Data Warehouse Optimization Through Differential Patterns and Schema Change

Jens Frøkjær, Palle B. Hansen, Ivan V. S. Larsen and Tom Oddershede
Department of Computer Science, Aalborg University
{fr0,palle,ivanvsl,tom}@cs.aau.dk

May 31, 2005

Abstract

Data warehousing is a common tool for decision support, which in order to achieve high precision must hold masses of historical data. In this paper, we present a method for reducing the size of the data warehouse. The method describes how the attributes should be split into three different groups and how the schema of the data warehouse should be changed. How the attributes are represented in the warehouse is determined by the attribute type. The size reduction of the warehouse is done using differential compression and re-use of patterns. We also show how the method is carried out on real GPS data collected by busses in public transportation. This will be done on a Microsoft SQL-Server 2000. Finally a set of performance tests are carried out and the results are compared. The main conclusions of this article are that space can almost always be saved and sometimes better query performance can be achieved.

1 Introduction

A data warehouse is a collection of historical data, which can be used for decision support. The data warehouse is a denormalized database where the tables typically are stored in a star schema [Kim02]. The main purpose of the star schema is to ease the understanding of how data is related. Using the star schema reduces the amount of joins needed to perform a query, which leads to simpler queries. For the sake of simplicity in the queries the normalized schema of the data warehouse is sacrificed. Over time the data in the data warehouse will increase to take up huge amounts of disk space. To prevent this, data compression can be applied, which is the topic of this paper.

Compression comes in two forms, *reversible* and *non-reversible*. Reversible compression means that the original data can be fully reconstructed after it has been compressed whereas non-reversible data cannot [RH93]. Non-reversible compression is not always so desirable because it might reduce the number of useful queries that can be performed on the data.

When data in a database is to be saved on a disk, compression can be applied in two different places. Either the compression is handled directly by the database management systems (DBMS) or it is done at query level. When doing compression at query level, it is *DBMS independent*.

The aim of this article is to describe a method of reversible and semantic independent compression of data in a data warehouse while preserving the query processing speed from the uncompressed warehouse. The compression/decompression is done at query level to ensure that the method can be used across different DBMSs. Preserving the query speed can possibly be

achieved by reducing the amount of disk I/O through compression. Use of CPU time instead of disk space can be even more important in the future if the current improvements in CPU and disk speeds continue. CPU speed is improving faster than disk speed. Therefore it is useful to minimize disk I/O, even though the cost is more CPU computations [PRWS00, Ses95].

This paper is structured as follows. In Section 2 related work within the areas of compression and databases is presented and compared to our work. An example which will be used throughout this paper is presented in Section 3. Section 4 introduces the three types of attributes. Section 5 establishes a general model for reducing the size of a data warehouse using differential compression and in which situations the model can be applied. In Section 6 the actual compression method is presented and it is discussed how it behaves on the three attribute types. The handling of decompression is described in Section 7. General rewrite rules for querying the compressed warehouse is formalized in Section 8. In Section 9 the model is applied to data collected by GPS devices located on busses. In Section 10 a number of performance tests are described and the results are presented. Finally Section 11 concludes on the method described in this paper.

2 Related Work

Databases are usually not compressed because traditional data compression techniques need large data chunks to be efficient [RH93]. Because of this, random access to only small parts of the data is not possible to execute within reasonable time[Kor01]. This is because large amounts of data has to be decompressed although only a small part of the data is needed to perform the query.

Another approach to compression in databases is to compress the result of a query [Che02]. This is useful in situations where the bandwidth is limited or when the client receiving the query result has limited memory. This approach does not directly improve on the execution time but reduces the network overhead.

In [GRDT99] a strategy to make data warehouses smaller and faster is proposed. This is done by rebuilding the index used in the database. This requires changes in the DBMS, and therefore moving the method from one DBMS to another is a complicated process.

[Ses95] presents a number of different compression methods and experiments. It claims that compression in databases always should be used, not only to reduce the amount of disk space, but also to achieve better performance, due to of the reduced number disk I/Os.

[Riz03] proposes a way of optimizing the I/O in a data warehouse through planning. It is an investigation of how to distribute tables and indexes over several physical disks. Different disk layouts are shown and compared to each other.

In contrast, our work is different in the way that it is independent of the DBMS on which the method is carried out. Our method can be carried out on top of any DBMS as it only changes the schema of the warehouse. Therefore the method can be used on any DBMS and no changes to this are required.

3 Example

Imagine a simple warehouse of real-time bus data collected by busses at numerous lines and journeys. Data is obtained at specific points on the line by GPS devices and stored in the warehouse[Fjä03]. Each tuple in the **Busdata** table corresponds to exactly one received GPS record from a specific bus at a specific line at a specific journey. In order to determine the delay relative to the bus schedule the **Busdata** table holds both the actual arrival time and the target arrival time.

The warehouse has a number of dimension tables that contain additional information related to the fact table (**Busdata**). The attributes with the suffix *DimPK* are foreign keys to related dimension tables and the attributes with the suffix *DD* are degenerated dimensions. The schema of the warehouse can be seen in Figure 1.

The dimension tables are filled beforehand with relatively static data, for example the **LineDim** dimension contains information about all bus lines, in this case it contains the name of the line.

Such a warehouse can for example be used for examining how many per cent of the points where the delay

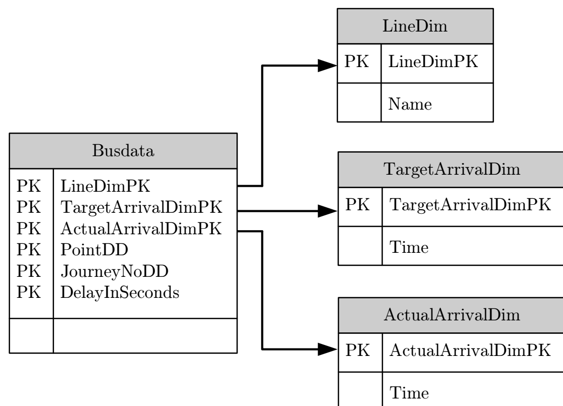


Figure 1: The standard warehouse schema

is larger than three minutes, to determine whether or not the busses are mainly on schedule or whether the schedule should be changed. A query doing this selection is shown in Listing 1.

```
1 SELECT
2    (SELECT COUNT(*)
3      FROM Busdata
4      WHERE DelayInSeconds > 180
5   /
6    SELECT COUNT(*)
7      FROM Busdata)*100
8 AS Delay
```

Listing 1: Selecting percentage of points that are delayed more than 3 minutes

Lines 2-4 select the number of tuples in the fact table where the delay is larger than 180 seconds. Lines 6-7 selects the actual number of tuples in the warehouse. Finally the results from the two sub-selects are divided in order to get the percentage of delays.

The example in Listing 1 will be used throughout this article.

4 Attribute Types

In order to use the compression method described in this article one must first of all understand these three new attribute types for the fact table. We will use the simple warehouse from Section 3 as an example.

4.1 Structural

A structural attribute in the **Busdata** table could for example be the `LimeDimPK` attribute that binds information together in the sense that tuples with the same unique `LineDimPK` are related. Another example could be the `JourneyNoDD` attribute. $V(\mathcal{S}, r)$ indicates the number of distinct values of $\mathcal{S}$ in the relation r. One could say that $V(\mathcal{S}, r)$ [Sun01], where $\mathcal{S}$ is the set of structural attributes in this case {`LineDimPK`,`JourneyNoDD`} and r is the fact table, in this case **Busdata**, should return a low number, at least compared to other combinations of attributes.

These attributes would typically be queried with both equality and inequality in the `WHERE` clause and can be used in the `GROUP BY` clause.

4.2 Accumulative

Accumulative attributes are attributes that should not be used in the `WHERE` clause or at least only where the operators $\geq$ or $>$ are used. Accumulative attributes could typically be dates where a query could be: "*in the last month what is the average delay*". In the **Busdata** table accumulative attributes could be `TargetArrivalDimPK` and `ActualArrivalDimPK`. As the name of the attribute type implies these are well suited for attributes that are mostly used for accumulation like the one in Listing 2.

```
1 SELECT AVG(TargetArrivalDimPK-
      ActualArrivalDimPK) FROM ...
```

Listing 2: A simple aggregating query

4.3 Critical

These attributes can be seen as specializations of the accumulative attributes, but when using these it is just as fast to evaluate the operators $\leq$ and $<$ as it is to evaluate $\geq$ and $>$ in the `WHERE` clause. This feature comes at the price of a lower compression rate. When tuples share the same values in the structural attributes then the variation of the value within the critical attribute should be low compared to the full domain of the critical attribute. This type of attribute is good for range-queries but not equality. `DelayInSeconds` is an example of such an attribute.

4.4 Summary

In Figure 2 an overview of the properties of the three attribute types can be seen. For operators it shows how the attribute type acts when placed on the left hand side. Usage shows if the attribute type is recommended for use in the `WHERE` clause. $\mathcal{S}$,$\mathcal{A}$ and $\mathcal{C}$ represents the structural, accumulative and critical attribute sets, respectively. GB shows if the attribute type is recommended for use in the `GROUP BY` clause. Operators which are marked with ✓supports the features of the attribute types described earlier. The ÷ indicates that performance probably will degrade severely. Performance will also be influenced badly on the attribute types which are not marked, but not as severe.

Type	Usage	$V(Type, r)$	$<$	$=$	$>$	GB
$\mathcal{S}$	In `WHERE`	$V(\mathcal{S}, r)$=Low	✓	✓	✓	✓
$\mathcal{A}$	Not `WHERE`	$V(\mathcal{A} \cup \mathcal{C}, r)$=High		÷	✓	÷
$\mathcal{C}$	In `WHERE`	$V(\mathcal{A} \cup \mathcal{C}, r)$=High	✓		✓	÷

Figure 2: Summary of attribute types

In the rest of this article we will assume that the attributes have the types as specified in Figure 3.

Attribute	**Type**
`LineDimPK`	Structural
`TargetArrivalDimPK`	Accumulative
`ActualArricalDimPK`	Accumulative
`PointDD`	Accumulative
`JourneyNoDD`	Structural
`DelayInSeconds`	Critical

Figure 3: Types of attributes

5 The Model

Consider that the simple warehouse in Section 3 should be used for historical analysis, and therefore tuples are generally never deleted. If we assume that we have 20 bus lines, and that we get GPS information once a minute per bus, and there are three busses per line; this comes to over 30 million tuples a year. All this data uses much disk space and accumulated over years this data set could become very large.

We need a compact representation of the data without losing performance on a wide range of query types. In the rest of this article we will apply a compression technique on this simple warehouse. We will now present the model behind the technique.

This model is based on *differential compression* which according to [RH93] a compression of more than 90% can be achieved when data vary slowly. This is because for most elements only the relationship between them needs to be stored. If data vary fast there is nothing gained by only saving the relationship, because storing the actual value would not take up more space.

Original data:	500	510	520	405	525
Differential pattern:	500	10	10	-115	120

Figure 4: Differential compression performed on a list of integers with a differential pattern as result

From now on a differential pattern is the result of a differential compression performed on a list of integers. See Figure 4.

The compression in the model is achieved through the reuse of differential patterns. The reuse is done by using two arrays. A list of integers is split into a number of small sublists of the size n. On each of these sublists a differential compression is performed and the result is a differential pattern which is stored in the second array. Note that the first integer in the differential pattern is stored in the first array and rest of the sublist is stored in the second array. This can be seen by comparing Figure 4 to the first entry in Array A in Figure 5. This is done to promote the reuse of the differential patterns. If the first integer was stored in the second array it would not be differential compression but substitution.

The differential pattern is stored in a second array and a reference is made to this element. If the differential pattern already exists in the second array a reference is made to the existing pattern. Therefore each pattern is only stored once, see Figure 5. For example is the pattern with the `DeltaId` 1 referred to by both elements one and four in Array A.

If several lists are required to be compressed, the same differential pattern array will be used, and one

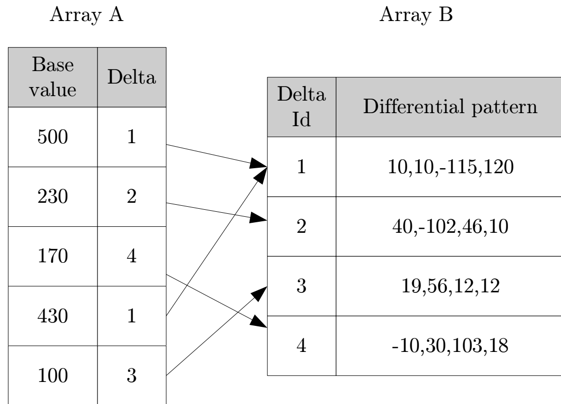


Figure 5: The array structure of the data after compression is performed

array is created for each list. When a pattern can be referenced several times, space is saved.

5.1 Prerequisites

We have decided to compress tables containing integer attributes which does not allow *null*s because they usually populate the major part of a data warehouse. The main table in the data warehouse is the fact table, which mostly contains references to other tables. These references are stored as integers. Therefore our compression method will be a refined version of the differential compression. The method can be used on all tables which only contain integers.

5.2 Calculating Delta-Patterns

In order to apply this model to a data warehouse a table to hold the delta patterns must be created. When the number of elements in the delta patterns is n then the format of this table is:

Delta(<u>`DeltaId`</u>, `D1`, `D2`, `D3`, ..., `D`n)

Only every n'th row of the original fact table is stored in the new fact table. However the new fact table must be reorganized in such a way that it corresponds to the new model. As an example we will compress the attribute `DelayInSeconds` in the data warehouse described in Section 3. First of all the original attribute is deleted and two new attributes are added to the fact table. The first is the `DelayInSecondsDelta` attribute which is a foreign key to a tuple in the **Delta** table. The second attribute is the `DelayInSecondsMax` which, as opposed to the example in Section 5, refers to the largest integer in the set of integers that is compressed. The original data from Figure 4 will be shown with delta pattern compression in Figure 6.

Original data:	500	510	520	405	525
Delta pattern:	-25	-15	-5	-120	0

Figure 6: Delta pattern compression performed on a list of integers, with a delta pattern as result where the max value is 525

Now we need to find the different delta patterns for the `Delta` table. This is done by selecting n rows from the `DelayInSeconds` attribute in the **Busdata** table. These rows all share the same values in the structural attributes, and they are put in the set m. From this set of integers $max(m_1, m_2, ..., m_n)$ is calculated and `DelayInSecondsMax` is set to this value which we will refer to as col_{max}. Now the delta pattern must be found which is done by adding, if it does not exist, the tuple m' which is $(m_1 - col_{max}, m_2 - col_{max}, ..., m_n - col_{max})$ to the **Delta** table. The attribute `DelayInSecondsDelta` is set to the value of `DeltaId` for m'. If a pattern cannot be filled fully then the rest is filled with *null*s.

When a new chunk of data is to be inserted into the compressed data warehouse, a new standard uncompressed fact table containing only the new data is created. From this the compression can be applied as described above and the compressed data is appended to an already compressed fact table.

6 Compression

This section will explain how the compression is applied on the three different attribute types described in Section 4. The table **Busdata** from the simple warehouse described in Section 3 will be used as an exam-

ple. The name of this new compressed table will be **CBusdata**.

6.1 Deciding n

The first thing to do is to find the size of the delta pattern. There is no easy way to find the perfect n value, as this is dependent on the data, but it is not an impossible task to try different values to see where the best compression is achieved. A rule of thumb for the size of n is to set it to about $\frac{n_r}{V(\mathcal{S},r)}$ where n_r is the number of tuples in the standard warehouse. This number is the average length of the delta pattern.

6.2 Structural

The relationship between the **Delta** table and the compressed fact table is a one-to-many. This means that several tuples in the compressed fact table may refer to the same tuple in the **Delta** table.

One tuple in the compressed fact table will actually, through the use of the **Delta** table, represent a number of tuples in the original fact table. These original tuples all share the same values in the structural attributes. However this does not mean that all tuples in the compressed fact table, that refers the same tuple in the **Delta** table all share the same values in the structural attributes as only the delta patterns are shared.

When the structural attributes have the same value there is no reason to apply a delta pattern. Such a pattern would anyway only consist of zeroes. Therefore there is no need to waste space making references to the **Delta** table for these attributes.

Hence the new schema is not changed with regard to the structural attributes, so these are just moved directly to the **CBusdata** table. The format of this table is for now the following

CBusdata(`LineDimPK`, `JourneyNoDD`, ...)

6.3 Accumulative

In order to compress an accumulative attribute the approach looks a lot like the one used in Section 5.2. Two attributes are created in the **CBusdata** table for each original attribute (we shall refer to this as *col*) in the **Busdata** table. The two attributes that are added are denoted as *col*`Delta` and *col*`Max` where *col*`Delta` is a foreign key to the tuple in the **Delta** table that holds the delta pattern, and the *col*`Max` attribute holds the maximum value of the original attributes. The format of the **CBusdata** table is for now the following

CBusdata(..., `TargetArrivalDimPKDelta`, `TargetArrivalDimPKMax`, `ActualArrivalDimPKDelta`, `ActualArrivalDimPKMax`, `PointDimDDDelta`, `PointDimDDMax`, ...)

6.4 Critical

As described in Section 4.3 the critical attributes are just a specializations of the accumulative attributes. Therefore basically the same compression technique as in Section 6.3 is applied. The only difference is that yet another attribute is added to the **CBusdata** table. This attribute is denoted as *col*`Min` which holds the minimum value of the original attribute. The format of the **CBusdata** table is now the following

CBusdata(..., `DelayInSecondsDelta`, `DelayInSecondsMin`, `DelayInSecondsMax`, ...)

6.5 Summary

We have now introduced the compression method for the three different attribute types and how they are inserted into in the **CBusdata** table. Before the model is complete we shall introduce yet another attribute which is denoted `SizeOfDelta` for the purpose of performance which we will elaborate on in Section 8.

The new full schema of the simple data warehouse from Section 3 is shown in Figure 7.

6.6 Inserts

Section 5.2 describes a general strategy for transforming a warehouse schema into a new schema corresponding to the model and also how data is processed to be held in the compressed fact table. This however

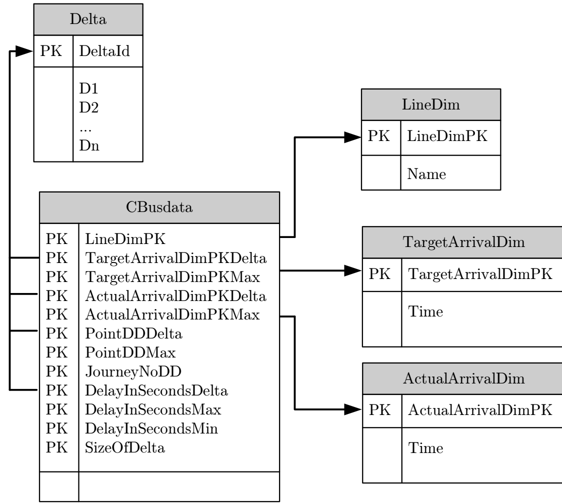


Figure 7: The new warehouse schema

does not elaborate on how single inserts to the compressed fact table are done. This section will clarify how these inserts are done on an already compressed table. Listing 3 shows a general approach for making single inserts in the compressed warehouse. This algorithm must be executed atomically.

If a tuple in the compressed fact table with the same structural attributes exists and `SizeOfDelta`$< n$

- Update this tuple to reflect the new maximum and minimum values
- If the delta patterns for this tuple are referred to by other tuples, copy these patterns
- Change these delta patterns to reflect on the inserted tuple

Else

- Create a tuple in the compressed fact table with the value of the inserted tuple
- Create a delta pattern, if it does not exist, with the value $(0, null, ..., null)$
- Refer to this delta pattern from the newly created tuple

Listing 3: Insert algorithm

7 Decompression

Even though the new representation of the fact table is different from the original, it is still possible to recreate the original fact table. The decompression happens at query level during execution, where the original fact table is calculated in a view and afterwards the query is performed on that view. The creation of this view can be done automatically. Listing 4 shows the algorithm for creating the view. This function takes in the parameters `n` and `Attribute`. `n` corresponds to the n as described in Section 5.2 which is the number of values stored in each delta pattern. `Attribute` is the set of attributes in the original table. Each of these attributes has a distinct short name which we will denote col_Δ. The algorithm builds the SQL query and returns it as a string. The view recreating the standard **Busdata** table from Section 3 is shown in Listing 5.

The algorithm in Listing 4 uses four variables called `select`, `from`, `where` and `query` for building up the query. Line 3 secures that every attribute in the **Delta** table is joined with the fact table. Line 4 secures that every attribute in the original fact table is put in the view. Lines 5-6 adds the structural attributes to the select statement. Lines 8-10 creates the select, from and where statement for the accumulative and critical attributes. Line 13 makes sure that delta attributes with *null* values are not joined. Line 14 creates the subselect and in lines 15-18 the `UNION ALL` is added to the query. Line 19 resets the `select`, `from` and `where` variables to the empty string as they are reused in the `for` loop. Finally the whole query is returned in line 21.

8 Query Rewriting

In this section we shall discuss a general approach for rewriting the queries performed on the compressed warehouse. Of course all queries can be performed on the view described in Section 7, but the execution time is almost always longer than on the standard warehouse. Therefore we shall introduce general rewrite rules from a query on the standard warehouse to a query on the compressed warehouse.

As described in Section 6.5 another attribute called `SizeOfDelta` is added to the **CBusdata** table. This attribute is used for achieving better performance when applying the rewrite rules. It indicates how many attributes in the tuple in the **Delta** table that are not

```
1  function create_view(n, Attribute) : String {
2      query ←(CREATE VIEW Busdata AS ()
3      for (1 to n as i) {
4          foreach(Attribute as col) {
5              if (col ∈ S)
6                  select ←select + col
7              else {
8                  select ←select + ((col_max + col_Δ.Dδ) AS
                       col)
9                  from ←from + (Delta AS col_Δ)
10                 where ←where + (
                       colDelta = col_Δ.DeltaId)
11             }
12         }
13         where ←where + (AND SizeOfDelta >= i)
14         query ←query + (SELECT select FROM from
               WHERE where)
15         if (i ≠ n)
16             query ←query + (UNION ALL)
17         else
18             query ←query + ())
19         select, from, where ←ε
20     }
21     return query
22 }
```

Listing 4: The algorithm for creating view

```
1  CREATE VIEW Busdata AS (
2      SELECT LineDimPK,
3          (ActualArrivalDimPKMax+AA.D1) AS
               ActualArrivalDimPK,
4          (TargetArrivalDimPKMax+TA.D1) AS
               TargetArrivalDimPK,
5          (PointDimDDPKMax+PO.D1) AS PointDimDD,
6          JourneyNo,
7          (DelayInSecondsMax+DL.D1) AS
               DelayInSeconds
8      FROM
9          Fact, Delta AS AA, Delta AS TA, Delta
               AS PO, Delta AS DL
10     WHERE ActualArrivalDimPKDelta=AA.DeltaId
11         AND TargetArrivalDimPKDelta=TA.DeltaId
12         AND PointDimDDDelta=PO.DeltaId
13         AND DelayInSecondsDelta=DL.DeltaId
14         AND SizeOfDelta >= 1
15 UNION ALL
16     SELECT LineDimPK,
17         (ActualArrivalDimPKMax+AA.D2)
18 ...
19         DelayInSecondsDelta=DL.DeltaId
20         AND SizeOfDelta >= N)
```

Listing 5: The view created by the algorithm in Listing 4

null. This ensures that only for example three attributes are calculated if the actual size of the pattern is three.

8.1 General Approach

This section will apply the rewrite rules for different parts of the standard fact table SQL query. We shall only describe the comparison operators $>, =$ and $<$ from the `WHERE` clause. In addition, the aggregation functions `sum`,`max`,`count` and `avg` will be described. Other operators and aggregation functions can be rewritten using similar rules. During the next three sections definitions are introduced that will be used in Section 8.1.4 for the final rewrite.

8.1.1 Initial Definitions

$\mathfrak{F}$ is defined as the attributes in the dimension tables. Now the set B can be defined as $\mathbb{N} \cup \mathcal{S} \cup \mathcal{A} \cup \mathcal{C} \cup \mathfrak{F}$ and the element $x \in B$. All queries to the standard warehouse should be in the form as shown in Listing 6 where `CL` $\subseteq B$, `AL` $\subseteq \{\mathrm{f}(x_1), \ldots, \mathrm{f}(x_k)\}$ where $f \in \{sum, max, count, avg\}$, `FL` is a set of dimension tables, `WC` is a the `WHERE` clause containing only the boolean operators `AND`, `OR` and `NOT`. Additional two sets are defined; `WL` which is a subset of B containing the attributes used in `WC`. We define Q as a subset of B with the attributes from `AL` and `HC` in union with `WL` $\cup$ `CL` $\cup$ `GL`.

```
1 SELECT ⟨CL⟩, ⟨AL⟩
2 FROM Fact, ⟨FL⟩
3 WHERE ⟨WC⟩
4 GROUP BY ⟨GL⟩
5 HAVING ⟨HC⟩
```

Listing 6: A query for the standard warehouse

If nothing is written in the `WC` or `HC` clause then we shall assume that $1 = 1$ is written. If the `AL` is not empty and nothing is written in the `GL` clause then the `GL` is implied to the empty set.

8.1.2 Rewriting Q

Now we take the initial steps for rewriting the query on the standard warehouse to a query on the compressed warehouse. We start off by rewriting the elements in

the set Q defined in Section 8.1. The algorithm for rewriting the set Q is described in Listing 7.

```
1  procedure transform_q(Q) {
2      foreach(Q as col) {
3          if(col ∈ S ∪ N ∪ F) {
4              CL′ ←CL′ ∪ {(col)}
5          }
6          elseif (col ∈ A ∪ C) {
7              CL′ ←CL′ ∪ {(col_max + col_Δ.Dδ AS col)}
8              FL′ ←FL′ ∪ {(Delta AS col_Δ)}
9              WL′ ←WL′ ∪ {(colDelta = col_Δ.DeltaId)}
10         }
11     }
12 }
```

Listing 7: The algorithm for rewriting Q

The algorithm takes in a parameter Q and results in three new sets CL′,FL′ and WL′ which are saved for later. Lines 3-4 add the structural attributes directly to the set CL′. Lines 6-9 add the necessary elements to CL′,FL′ and WL′ for the accumulative and critical attributes. The symbol δ will later be substituted with the numbers 1 to n. col_{max} is just a notion of the attribute with the suffix max. Line 7 gets the actual value from the delta table for the given attribute. Lines 8-9 make sure that the **Delta** table is joined for each attribute that needs to access the values in the **Delta** table.

8.1.3 Rewriting WC

As Section 8.1.2 covered the rewrite of the set Q, the only thing left from Listing 6 that needs to be rewritten is the WC clause. This should be done by applying the rewrite rules from Appendix A onto each boolean subexpression in WC. We now define WC′ which contains the rewritten expressions from WC. This is a result of syntactical substitution. The expressions in WC′ are furthermore put in conjunction normal form in order to ease the final rewrite process.

Appendix A describes general rewrite rules for the WC clause. For example when the WC clause in Listing 1 says DelayInSeconds$>$ 180 then a suitable rule must be found. As DelayInSeconds is a critical attribute and 180 is a numeral then the rule for $C > N$ must be applied. The rule says $C_{\max} > N \wedge (C_{\max} + C_{\Delta}.D\delta) > N$. The first part of the rule makes sure that rows are joined with the **Delta** table only if the delta pattern actually contains a value larger than 180. The second part of the rule makes the join with the delta table. Therefore the WC′ clause will now look like DelayInSecondsMax $>$ 180 AND (DelayInSecondsMax+DelayInSecondsAA.Dδ) $>$ 180.

8.1.4 Final Rewrite

In the preceding sections we have defined and constructed the sets CL′,FL′ and WL′ and the boolean expression WC′ which will be used for the rewrite. The only thing left is to define yet another boolean expression WC″ which should be set to $(\bigwedge_{y \in \mathtt{WL}'} y) \wedge \mathtt{WC}'$. This just says that every element in WL′ is and'ed together and finally and'ed together with WC′. Furthermore subclauses in WC″ containing only elements in the compressed fact table should be placed first (to promote short-circuiting). The final rewrite should be done by using Listing 8.

```
1  SELECT ⟨CL⟩, ⟨AL⟩ FROM (
2        (SELECT ⟨CL′_{δ→1}⟩
3        FROM CFact, ⟨FL′⟩, ⟨FL⟩
4        WHERE SizeOfDelta >= 1 AND ⟨WC″_{δ→1}⟩)
5     UNION ALL
6  ...
7     UNION ALL
8        (SELECT ⟨CL′_{δ→n}⟩
9        FROM CFact, ⟨FL′⟩, ⟨FL⟩
10       WHERE SizeOfDelta >= n AND ⟨WC″_{δ→n}⟩)
11 ) GROUP BY ⟨GL⟩ HAVING ⟨HL⟩
```

Listing 8: The final rewrite

The SQL statement in Listing 8 consists of one outer statement and one inner statement that consists of n union'ed statements. Line 1 selects the attributes and the aggregations from the inner statement. Lines 2-5 are the select statement that is union'ed n times, where $\delta \rightarrow m$ means that δ is substituted with m in the concrete element.

8.2 Further Rewrite

In the case where GL $\subseteq \mathcal{S}$ a further rewrite of the queries can be done. Remember that if GL$= \emptyset$ then it is also a subset of $\mathcal{S}$. In order to do this rewrite the rules from Section 8.1 must first be applied. We define three new boolean expressions w, w' and w'' which all are sub expressions of WC′. w is defined as the sub-clauses

in $\mathtt{WC}'$ that only contain elements from the compressed fact table and $\mathbb{N}$. A temporary expression t is defined as the sub-clauses from $\mathtt{WC}'$ containing elements from the delta table and not containing elements from $\mathfrak{F}$, now w' is defined as $\mathtt{SizeOfDelta} \geq \delta \wedge t$. Finally w'' is the sub-clauses containing elements from $\mathfrak{F}$. How these three sets are related is shown in Figure 8. Now AL and HC must be modified as described in Appendix B, which results in $\mathtt{AL}'$ and $\mathtt{HC}'$.

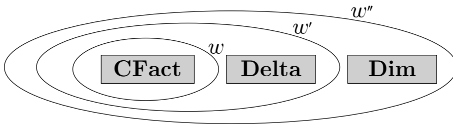


Figure 8: The relation between w, w', w''

Now the algorithm in Listing 9 can be used for the final rewrite of a query on a standard warehouse to a query on the compressed warehouse. $\mathtt{WL}'''$ should be set to $(\bigwedge_{y \in \mathtt{WL}'} y)$

```
1  SELECT ⟨CL⟩ , ⟨AL′⟩
2  FROM CFact , ⟨FL′⟩ , ⟨FL⟩
3  WHERE ⟨w⟩ AND ⟨WL′′′⟩
4  GROUP BY ⟨GL⟩
5  HAVING ⟨HC⟩
```

Listing 9: The new final rewrite

8.3 Example Rewrite

In this section we will apply the rules described above onto the example given in Listing 1. The result, when the rules have been applied, is shown in Listing 10. As it appears, the query is substantially larger than the one in Listing 1.

The lines 2-9 in Listing 10 refer to the lines 2-4 in Listing 1 and the Lines 12-18 in Listing 10 refer to the lines 6-7 in Listing 1.

9 Public Transportation Case

In order to demonstrate the model described in this article, we have begun a co-operation with Nordjyllands Trafikselskab (NT), which is the local bus company located in Northern Jutland. NT has GPS devices in all

```
1  SELECT
2    (SELECT SUM(
3      b2n(SizeOfDelta>=1 AND
4      (DelayInSecondsMax+disaaa.D1)>180)*1*1+
5  ...
6      b2n(SizeOfDelta>=n AND
7         (DelayInSecondsMax+disaaa.Dn)>180)
             *1*1)
8      FROM CBusdata, Delta as disaaa
9      WHERE DelayInSecondsMax>180 AND
10        (DelayInSecondsDelta=disaaa.DeltaId))
11 /
12   (SELECT SUM(
13     b2n(SizeOfDelta>=1 AND 1=1)*1*1+
14 ...
15     b2n(SizeOfDelta>=n AND 1=1)*1*1)
16     FROM CBusdata
17     WHERE 1=1)
18 *100 AS Delay
```

Listing 10: Rewrite of Listing 1

their busses and real-time data is sent to a server. GPS data for one day sums up to about 100.000 tuples.

The first thing to do is to convert their real-time data into data warehouse schema. The format of the real-time data from the **expdatedcall** table is described in the `PubTrans` document[Fjä03]. The format of the fact table from the standard warehouse is outlined in Figure 9. For the sake of simplicity we have omitted the dimension tables as they are unimportant in this context. Appendix C shows the full star schema of the standard data warehouse.

Microsoft SQL-Server 2000 is used as the database server. As this DBMS does not support more than 16 combined primary keys, we shall build the fact table without using primary keys, as a key consisting of only 16 attributes cannot be found.

To compress this warehouse the different attribute types as described in Section 4 must be identified. Figure 9 shows how the different attributes are categorized. This categorization is based on knowledge gained by studying data provided by NT.

Currently at NT, real-time data is discarded every day in order to keep data for three days only. Therefore a data warehouse model is easy to apply to this existing model as the data just should be copied to another database keeping the data warehouse before the data is deleted from the source. Here the conversion to a standard fact table can be done at any time (typically this should be done at night in order to prevent slow-

Attribute	Type
`LineDimPK`	Structural
`ProductDimPK`	Structural
`ContractorDimPK`	Structural
`OriginDimPK`	Structural
`EarliestDepartureDateDimPK`	Structural
`EarliestDepartureTimeDimPK`	Accumulative
`TargetDepartureDateDimPK`	Structural
`TargetDepartureTimeDimPK`	Accumulative
`ActualDepartureDateDimPK`	Structural
`ActualDepartureTimeDimPK`	Accumulative
`LatestArrivalDateDimPK`	Structural
`LatestArrivalTimeDimPK`	Accumulative
`TargetArrivalDateDimPK`	Structural
`TargetArrivalTimeDimPK`	Accumulative
`ActualArrivalDateDimPK`	Structural
`ActualArrivalTimeDimPK`	Accumulative
`OperatorDimPK`	Structural
`JourneyDimPk`	Structural
`ArrivalDelayDimPK`	Critical
`DepartureDelayDimPK`	Critical
`IsAtStopAreaDimPK`	Accumulative

Figure 9: Types of attributes in the NT case

ing other processes down) whereafter this data can be compressed and placed in the actual data warehouse. All these tasks can be done automatically.

10 Performance

In this section we will test performance on the model described in this article. The performance will be measured both in storage and speed in comparison to the standard warehouse. Tests will be performed on PubTrans data with $598,147$ tuples in the fact table in the standard warehouse which corresponds to 8 days of data from NT. The tests were performed on a computer with a 1.5 Ghz Pentium M processor and 512 MB RAM. The queries were run five times and the best and the worst execution times were ignored and the average of the remaining three was calculated.

10.1 Simple Warehouse

During this article the SQL query from Listing 1 has been a running example and now the time has come to test it's performance. The simple warehouse has been filled with real PubTrans data in order to make the tests as realistic as possible. With regard to the compressed warehouse the tests have been performed with an n value of $7, 14, 21$ and 28.

Left side of Figure 10 shows the storage gain compared to the standard warehouse. This is measured on the fact table only as the dimension tables are the same. With an n value of 28 the compressed warehouse size is reduced to 56% of the standard warehouse.

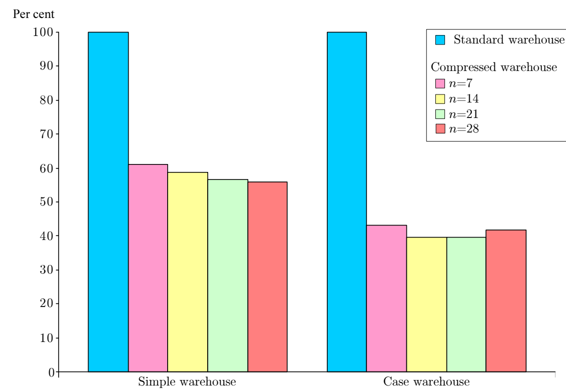


Figure 10: The storage test of the simple- and the full warehouse

The query from Listing 1 is a rather realistic query which is also good for performance testing as it must select some specific data from the warehouse but also count the size of the whole warehouse.

Figure 11 shows the speed test of the simple warehouse. The execution time of the query on the standard warehouse is 190 milliseconds where the best of the queries on the compressed warehouse is 315 milliseconds. As described in Section 8 the execution time on the view is almost always larger than the one on the standard warehouse. This can also seen from the figure where the execution time is $12,717$ milliseconds.

As it appears from this example warehouse a storage gain is achieved. A slightly worsening in the execution speed for the query from Listing 1 is experienced. But

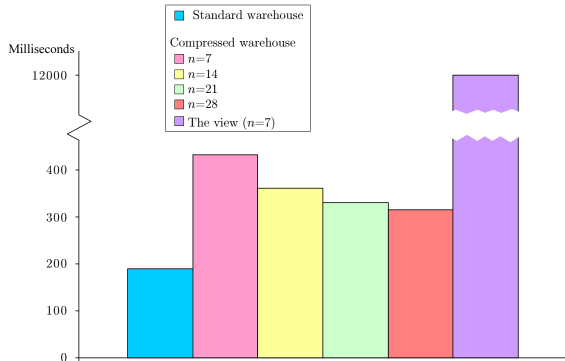


Figure 11: The speed test of the small warehouse

as it will appear from the next section not all queries are slower than the one on the standard warehouse.

10.2 Transportation Case

In this section we will test the performance of the warehouse from the public transportation case described in Section 9. Some of the queries are deduced from wishes expressed by NT. In this section we will not look at the performance of the view as tests have shown that the execution time is significantly worse.

As in Section 10.1 the performance in comparison to the standard warehouse is tested with regard to storage and speed. Also here n values of $7, 14, 21$ and 28 have been tested. The queries on the compressed warehouse is rewritten using the approach from Section 8.

A description of each query that is tested is shown below. The queries can be seen in Appendix D:

Q1 For two given points determine the average time between them.

Q2 Determine the average time between a set of points.

Q3 Determine the percentage delay for a given contractor.

The right side of Figure 10 shows the storage gain in the case warehouse compared to the standard warehouse. With an n value of 14 the warehouse only occupies 40% of the standard warehouse. As it appears from the figure a significantly larger compression in the case warehouse is achieved than on the simple warehouse. An explanation for this could be that the case warehouse contains a lot more attributes that are compressed and uses the same delta table which thereby potentially reuses more patterns.

Figure 12 shows how the rewritten queries perform speed wise compared to the standard warehouse with different n values.

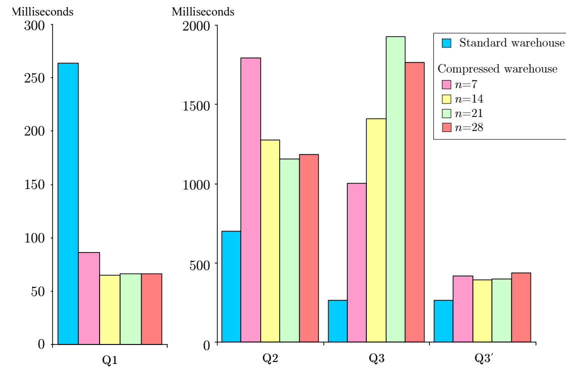


Figure 12: The speed test of the case warehouse

From Figure 12 one can see that the query **Q1** performs significantly better on the compressed warehouse than on the standard warehouse, that is a speed increase of 300%. The reason for this improvement is the use of structural attributes in the `WHERE` clause. This makes it able to eliminate a lot of rows before the join. As the fact table is n times smaller it is much faster to run through this.

Query **Q2** is a further developed version of query **Q1**, but running the query for many sets of points. Intuitively it should run faster than the one on the standard warehouse as it is basically the same query as **Q1**. One reason why this is not the case is that the execution planner cannot optimize as much as it can on the standard warehouse.

Query **Q3** is rather slow on the compressed warehouse compared to the standard warehouse. The reason for this is that a dimension table is involved which makes the query run much slower. This it because the rewrite rules state that the join must be done for each union. One could imagine that a preprocessor could be

built that makes this join before the query and substitutes parts of the WHERE clause with these values. The results for query **Q3′** show some preliminary results of a rewrite of query **Q3**.

As it appears from this section in some cases it is safe to choose a large n both speed and storage wise. When using the rule of thumb from Section 6.1 on the public transportation case n should be about 15. However, in some cases, for example **Q3**, the execution time grows together with n. Therefore the user must make several tests when determining n in order to get one that fits his needs.

11 Conclusion

In this paper we have presented our idea for data warehouse optimization through differential patterns and schema change. We have shown that with a combination of differential compression, reuse of patterns and schema change a storage gain can be achieved. This was done without limiting the query possibilities or loosing precision in the query results.

We have shown algorithms for compressing and decompressing the data and demonstrated a way for making the compression transparent to the user. The price for compressing the warehouse is more complex queries but these can be rewritten automatically. The model was implemented, and we showed some preliminary results that were obtained on real data. We have shown that space can almost always be saved and in some cases even better performance can be achieved.

In this article we have focused solely on data warehouses, but nothing prevents applying this model to other databases.

Acknowledgements

We would like to thank Jens Mogensen, Nordjyllands Trafikselskab, for co-operation and for providing us with data, without which this project could not have been realized. We would also like to thank Kristian Torp for guidance and help throughout the project.

References

[Che02] Zhiyuan Chen. *Building Compressed Database Systems*. PhD thesis, Cornell University, 2002.

[Fjä03] Stefan Fjällemark. *PubTrans Real-time Output Interface Reference Manual - Version 2.3E*. PubTrans User Group, 2003.

[GRDT99] Kiran B. Goyal, Krithi Ramamritham, Anindya Datta, and Helen M. Thomas. Indexing and compression in data warehouses. In *DMDW*, page 11, 1999.

[Kim02] Ralph Kimball. *The Data Warehouse Toolkit*. Wiley, 2002.

[Kor01] Zhiyuan Chen & Johannes Gehrke & Flip Korn. Query optimization in compressed database systems. *SIGMOD*, 2001.

[PRWS00] Scott F. Kaplan Paul R. Wilson and Yannis Smaragdakis. The case for compressed caching in virtual memory systems. *Dept. of Computer Sciences University of Texas at Austin Austin, Texas 78751-1182*, page 16, 2000.

[RH93] Mark A. Roth and Scott J. Van Horn. Database compression. *SIGMOD RECORD*, 22(3), September 1993.

[Riz03] Matthias Nicola & Haider Rizvi. Storage layout and i/o performance in data warehouses. *DMDW*, 2003.

[Ses95] Gautam Ray & Jayant R. Haritsa & S. Seshadri. Database compression: A performance enhancement tool. *COMAD*, 1995.

[Sun01] Abraham Silberschatz & Henry F. Korth & S. Sundarshan. *Database System Concepts*. McGraw-Hill, 4th bk&cd edition edition, 2001.

A Rewrite Rule for `WC`

This appendix deals with the rewriting of the `WC` clause from Section 8.1. Now each boolean sub-expression in `WC`, on the form $y \oplus y'$, where $y, y' \in B$ and $\oplus \in \{>, =, <\}$, should be rewritten using the rules described in this appendix.
The symbol $\rightarrow$ means that the query should be rewritten using the rule on the right hand side.
The symbol $\Rightarrow$ means that the query should be rewritten using the rule on the right hand side. The right hand side should furthermore be rewritten using other rules from this appendix.
A number of letters are used in this section. $S, S' \in \mathcal{S}$, $A, A' \in \mathcal{A}$, $C, C' \in \mathcal{C}$, $N, N' \in \mathbb{N}$ and $F, F' \in \mathfrak{F}$. Furthermore a symbol $\oplus'$ is defined which means that the symbol is inverted, for example if $\oplus$ represents a $<$ then $\oplus'$ represents $>$. If $\oplus$ represents an $=$ then $\oplus'$ also represents an $=$.
The symbol $x \in \mathcal{S} \cup \mathcal{A} \cup \mathcal{C} \cup \mathbb{N} \cup \mathfrak{F}$ which represents the variable that these rules apply to. The symbol Δ is a unique short name for the attribute. The symbol δ will later be substituted with the numbers 1 to n.

A.1 Rewriting $S \oplus x$

These rules apply when a structural attribute is compared to x.

$\mathcal{S}$ attributes can be compared directly	$\underline{S \oplus S'}$	$\rightarrow$	$S \oplus S'$
In the two first rules a short circuiting can be done before the join	$\underline{S < A}$	$\rightarrow$	$S < A_{\max} \wedge S < (A_{\max} + A_{\Delta}.D\delta)$
	$\underline{S = A}$	$\rightarrow$	$S \leq A_{\max} \wedge S = (A_{\max} + A_{\Delta}.D\delta)$
	$\underline{S > A}$	$\rightarrow$	$S > (A_{\max} + A_{\Delta}.D\delta)$
The first rule is similar to the first above. The next a new short circuiting clause is appended. The last is short circuited on C_{min}	$\underline{S < C}$	$\rightarrow$	$S < C_{\max} \wedge S < (C_{\max} + C_{\Delta}.D\delta)$
	$\underline{S = C}$	$\rightarrow$	$S \leq C_{\max} \wedge S \geq C_{\min} \wedge S = (C_{\max} + C_{\Delta}.D\delta)$
	$\underline{S > C}$	$\rightarrow$	$S > C_{\min} \wedge S > (C_{\max} + C_{\Delta}.D\delta)$
	$\underline{S \oplus N}$	$\rightarrow$	$S \oplus N$
$\mathcal{S}$ and $\mathfrak{F}$ attributes can be compared directly	$\underline{S \oplus F}$	$\rightarrow$	$S \oplus F$

A.2 Rewriting $A \oplus x$

These rules apply when an accumulative attribute is compared to x.

Use the rules from Section A.1

$\underline{A \oplus S} \;\Rightarrow\; S \oplus' A$

No Short circuiting can be done here. The two attributes must be compared fully.

$\underline{A \oplus A'} \;\rightarrow\; (A_{\max} + A_{\Delta}.D\delta) \oplus (A'_{\max} + A'_{\Delta}.D\delta)$

The first rule must be compared fully where the two others can be short circuited on C_{min}

$\underline{A < C} \;\rightarrow\; (A_{\max} + A_{\Delta}.D\delta) < (C_{\max} + C_{\Delta}.D\delta)$
$\underline{A = C} \;\rightarrow\; A_{\max} \geq C_{\min} \wedge (A_{\max} + A_{\Delta}.D\delta) = (C_{\max} + C_{\Delta}.D\delta)$
$\underline{A > C} \;\rightarrow\; A_{\max} > C_{\min} \wedge (A_{\max} + A_{\Delta}.D\delta) > (C_{\max} + C_{\Delta}.D\delta)$

These rules are similar to the ones above.

$\underline{A < N} \;\rightarrow\; (A_{\max} + A_{\Delta}.D\delta) < N$
$\underline{A = N} \;\rightarrow\; A_{\max} \geq N \wedge (A_{\max} + A_{\Delta}.D\delta) = N$
$\underline{A > N} \;\rightarrow\; A_{\max} > N \wedge (A_{\max} + A_{\Delta}.D\delta) > N$

These rules are similar to the ones above

$\underline{A < F} \;\rightarrow\; (A_{\max} + A_{\Delta}.D\delta) < F$
$\underline{A = F} \;\rightarrow\; A_{\max} \geq F \wedge (A_{\max} + A_{\Delta}.D\delta) = F$
$\underline{A > F} \;\rightarrow\; A_{\max} > F \wedge (A_{\max} + A_{\Delta}.D\delta) > F$

A.3 Rewriting $C \oplus x$

These rules apply when a critical attribute is compared to x.

Use the rules from Section A.1

$\underline{C \oplus S} \;\Rightarrow\; S \oplus' C$

Use the rules from Section A.2

$\underline{C \oplus A} \;\Rightarrow\; A \oplus' C$

The first rule is short circuited on C_{min} and the second is also short circuited on C_{max}. Instead of the last rule the first shall be applied

$\underline{C < C'} \;\rightarrow\; C_{\min} < C'_{\max} \wedge (C_{\max} + C_{\Delta}.D\delta) < (C'_{\max} + C'_{\Delta}.D\delta)$
$\underline{C = C'} \;\rightarrow\; C_{\min} \leq C'_{\max} \wedge C_{\max} \geq C'_{\min} \wedge (C_{\max} + C_{\Delta}.D\delta) = (C'_{\max} + C'_{\Delta}.D\delta)$
$\underline{C > C'} \;\Rightarrow\; C' < C$

The first and last rule are the same just with inverted operators and are short circuited on different attributes. The rule in the middle is short circuited on both attributes

$\underline{C < N} \;\rightarrow\; C_{\min} < N \wedge (C_{\max} + C_{\Delta}.D\delta) < N$
$\underline{C = N} \;\rightarrow\; C_{\max} \geq N \wedge C_{\min} \leq N \wedge (C_{\max} + C_{\Delta}.D\delta) = N$
$\underline{C > N} \;\rightarrow\; C_{\max} > N \wedge (C_{\max} + C_{\Delta}.D\delta) > N$

These rules are similar to the ones above.

$\underline{C < F} \;\rightarrow\; C_{\min} < F \wedge (C_{\max} + C_{\Delta}.D\delta) < F$
$\underline{C = F} \;\rightarrow\; C_{\max} \geq F \wedge C_{\min} \leq F \wedge (C_{\max} + C_{\Delta}.D\delta) = F$
$\underline{C > F} \;\rightarrow\; C_{\max} > F \wedge (C_{\max} + C_{\Delta}.D\delta) > F$

A.4 Rewriting $N \oplus x$

These rules apply when a number is compared to x.

Use the rules from Section A.1	$\underline{N \oplus S} \Rightarrow S \oplus' N$
Use the rules from Section A.2	$\underline{N \oplus A} \Rightarrow A \oplus' N$
Use the rules from Section A.3	$\underline{N \oplus C} \Rightarrow C \oplus' N$
$\mathbb{N}$ attributes can be compared directly	$\underline{N \oplus N'} \rightarrow N \oplus N'$
$\mathbb{N}$ and $\mathfrak{F}$ attributes can be compared directly	$\underline{N \oplus F} \rightarrow N \oplus F$

A.5 Rewriting $F \oplus x$

These rules apply when a foreign attribute is compared to x.

Use the rules from Section A.1	$\underline{F \oplus S} \Rightarrow S \oplus' F$
Use the rules from Section A.2	$\underline{F \oplus A} \Rightarrow A \oplus' F$
Use the rules from Section A.3	$\underline{F \oplus C} \Rightarrow C \oplus' F$
Use the rules from Section A.4	$\underline{F \oplus N} \Rightarrow N \oplus' F$
$\mathfrak{F}$ attributes can be compared directly	$\underline{F \oplus F'} \rightarrow F \oplus F'$

B Rewrite Rule for AL and HC

This appendix deals with the rewriting of the AL and HC clauses from Section 8.2. The algorithms in Section 8.1 must be applied to the query before these rules are applied.
The symbol $\rightarrow$ means that the query should be rewritten using the rule on the right hand side.
The symbol $\Rightarrow$ means that the query should be rewritten using the rule on the right hand side. The right hand side should furthermore be rewritten using other rules from this appendix.
The following functions are needed for the rewrite:

$\underline{b2n(b)}$ $\rightarrow$ CASE WHEN b THEN 1 ELSE 0 END

$\underline{n2n(n)}$ $\rightarrow$ CASE WHEN n IS NULL THEN 0 ELSE n END

$\underline{greatest(b_1, n_1, b_2, n_2, \ldots, b_j, n_j)}$ $\rightarrow$ SELECT MAX(a) FROM (SELECT n_1 AS a WHERE b_1 UNION...UNION SELECT n_j AS a WHERE b_j) AS b

We define the number v as 1 if FL $= \emptyset$ else it is (SELECT COUNT($\star$) FROM <FL> WHERE SizeOfDelta>=δ AND WC$''$).
The symbol Δ is a unique short name for the attribute. $\delta \rightarrow m$ means that δ is substituted with m in the concrete element. For example if the rule says $w'_{\delta \rightarrow 1}$ then every time the symbol δ appears in w' it should be substituted with 1.

B.1 Rewriting sum(x)

These rules are used for summation. This takes the sum of all values from the **Delta** table and if the where clause applies to these values then 1 is returned. So if the where clause applies to all values and n is 5 then it basically reads sum(5).

$\underline{\text{sum}(S)} \rightarrow \text{sum}(b2n(w'_{\delta\rightarrow 1}) \cdot v_{\delta\rightarrow 1} \cdot S + \ldots + b2n(w'_{\delta\rightarrow n}) \cdot v_{\delta\rightarrow n} \cdot S)$

$\underline{\text{sum}(A)} \rightarrow \text{sum}(b2n(w'_{\delta\rightarrow 1}) \cdot v_{\delta\rightarrow 1} \cdot n2n(A_{\max} + A_{\Delta}.D1) + \ldots + b2n(w'_{\delta\rightarrow n}) \cdot v_{\delta\rightarrow n} \cdot n2n(A_{\max} + A_{\Delta}.Dn))$

$\underline{\text{sum}(C)} \rightarrow \text{sum}(b2n(w'_{\delta\rightarrow 1}) \cdot v_{\delta\rightarrow 1} \cdot n2n(C_{\max} + C_{\Delta}.D1) + \ldots + b2n(w'_{\delta\rightarrow n}) \cdot v_{\delta\rightarrow n} \cdot n2n(C_{\max} + C_{\Delta}.Dn))$

$\underline{\text{sum}(N)} \rightarrow \text{sum}(b2n(w'_{\delta\rightarrow 1}) \cdot v_{\delta\rightarrow 1} \cdot N + \ldots + b2n(w'_{\delta\rightarrow n}) \cdot v_{\delta\rightarrow n} \cdot N)$

$\underline{\text{sum}(F)} \rightarrow \text{sum}(b2n(w'_{\delta\rightarrow 1}) \cdot (\texttt{SELECT SUM(}F\texttt{) FROM } \langle\texttt{FL}\rangle \texttt{ WHERE } w''_{\delta\rightarrow 1}) + \ldots + b2n(w'_{\delta\rightarrow n}) \cdot (\texttt{SELECT SUM(}F\texttt{) FROM } \langle\texttt{FL}\rangle \texttt{ WHERE } w''_{\delta\rightarrow n}))$

B.2 Rewriting max(x)

These rules basically takes the greatest value of the values from the **Delta** table where the where clause applies

$\underline{\max(S)} \rightarrow \max(\mathit{greatest}((w'_{\delta\rightarrow 1} \wedge v_{\delta\rightarrow 1} \geq 1) \vee \ldots \vee (w'_{\delta\rightarrow n} \wedge v_{\delta\rightarrow n} \geq 1), S))$

$\underline{\max(A)} \rightarrow \max(\mathit{greatest}(w'_{\delta\rightarrow 1} \wedge v_{\delta\rightarrow 1} \geq 1, (A_{\max} + A_{\Delta}.D1), \ldots, w'_{\delta\rightarrow n} \wedge v_{\delta\rightarrow n} \geq 1, (A_{\max} + A_{\Delta}.Dn)))$

$\underline{\max(C)} \rightarrow \max(\mathit{greatest}(w'_{\delta\rightarrow 1} \wedge v_{\delta\rightarrow 1} \geq 1, (C_{\max} + C_{\Delta}.D1), \ldots, w'_{\delta\rightarrow n} \wedge v_{\delta\rightarrow n} \geq 1, (C_{\max} + C_{\Delta}.Dn)))$

$\underline{\max(N)} \rightarrow \max(\mathit{greatest}((w'_{\delta\rightarrow 1} \wedge v_{\delta\rightarrow 1} \geq 1) \vee \ldots \vee (w'_{\delta\rightarrow n} \wedge v_{\delta\rightarrow n} \geq 1), N))$

$\underline{\max(F)} \rightarrow \max(\mathit{greatest}(w'_{\delta\rightarrow 1} \wedge v_{\delta\rightarrow 1} \geq 1, (\texttt{SELECT MAX(F) FROM } \langle\texttt{FL}\rangle \texttt{ WHERE } w''_{\delta\rightarrow 1}), \ldots, w'_{\delta\rightarrow n} \wedge v_{\delta\rightarrow n} \geq 1, (\texttt{SELECT MAX(F) FROM } \langle\texttt{FL}\rangle \texttt{ WHERE } w''_{\delta\rightarrow n})))$

B.3 Rewriting count(x)

Use the rules from Section B.1

$\underline{\text{count}(x)} \Rightarrow \text{sum}(1)$

B.4 Rewriting avg(x)

Use the rules from Section B.1

$\underline{\text{avg}(x)} \Rightarrow \text{sum}(x)/\text{count}(x)$

B.5 DBMS Specific Rewrite

There might be the need for DBMS specific rewrite and we will here cover those needed for Microsoft SQL-server 2000.

When the error *Cannot perform an aggregate function on an expression containing an aggregate or a subquery* is encountered this is usually due to a sub-`SELECT` within an aggregating `sum`. An example of such is shown in Listing 11 and a rewrite in Listing 13. Note that the `p1` and `p2` are in **table1** and `p3` is in **table2**.

```
1 SELECT p1,sum(8*2+3*
2       (SELECT COUNT(*) FROM table2 WHERE
            p3=p1)
3 ) FROM table1
```

Listing 11: An illegal query in Microsoft SQL-server

```
1 SELECT p1,
2       (SELECT SUM(8*2+3*x)
3        FROM (SELECT COUNT(*) AS x
4               FROM table2
5               WHERE p3=p1) AS sometable
6       )
```

Listing 13: A legal rewrite of the query from Listing 11

The error *Multiple columns are specified in an aggregated expression containing an outer reference...* is usually met when one aggregates over more than one outer reference. An example of such can be seen in Listing 12 and a legal rewrite in Listing 14.

```
1 SELECT p1,p2,
2       (SELECT SUM(p1*p2*p3)
3           FROM table2)
4 FROM table1 GROUP BY p1,p2
```

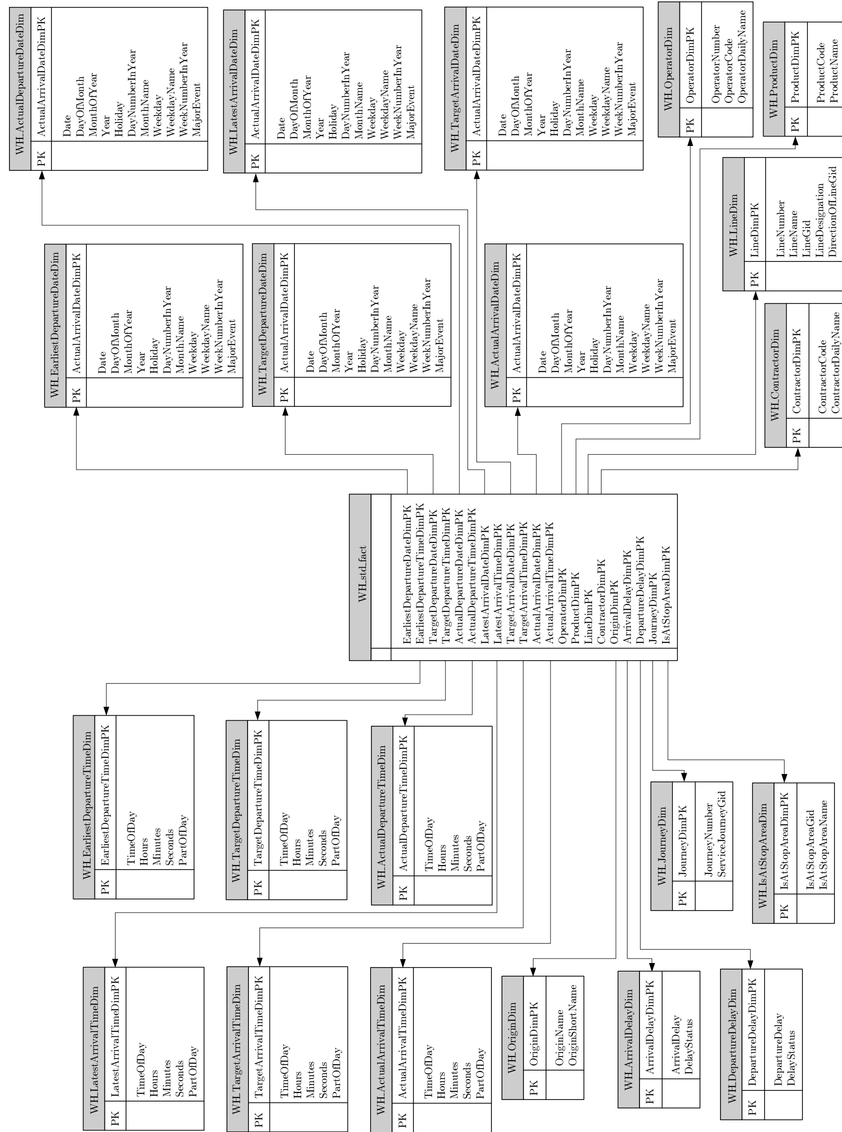
Listing 12: An illegal aggregation in Microsoft SQL-server

```
1 SELECT p1,p2,
2       (SELECT SUM(p1prime*p2prime*p3)
3        FROM table2, (SELECT p1 AS
            p1prime, p2 AS p2prime) AS
            sometable)
4 FROM table1 GROUP BY p1,p2
```

Listing 14: Rewrite of Listing 12

Please note that when using sub-selects in the `FROM` clause one must give them a name with the `AS`-keyword.

C Full Warehouse Schema



D Performance Queries

```
SELECT
      (SELECT AVG(CAST(ActualArrivalTimeDimPK AS FLOAT))
       FROM WH_std_fact
       WHERE (IsAtStopAreaDimPK=80
           AND ActualArrivalTimeDimPK>-1
           AND ActualDepartureTimeDimPK>-1
           AND WH_std_fact.LineDimPK=152
           AND WH_std_fact.OriginDimPK=15)
      )
/
      (SELECT AVG(CAST(ActualDepartureTimeDimPK AS FLOAT))
       FROM WH_std_fact
       WHERE (IsAtStopAreaDimPK=739
           AND ActualArrivalTimeDimPK>-1
           AND ActualDepartureTimeDimPK>-1
           AND WH_std_fact.LineDimPK=152
           AND WH_std_fact.OriginDimPK=15)
      )
```

Listing 15: The query **Q1**

```
SELECT Point1, Point2,LineDimPK,OriginDimPK,
      (SELECT AVG(CAST(ActualArrivalTimeDimPK AS FLOAT))
       FROM WH_std_fact
       WHERE (IsAtStopAreaDimPK=Point2
          AND ActualArrivalTimeDimPK>-1
          AND ActualDepartureTimeDimPK>-1
          AND WH_std_fact.LineDimPK=Point.LineDimPK
          AND WH_std_fact.OriginDimPK=Point.OriginDimPK)
      )
/
      (SELECT AVG(CAST(ActualDepartureTimeDimPK AS FLOAT))
       FROM WH_std_fact
       WHERE (IsAtStopAreaDimPK=Point1
           AND ActualArrivalTimeDimPK>-1
           AND ActualDepartureTimeDimPK>-1
           AND WH_std_fact.LineDimPK=Point.LineDimPK
           AND WH_std_fact.OriginDimPK=Point.OriginDimPK)
      )
FROM Point;
```

Listing 16: The query **Q2**

```
SELECT
      (SELECT CAST(COUNT(*) AS FLOAT)
       FROM WH_std_fact,WH_DepartureDelayDim
       WHERE(WH_std_fact.DepartureDelayDimPK > -1
          AND WH_std_fact.ContractorDimPK = 5
          AND NOT WH_DepartureDelayDim.DelayStatus='green'
          AND WH_std_fact.DepartureDelayDimPK = WH_DepartureDelayDim.DepartureDelayDimPK)
      )
/
      (SELECT CAST(COUNT(*) AS FLOAT)
       FROM WH_std_fact
       WHERE (WH_std_fact.DepartureDelayDimPK > -1
          AND WH_std_fact.ContractorDimPK = 5)
      )
```

Listing 17: The query **Q3**

Summary

In this article we have described a general compression method for data warehouses in order to achieve a storage and performance gain. The method describes how the attributes should be split into three different groups, namely structural, accumulative and critical. The method also states how the schema of the data warehouse should be changed. How the attributes are represented in the new warehouse is determined by the attribute type. A structural attribute is an attribute that binds information together, that is $V(\mathcal{S}, r)$ should be rather low. Accumulative and critical attributes are represented with respectively two or three attributes in the new fact table. The reason for the choice of the attribute type depends on different properties of the original attribute. A general example is presented which is used throughout the whole article.

The compression is based on differential compression with some slightly modifications, for example are all values in the delta pattern relative to the maximum value of the compressed attributes. The compression is achieved through the re-use of patterns, that is we expect the different chunks of data have the same pattern.

The method is implemented by adding a new table to the existing schema called the delta table which holds n values. The fact table is modified so that is corresponds to the chosen attribute types. The compression is applied to the fact table and the dimension tables are left alone.

In the article we show that the model can actually be used in a real-life scenario, as we have had a co-operation with Nordjyllands Trafikselsskab which has provided us with real PubTrans data where we show that the compressed warehouse only occupies 40% of the standard warehouse.

A general decompression technique is also described which is basically just a view that recreates the original data warehouse. The execution speed on this view is almost always larger than on the standard warehouse.

Therefore we have shown some general rewrite rules for queries on the standard warehouse to queries on the compressed warehouse which should make the queries run faster than on the view. These rules are based on syntactical substitution where different parts of the SQL statement are gradually replaced by similar statements to the compressed warehouse. These rules can be implemented as a compiler so that the compression becomes transparent to the user.

In the last part of the article we describe how we have tested the performance of the model. The performance is measured both storage and speed wise. The tests have been performed on different n sizes in order to test where the best performance is achieved. We show that on some queries the execution speed is actually faster than on the standard warehouse. The tests also show that some queries to the compressed warehouse are slower.

In conclusion we have shown that space can almost always be saved and in some cases even better performance can be achieved. In this article we have focused solely on data warehouses, but nothing prevents applying this model to other databases.

Jens Frøkjær	*Palle Bertram Hansen*
Ivan Vigsø Sand Larsen	*Tom Oddershede*