di is dimensions and Mi is measures
  - Example
    Sales(PN, Date, Location; amount, money)
Fundamental difference of RDW and MDW

- **RDW** uses relational tables as their data structure.
  - A "cell" in a multidimensional space is a tuple with some attributes identifying the location of the cell in the space and other attributes containing the values of measures of the cell.
- **MDW** stores datasets in multidimensional structures
  - Only store the values of measures in a multidimensional space. The position of the measure values in the space can be calculated by the dimension values.
  - For example, multidimensional array can be used

Reason 1

Multidimensional space created by cross product of dimensions is naturally sparse

- For example, in international trade dataset
  \[ \text{Trade(exporting country, import country, materials, year and month; amount)} \]
  only a small number of materials are exported from any given country to other countries.

Reason 2

MDW needs to compress the descriptors of the multidimensional space

- Using relational database, the dimensions organized in table will create a repetition of the values of each dimension.
- In the extreme, but often realistic case, the full cross product is stored, the number of times that each value of a given dimension repeats is equal to the product of the cardinalities of the remaining dimensions.

Other Reasons

- Often the data values are skewed in some datasets, where there are a few large values and many small values.
- In some datasets, data values are large but close to each other.
- Sometimes certain values tend to appear repeatedly.

Aggregation

- Definition of aggregation \( \text{Agg}(R, S, F) \)
  - collapse away dimensions \( R-S \) to obtain a more concise dataset, or
  - classify items into groups and determine one value per group.

Compression of DW
Steps of Compressing MDW

- Given MDW $R(D_1, ..., D_n; M_1, ..., M_m)$
  - **Step 1**
    - $R$ is stored in multidimensional array to remove the need for storing the dimension values.
  - **Step 2**
    - The array is transformed into a linearized array by an array linearization function
  - **Step 3**
    - The linearized array is compressed by mapping-complete compression method.

Let $R=\text{Sales}(PN, Location; amount, money)$

<table>
<thead>
<tr>
<th>Loca.</th>
<th>PN</th>
<th>Amount</th>
<th>money</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>p1</td>
<td>500</td>
<td>5000</td>
</tr>
<tr>
<td>L1</td>
<td>p2</td>
<td>200</td>
<td>800</td>
</tr>
<tr>
<td>L2</td>
<td>p1</td>
<td>300</td>
<td>3000</td>
</tr>
<tr>
<td>L2</td>
<td>p2</td>
<td>100</td>
<td>500</td>
</tr>
<tr>
<td>L3</td>
<td>p2</td>
<td>600</td>
<td>6000</td>
</tr>
<tr>
<td>L4</td>
<td>p3</td>
<td>400</td>
<td>8000</td>
</tr>
<tr>
<td>L5</td>
<td>p1</td>
<td>200</td>
<td>4000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Amount</th>
<th>Money</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 200</td>
<td>L1 5000</td>
</tr>
<tr>
<td>L2 300</td>
<td>L2 800</td>
</tr>
<tr>
<td>L3 600</td>
<td>L3 6000</td>
</tr>
<tr>
<td>L4 400</td>
<td>L4 8000</td>
</tr>
<tr>
<td>L5 200</td>
<td>L5 4000</td>
</tr>
</tbody>
</table>

Each measure of $R$ is stored in a separate array
- Each dimension of $R$ is used to form a dimension of these $n$-dimensional arrays.
- The dimension values of $R$ are not stored. They are the indices of the array to determine the position of the measure values in the array.

Each of the $n$-dimensional arrays is mapped into a *linearized array* by linearization function.
- Assume that the values of the $i^{th}$ dimension of $R$ is encoded into $\{0, 1, ..., d_i-1\}$
- A *dimension order* of $R$, denoted by $D_1, D_2, ..., D_n$, is an order in which measure values of $R$ are stored in linearized array by the linearization function with $D_i$ as the $i^{th}$ dimension.
- The linearization function with order $D_1, D_2, ..., D_n$ is $L(x_1, x_2, ..., x_n) = x_1d_2d_3...d_n + x_2d_3...d_n + ... + x_{n-1}d_n + x_n$.
- Different dimension orders leads to different orders of the measure values in linearized array.
- The reverse array linearization function

Dimension order is $(\text{Location, PN})$

<table>
<thead>
<tr>
<th>Amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 500</td>
</tr>
<tr>
<td>L2 300</td>
</tr>
<tr>
<td>L3 600</td>
</tr>
<tr>
<td>L4 400</td>
</tr>
<tr>
<td>L5 200</td>
</tr>
</tbody>
</table>

Dimension order is $(\text{Location, PN})$

<table>
<thead>
<tr>
<th>Money</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 5000</td>
</tr>
<tr>
<td>L2 5000</td>
</tr>
<tr>
<td>L3 6000</td>
</tr>
<tr>
<td>L4 8000</td>
</tr>
<tr>
<td>L5 4000</td>
</tr>
</tbody>
</table>
Step 3

- Header compression method
  - Header: A vector of counts
  - Compressed file: with the missing data removed
  - It is mapping complete
    - forward mapping
    - backward mapping
- Header compression method is used to compress the linearized array

G-Aggregation

- Input: Compressed dataset $R$
- Output: $\text{Agg}(R, S, F)$
- “General” algorithm in the sense that it can be used in all situations
- Performs aggregation in two phases.
  - Phase one: transposition phase
    - Transposes the dimension order of the input into a favorable order so that the aggregation can be easily computed.
  - Phase two: aggregation phase
    - Computes the aggregation by one scan of the transposed input.

Idea: Compute $\text{Agg}(R, \{B, C\}, \text{Sum})$

```
<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>
```

Transposition

```
<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>
```

Aggregation

```
\text{R} = (R, C, A, D, M)
```

Implementation: Compute $\text{Agg}(R, \{\text{PN}\}, \text{Sum})$

```
<table>
<thead>
<tr>
<th>Loca. PN</th>
<th>Amount</th>
<th>Trans.</th>
<th>PN</th>
<th>Loca. Amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>p1</td>
<td>$\text{Agg}(\text{R}, {\text{PN}}, \text{Sum})$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L2</td>
<td>p2</td>
<td>500</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L3</td>
<td>p3</td>
<td>600</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L4</td>
<td>p4</td>
<td>800</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L5</td>
<td>p5</td>
<td>100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L6</td>
<td>p6</td>
<td>200</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>PN</th>
<th>Amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>1000</td>
</tr>
<tr>
<td>p2</td>
<td>900</td>
</tr>
<tr>
<td>p3</td>
<td>600</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Comp. File</th>
<th>Amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>500</td>
</tr>
<tr>
<td>L2</td>
<td>300</td>
</tr>
<tr>
<td>L3</td>
<td>200</td>
</tr>
<tr>
<td>L4</td>
<td>100</td>
</tr>
<tr>
<td>L5</td>
<td>400</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Comp. File</th>
<th>Amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>300</td>
</tr>
<tr>
<td>L2</td>
<td>200</td>
</tr>
<tr>
<td>L3</td>
<td>100</td>
</tr>
<tr>
<td>L4</td>
<td>400</td>
</tr>
<tr>
<td>L5</td>
<td>400</td>
</tr>
</tbody>
</table>
```
**Complexity of G-Aggregation**

- The average I/O cost is $O(N + N_r \log N)$
- The average CPU cost is $O(N(n + \log B) + N_r \log N)$

**Algorithm**

- It reads blocks of the compressed $R$ one by one.
- For each data item $v$ of $R$, the following is done:
  1. Backward mapping is performed to obtain $v$'s logical position;
  2. Dimension values of $v$, $(x_1, ..., x_d)$, are recovered by reverse L. F from logical position of $v$;
  3. The values $(a_1, ..., a_k)$ of the group-by dimensions are selected from $(x_1, ..., x_d)$;
  4. If there is a $w$ that is identified by $(a_1, ..., a_k)$ in the output buffer, aggregate $v$ to $w$, otherwise insert $v$ with $(a_1, ..., a_k)$ as a tag into the output buffer using hash method.
- Finally, the algorithm builds the new header file and writes the output buffer to the result file discarding the tags.

**Complexity of M-Aggregation**

- The average I/O cost is $O(N + N_r \log N)$
- The average CPU cost is $O(N(n+h) + N_r \log N)$

**Input:** Compressed dataset $R$

**Output:** $Agg(R, S, F)$

**Advantages**

- It is superior to G-Aggregation in case of the output fitting into memory.
- M-Aggregation computes aggregation by only one scan of the compressed dataset $R$.

**Prefix-Aggregation**

**Input:** Compressed dataset $R$

**Output:** $Agg(R, S, F)$

**Advantages**

- It takes advantage of the situation where $S$ contains a prefix of the dimension order of $R$.
- It performs aggregation in main memory by one scan of the compressed $R$.
- It requires a main memory buffer large enough to hold each portion of the resulting compressed array for each “point” in the subspace composed by the prefix.
A notation
- \( R(D_1, ..., D_k, a_{k+1}, ..., a_{k+p}, D_{k+p+1}, ..., D_n; M) \)
  represents a subset of \( R(D_1, ..., D_n; M) \) whose
  dimension values on \( (D_{k+1}, ..., D_{k+p}) \) are
  \( (a_{k+1}, ..., a_{k+p}) \).
- For example

<table>
<thead>
<tr>
<th>R(A, B, C, D; M)</th>
<th>R(A, I, I, D; M)</th>
<th>R(I, B, C, I; M)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A B C D M</td>
<td>A B C D M</td>
<td>A B C D M</td>
</tr>
<tr>
<td>1 1 1 1 4</td>
<td>1 1 1 1 4</td>
<td>1 1 1 1 4</td>
</tr>
<tr>
<td>1 1 1 2 5</td>
<td>1 1 1 2 5</td>
<td>1 2 1 1 3</td>
</tr>
<tr>
<td>1 2 1 1 3</td>
<td>1 2 1 1 3</td>
<td>1 2 1 1 3</td>
</tr>
<tr>
<td>2 1 1 1 1</td>
<td>2 2 1 2 4</td>
<td>2 2 1 2 4</td>
</tr>
<tr>
<td>2 2 1 2 2</td>
<td>2 2 1 2 2</td>
<td>2 2 1 2 2</td>
</tr>
</tbody>
</table>

Idea of Prefix-Aggregation
- \( \text{Agg}(R(A, B, C, D; M), (A, B, D), \text{Sum}) \)
- For example

<table>
<thead>
<tr>
<th>R(A, B, C, D; M)</th>
<th>R(A, B, D; M)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A B C D M</td>
<td>A B D M</td>
</tr>
<tr>
<td>1 1 1 1 4</td>
<td>1 1 1 4</td>
</tr>
<tr>
<td>1 1 1 2 5</td>
<td>1 1 2 5</td>
</tr>
<tr>
<td>1 2 1 1 3</td>
<td>1 2 1 3</td>
</tr>
<tr>
<td>2 1 1 1 1</td>
<td>2 1 1 1</td>
</tr>
<tr>
<td>2 2 1 2 4</td>
<td>2 2 1 2</td>
</tr>
<tr>
<td>2 2 1 2 2</td>
<td>2 2 1 2</td>
</tr>
</tbody>
</table>

Complexity of Prefix-Aggregation
- The average I/O cost is \( O(N+N_r) \).
- The average CPU cost is \( O(nN+N_r) \).

Infix-Aggregation
- Input: Compressed dataset \( R \)
- Output: \( \text{Agg}(R, S, F) \)
- Advantages
  - It takes advantage of the situation where \( S \) is an infix of the dimension order of \( R \).
  - It has higher performance than G-Aggregation.
  - It works with any size memory.

Idea of Infix-Aggregation
- \( \text{Agg}(R(A, B, C, D, E; M), (C, D), \text{Sum}) \)
- Result is \( R(C, D; M) \)
- \( (C, D) \) is an infix of \( (A, B, C, D, E) \)
- **Complexity of Infix-Aggregation**
  - The average I/O cost is \( O(n \log n + N, \log N) \)
  - The average CPU cost is \( O(nN + N \log N) \)

- **Selection of Algorithms**

- **Comparison of Algorithms**
  - Observation 1.
    - \( G\text{-Aggregation} \geq \text{Prefix-Aggregation} \),
    - \( G\text{-Aggregation} \geq \text{M-Aggregation} \),
    - \( M\text{-Aggregation} \geq \text{Prefix-Aggregation} \),
    - \( \text{Infix-Aggregation} \geq \text{Prefix-Aggregation} \).
  - Observation 2.
    - If \( \text{Infix-Aggregation} \geq \text{M-Aggregation} \),
      then \( \text{Infix-Aggregation} \geq \text{M-Aggregation} \).

- **Notations for Selection** (\( \text{Agg}(R, S, F) \))
  - \( \alpha = \) “\( S \) contains a prefix of the dimension order of \( R \)”
  - \( \beta = \) “Size of agg. result can fit into available memory”
  - \( \gamma = \) “\( S \) contains a prefix of the dimension order of \( R \)”
  - \( \eta = \) “available memory satisfies the requirement of \( \text{Prefix-Aggregation} \)”
  - \( A = \) “\( \text{Infix-Aggregation} \geq \text{Cost} \), \( \text{G-Aggregation} \)”
  - \( B = \) “Condition of Observation 2”
  - \( C = \) “\( \text{Infix-Aggregation} \geq \text{Cost} \), \( \text{M-Aggregation} \)”

- **Selection Algorithm**
  ```
  If \( \gamma \land \eta \) Then select \( \text{Prefix-Aggregation} \);
  Else If \( \beta \land ((B \lor C) \lor \lnot \alpha) \) Then select \( \text{M-Aggregation} \);
  Else If \( \alpha \land \lnot A \) Then select \( \text{Infix-Aggregation} \);
  Else select \( \text{G-Aggregation} \).
  EndIf
  ```
Experiment Results

Performance vs Number of Valid Data items

- Dataset scheme
  - 15 dimensions and one measure.
  - Data types of dimensions are 4-byte integer.
  - Data type of measure is 4-byte float number.
- Four datasets are randomly generated
  - Valid data items: 1,000,000, 5,000,000, 10,000,000, and 20,000,000.
  - Header size of each dataset was 50% of the dataset size.
  - Agg. result size of each dataset was 20% of the dataset size.

Experimental Result 1

- 

Experimental Result 2

- 

Experimental Result 3

- 

Experimental Result 4

-
Experimental Result 5

Performance vs Compression Ratio

- Conducted experiments while dataset size is fixed and dimension number varies
  - Dimension number of datasets was varied from 2 to 20
  - Number of data entries was fixed at 10,000,000, each with 64 bytes
  - Agg. result size was kept at 2,000,000 data entries
- Conducted experiments while the header size varies. The details of the experiments see [24].

Experimental Result 6

Experimental Result 1

Experimental Result 2

Experimental Result 3
Experimental Result 4

Figure. 17

<table>
<thead>
<tr>
<th>Dimension Number</th>
<th>Execution Time (Seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>M</td>
</tr>
<tr>
<td>5</td>
<td>Infix</td>
</tr>
</tbody>
</table>

Experimental Result 5

Figure. 18

<table>
<thead>
<tr>
<th>Dimension Number</th>
<th>Execution Time (Seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>M</td>
</tr>
<tr>
<td>5</td>
<td>Prefix</td>
</tr>
</tbody>
</table>

Experimental Result 6

Figure. 19

<table>
<thead>
<tr>
<th>Dimension Number</th>
<th>Execution Time (Seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>G</td>
</tr>
<tr>
<td>5</td>
<td>Sort</td>
</tr>
<tr>
<td>10</td>
<td>Hash</td>
</tr>
</tbody>
</table>

Performance vs Memory Size

- Dataset scheme
  - 15 dimensions and 1 measure
  - Has 10,000,000 data entries of 64 bytes

- Experiments
  - In the first three sets of the experiments
    - Agg. result size is fixed at 2,000,000 data entries
    - Memory size varies from 10k bytes to 15M bytes
  - In the fourth and fifth sets of experiments
    - Agg. result size is fixed at 200,000 data entries
    - Memory size varies from 12.5M to 35M bytes

Experimental Result 1

Figure. 20

<table>
<thead>
<tr>
<th>Memory Size (Bytes)</th>
<th>Execution Time (Seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10k</td>
<td>G</td>
</tr>
<tr>
<td>50k</td>
<td>Prefix</td>
</tr>
<tr>
<td>100k</td>
<td>Sort</td>
</tr>
<tr>
<td>200k</td>
<td>Hash</td>
</tr>
</tbody>
</table>

Experimental Result 2

Figure. 21

<table>
<thead>
<tr>
<th>Memory Size (Bytes)</th>
<th>Execution Time (Seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10k</td>
<td>G</td>
</tr>
<tr>
<td>50k</td>
<td>Infix</td>
</tr>
<tr>
<td>100k</td>
<td>Sort</td>
</tr>
<tr>
<td>200k</td>
<td>Hash</td>
</tr>
</tbody>
</table>
**Performance vs Dimension Size**

- **Experimental Result 1**
  - Dimension size has great effect on Infix, and a little effect on other algorithms.
  - **Dataset scheme:**
    - 15 dimensions and 1 measure
    - Dataset size is fixed at 10,000,000 data entries, each with 64 bytes
  - **Experiments**
    - First 2 dimensions have fixed sizes 2 and 5
    - Agg. dimension set of each agg. begins from the fourth dimension.
    - Size of the third dimension varies from 1 to 5,000
    - Thus, the number of the runs that need to be merged by Infix varies from 10 to 50,000.
Experimental Result 2

Figure 27

Infix
G
Sort
Hash
M

Execution Time (Seconds)

Number of Runs

10 50 100 150 200 250 300 350 400 450 500

0 100 200 300 400 500

Conclusion and Future Research

Conclusion

• Four agg. algorithms was proposed
• The algorithms operate directly on compressed datasets without decompressing first
• A decision procedure is also given to select the most efficient algorithm based on aggregation request, available memory, as well as the dataset parameters for a given aggregation request.

The analysis and experimental results show that the proposed algorithms in this paper have better performance than the previous aggregation algorithms.

In conclusion, direct manipulation of compressed data is an important tool for managing very large datasets.

Future Research

• We are currently working on developing new compression methods for DWs and DBs
• We are also working on other operators on compressed DWs and DBs
• We will work on data operation algorithms applicable to other kinds of compression methods other than mapping-complete compression methods on DWs and DBs
• We will also work on query optimization and processing method for compressed DWs and DBs.

Q&A