Practical approach to selecting data warehouse views using data dependencies

Gillian Dobbie and Tok Wang Ling

Department of Computer Science, National University of Singapore, Singapore
{dobbie,lingtw}@comp.nus.edu.sg

Abstract. Data warehouses integrate information from heterogeneous sources and enable efficient analysis of the information. The two main characteristics of data warehouses are the huge volumes of data they store and the requirement of fast access to the data. Because of the huge volumes of data, simple search techniques are not sufficient. Materialized views in data warehouses are typically complicated, based on many tables, often containing summarized information, but are very important for improving access to the data. Because data warehouses are expected to contain current information, it is also important that the data warehouse, and the views, can be easily updated periodically. The selection of materialized views changes over time, with new materialized views created and old ones dropped. So, the selection of materialized views is crucial. Most research to date has treated the selection of materialized views as an optimization problem with respect to the cost of view maintenance and/or with respect to the cost of queries. In this paper, we consider practical aspects of data warehousing. We identify problems with the star and snowflake schema and suggest solutions, that we call enhanced star schema and enhanced snowflake schema. We also identify practical problems that may arise during view design and suggest heuristics based on data dependencies that can be used to measure if one set of views is better than another set of views, or used to improve a set of views with respect to speed of access.

1 Introduction

A data warehouse stores huge volumes of data that has been gathered from one or more sources for the purpose of efficiently processing decision support or online analytic processing (OLAP) queries. These queries typically find patterns, information, or trends in the data. The main requirements of a data warehouse are to store huge volumes of data and to provide fast access to the data. Like in traditional database systems, frequently asked queries or subparts of frequently asked queries may be precomputed and stored in tables to provide faster access. These stored tables that contain precomputed information are called materialized views. Obviously there are many possible views that could be materialized, and the selection of which views to materialize in order to improve the speed of access is crucial. A bad selection can even slow the speed of access. Views in data warehouses are usually more complicated than in traditional database systems, typically based on many tables and including aggregation or summarization of the underlying data in the data warehouse. Data within the data warehouse changes with time, usually with new data being added periodically. Maintenance of the views as new data is added is a very real problem. So, the selection of which views to materialize, depends both on the speed of access to the data and the ease of maintenance of the views.

Most research to date has treated the selection of materialized views as an optimization problem with respect to the cost of view maintenance and/or with respect to the cost of queries [1, 3, 4, 9, 11, 12]. Each paper proposes an algorithm designed within the framework of general query and maintenance cost models without considering the physical properties of the actual data. In this paper, we identify practical problems that may arise during view design and suggest heuristics based on data dependencies that can be used to measure if one view is better than another or used to improve a set of views.

Our work is related to physical database design and materialized view design in the relational database area. The differences between the data warehouse and relational database areas that effect the selection of materialized views include:

- typically there is a lot more data stored in a data warehouse than in a transactional database,
- data is a data warehouse is non-volatile,
- updates in a data warehouse are derived from additions of new data to the underlying source rather than changes to existing data,
- data warehouse views are likely to be more complicated, containing a lot of aggregation and summarization of the underlying data, and
- queries in a data warehouse analyze a large data set rather than ask for information about a particular data point. For example, a typical data warehouse query would be to plot the sales for employee 5 by month over 1999. A typical database query would ask how many widgets did employee 5 sell in May 1997.

The paper is organized as follows. Section 2 provides background information relevant to the rest of the paper and presents the enhanced star and snowflake schema. There is a sample data warehouse in Section 3 that is used in the following sections. Section 4 outlines problems that can arise due to view design, while heuristics for good design are outlined in Section 5, and demonstrated in Section 6. In Section 7, we compare our work with others and we conclude in Section 8.

2 Background

In this section, we introduce concepts, show inadequacies of the star and snowflake schema, and present the enhanced star schema and enhanced snowflake schema.

2.1 Star and Snowflake schema

The schema of a data warehouse that is built on top of a relational database is typically organized as a *star* or *snowflake* schema [6]. A *star schema* consists of a fact table, and a table for each dimension in the fact table. A star schema can be represented using an entity relationship diagram as shown in Figure 1. The fact table represents a relationship set between two or more dimension entities and has some measurement attributes (*salesQty* in Figure 1). Each dimension table represents an entity with a key and other single valued attributes.

Example 1. Consider a data warehouse that stores information about employees, products and the quantity of each product sold by each employee. There would be a dimension table for employee and another for product storing the employee and product details respectively. The fact table would contain the key of the employee dimension table, the key of the product dimension table and the quantity of a product sold by an employee. A dimension table can contain a dimension hierarchy, e.g. if each product belongs to a category and each category has many products, then we say there is a product dimension hierarchy. □

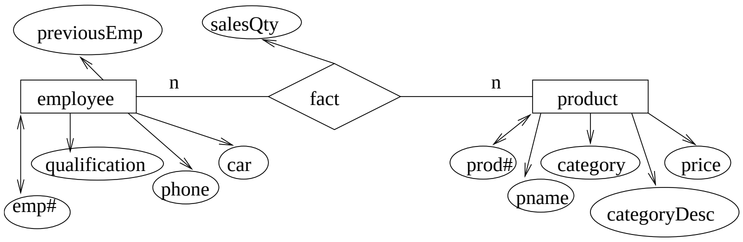

Fig. 1. ER diagram of typical star schema

A *snowflake schema* is like a star schema except it represents the dimensional hierarchies directly, normalizing the dimension tables. A snowflake schema can be represented using an entity relationship diagram as shown in Figure 2. The fact table is the same as the fact table in the star schema and the dimension tables are normalized.

Example 2. Consider the product dimension that has attributes product number, product name, category, categoryDesc and price. The functional dependencies $category \rightarrow categoryDesc$ and $prod\# \rightarrow \{pname, category, price\}$ exist. Using a star schema, the attributes are stored in one table, and the category and categoryDesc are repeated for every product in a particular category. Using a snowflake schema, the hierarchy is represented directly with a table *product(prod#, pname, category, price)* and another table *productCategory(category, categoryDesc)*. □

2.2 Enhanced Star and Snowflake schema

While the star and snowflake schemas described above are convenient, they are overly simplistic. In practice, not all attributes in a dimension table are single valued, not all keys in dimension tables consist of just one attribute, and the relationships between the levels in a dimension hierarchy may be m-to-n (rather

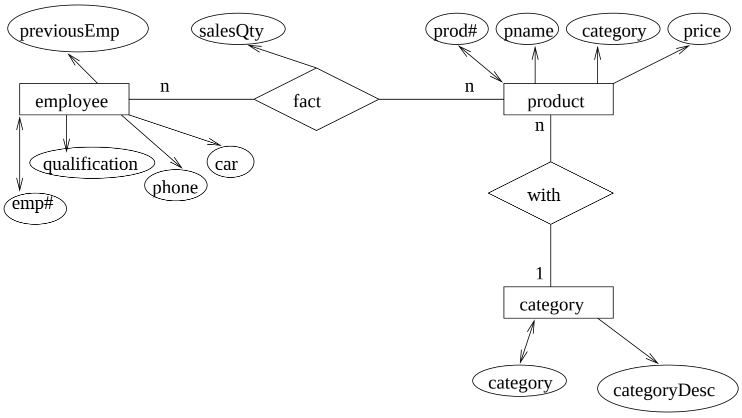

Fig. 2. ER diagram of typical snowflake schema

than 1-to-n). Consider the schema in Figure 3. Each employee may have more than one qualification, more than one phone and more than one car. That is the attributes in the employee dimension table are more likely to be multi-valued attributes and then the key is a composite key. Although we don't represent it in Figure 3, a product could also belong to more than one category. For example, a *health food drink* may belong to category *health food* and category *beverage*. The following functional dependencies hold on the schema in Figure 3:

$$\begin{array}{ll} \{emp\#,\ prod\#\} \rightarrow salesQty & emp\# \twoheadrightarrow car \\ emp\# \twoheadrightarrow qualification & prod\# \rightarrow \{pname,\ category,\ price\} \\ emp\# \twoheadrightarrow previousEmp & category \rightarrow categoryDesc. \\ emp\# \twoheadrightarrow phone & \end{array}$$

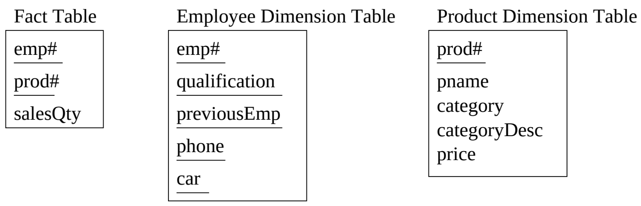

Fig. 3. Sample enhanced star schema

While the snowflake schema removes the redundancy in the dimension hierarchy, the other problems we have described are not alleviated.

An alternative organization would be to store the composite key of the employee dimension table in the fact table. The schema of the resulting fact table is $FactTable(emp\#,\ qualification,\ previousEmp,\ phone,\ car,\ prod\#,\ salesQty)$. This design is also not satisfactory. Not only are we duplicating all the employee information, we are also confusing the relationship between $emp\#$, $prod\#$, and $salesQty$.

In summary, the star and snowflake schemas that are proposed in many papers and books are overly simplistic and not practical for real world applications. Similarly, any view design heuristics that are based on the same assumptions will be overly simplistic.

2.3 Strong and weak functional dependencies

We introduce strong and weak functional dependencies [7] here and use them later in the paper. *Strong* and *weak* functional dependencies extend classical functional dependencies and when used in database design can provide better schemas than those provided using classical functional dependencies.

Strong functional dependency: Let $X \rightarrow Y$ be a functional dependency such that for each $z \in Y$, $X \rightarrow z$ is a full functional dependency. Then $X \rightarrow Y$ is a *strong functional dependency* if all the attributes in Y will not be updated, or if the update need not be performed at real-time or on-line and such updates seldom occur.

Example 3. A person's name seldom changes, so we can say there is a strong functional dependency between $employee\_number$ and $employee\_name$. We write this as $employee\_number \overset{S}{\rightarrow} employee\_name$. □

Weak functional dependency: Let X and Y be subsets of a table R, such that there is a non-trivial multi-valued dependency $X \twoheadrightarrow Y$ in R. If most of the X values are associated with a unique Y-value in R, except for the occasional X-value that may be associated with more than one Y-value, we say Y is *weakly functionally dependent* on X, and write $X \overset{W}{\rightarrow} Y$.

Example 4. Assume that typically an employee lists only one phone number, and very occasionally an employee lists more than one phone number, then we can say that $employee\_number \overset{W}{\rightarrow} employee\_phone$. □

3 Motivating Example

In this section we introduce a populated sample data warehouse that we use in later sections of this paper. While the volume of data in this sample data warehouse is unrealistic, the relationships between the fields are realistic and are used to motivate our work.

Fact table

emp#	prod#	salesQty
1	10	10
2	10	15
3	10	35
1	20	8
2	20	14
3	20	30
1	30	6
2	30	10
3	30	25

Employee

emp#	qualification	previousEmp	phone	car
1	HSC	Bob's fruit shop	8725911	DX6195
1	BSc	Bob's fruit shop	8725911	DX6195
1	MSc	Bob's fruit shop	8725911	DX6195
2	HSC	Quality Groceries	8741834	LX5255
2	BSc	Quality Groceries	8741834	LX5255
2	HSC	Shop and Save	8741834	LX5255
2	BSc	Shop and Save	8741834	LX5255
3	HSC	Shop and Save	7794580	MM5735
3	HSC	Shop and Save	6741856	MM5735

Product

prod#	pname	category	categoryDesc	price
10	weetbix	cereal	A cereal is typically eaten for breakfast. Cereals can be stored on shelves and have a shelf life of 3 months. ...	2
20	vegemite	spread	A spread is eaten with bread. Speads can be stored on shelves and have a shelf life of 6 months. ...	4
30	apple	fruit	Fruit (excluding bananas) must be stored in a partially refrigerated part of the shop. Fruit has a shelf life of 1 week to 1 month. ...	1
40	orange	fruit	Fruit (excluding bananas) must be stored in a partially refrigerated part of the shop. Fruit has a shelf life of 1 week to 1 month. ...	2

Fig. 4. Example data warehouse

The *Fact table* stores the quantity of each product sold by each employee. For example, the first tuple listed in the Fact table states that the employee with *emp#* 1 sold 10 of the product with *prod#* 10. The *Employee* table contains, for each employee, their employee number, qualifications, previous employers, phone numbers and car registration. Each employee may have more than one qualification, more than one previous employer, more than one phone number and more than one car. However, it is uncommon for an employee to have more than one telephone number or more than one car. The *Product* table contains the product number, name, the category the product belongs to, a text description of the category, and the price of the product. Each product belongs to one category. We expect the fact table to be modified often, the price to be modified sometimes and other attributes to change less frequently. The following dependencies hold on the data:

$\{emp\#, prod\#\} \rightarrow salesQty \qquad prod\# \rightarrow price$

$emp\# \twoheadrightarrow qualification$

$emp\# \twoheadrightarrow previousEmp$

$emp\# \xrightarrow{W} phone$

$emp\# \xrightarrow{W} car$

$prod\# \xrightarrow{S} pname$

$prod\# \xrightarrow{S} category$

$category \xrightarrow{S} categoryDesc.$

The key of the *Fact table* is {*emp#, prod#*}. The key of the *Employee* table is {*emp#, qualification, previousEmp, phone, car*}. The key of the *Product* table is {*prod#*}. Access patterns are also important when views are selected. The monthly reports include:

Q1. total quantity of items sold by each employee,
Q2. total quantity of items sold by product, listing both product number and product name,
Q3. total quantity of items sold by category, listing both category and category description,
Q4. total sales of items sold by employee,
Q5. total sales of items sold by product, listing both product number and product name,
Q6. total sales of items sold by category, listing both category and category description,
Q7. average number of items for all products sold by each employee,
Q8. average number of items for all employees by each product, and
Q9. details of each product, and employee number and phone number of the top performing employee for each product.

Less frequently the following information is required:

Q10. A manager is interested in the correlation between the average number of products sold by employees and the average number of qualifications of the employees.
Q11. To get a feel for what to look for when hiring people, a manager may want to know who the previous employers are of his three top sales staff.

4 Problems

In Section 5, we introduce heuristics that can be used for two related but different purposes, to judge if one set of views is better than another, and to improve a set of views. In this section, we compare different views based on the example in Section 3, and give reasons why one set of views is better than the other.

4.1 Frequently changing data

Strong functional dependencies provide us with information about whether an attribute is likely to change or not. In traditional databases, if there is a relation $R(A,\ B,\ C)$ where $A \rightarrow B$ and $B \rightarrow C$ then it is suggested that the relation is decomposed into $R_1(A,\ B)$ and $R_2(B,\ C)$. The reason being that when C is updated, it will need to be updated in many places if the R schema is used, but only once if the R_1 and R_2 schemas are used. Consider the scenario where the values of C are very unlikely to change, (i.e. $B \overset{S}{\rightarrow} C$) then there is no need to decompose R. The same reasoning can be followed when selecting data warehouse views.

Example 5. Consider the view $V(prod\#, pname,\ category,\ categoryDesc)$. Because $prod\# \rightarrow \{pname,\ category\}$ and $category \rightarrow\ categoryDesc$, with traditional database design, V would be decomposed. However, $categoryDesc$ is unlikely to be updated (i.e. $category \overset{S}{\rightarrow}\ categoryDesc$) so from the perspective of redundancy and updatability, there is no advantage in decomposing view V. □

4.2 Aggregates based on aggregates

Often aggregates are based on other aggregates, e.g. a total can be a sum of sums, an average is a sum divided by a count, total sales amount is the sum of items × price per item. If the underlying aggregate on which the new aggregate is based changes frequently, then you not only have to update the aggregate with each change, you also have to update the aggregate of the aggregate. We introduce a heuristic in Section 5 that suggests that such an aggregate is not materialized.

Example 6. Let $\Sigma salesQty_{prod\#}$ be the sum of the $salesQty$ grouped by $prod\#$ and $\mathbf{av}salesQty_{prod\#}$ be the $\Sigma salesQty_{prod\#}$ divided by the number of employees that have sold the product. Because $salesQty$ changes frequently, $\Sigma salesQty_{prod\#}$ will also change frequently, and so will $\mathbf{av}salesQty_{prod\#}$. When the underlying data is changing often, you don't want to have to update too many levels of aggregation. As a heuristic, we would suggest that one level of aggregation is OK and that the second level should be calculated at the time of the query. Store the view $V(prod\#,\ \Sigma salesQty_{prod\#},\ \#OfEmp)$, where $\#OfEmp$ is the number of employees selling product $prod\#$, and recalculate $\mathbf{av}salesQty_{prod\#}$ when it is needed. □

4.3 Usually single valued

In a data warehouse view an attribute may be multi-valued. For example, an employee may have many qualifications. In this case, in order to make the view more maintainable, the multi-valued attribute should be materialized separately from other single valued attributes that relate to the employee. However, if the multi-valued attribute usually has only one value, like an employee usually having only one phone number, then this information can be stored as though it is single valued with an extra (or overflow) table for the occasional extra phone number.

Example 7. Consider the scenario where a person may have more than one phone number and more than one car registration. That is $emp\# \twoheadrightarrow phone$ and $emp\# \twoheadrightarrow car\#$. Usually such a multivalued dependency would be modeled as $V_1(emp\#,\ phone)$ and $V_2(emp\#,\ car)$. Consider now, the scenario where although people may have many phones and many cars, most people usually have only one phone and one car. That is $emp\# \stackrel{W}{\rightarrow} phone$ and $emp\# \stackrel{W}{\rightarrow} car\#$. Access time will be decreased if the attributes are stored in one view $V(emp\#,\ phone,\ car)$ and any multiple values are stored in overflow views $V_1(emp\#,\ phone)$ and $V_2(emp\#,\ car)$. Consider a view that contains the two tuples with $emp\#$ 3 from the Employee dimension table in Section 3, the populated views would be $V(3,\ 7794580,\ MM5735)$ and $V_1(3,\ 6741856)$. □

Methods for maintaining the overflow tables are described in [7].

4.4 Usually multi-valued

In the previous section we considered an example where multi-valued attributes seldom have more than one value. In this section we consider the case where multi-valued attributes usually have many values. When designing traditional database systems, one of the overriding aims is to be to reduce the number of joins when answering queries, however Example 8 illustrates that there are occasions when selecting views for data warehouses where this is not the best approach. Example 8 demonstrates a view where two attributes are independent and frequently have multiple values.

Example 8. Consider view $V(emp\#,\ previousEmp,\ prod\#,\ pname,\ salesQty)$ where an employee has on average 10 previous employers and sells 100 products. To find the previous employer of an employee, it is necessary to access 1000 tuples on average. If instead, the view is decomposed into $V_1(emp\#,\ previousEmp)$ and $V_2(emp\#,\ prod\#,\ pname,\ sales)$, it'd be necessary to access only 10 tuples on average to find the previous employers of an employee. On the other hand, view V would be preferable if there were few previous employees, few products and $previousEmp$ and $pname$ are frequently accessed together. □

4.5 Anomalous data

If there is an attribute in a table that you will aggregate on, then this table can only be joined with other tables with the same key, otherwise anomalous data will be introduced and the following aggregation will be incorrect. This usually arises when you are joining a fact table (or a summarization of a fact table) and a dimension table. The fact table contains a measurement attribute that is likely to be aggregated in the future.

Example 9. Consider creating a view $V(emp\#,\ prod\#,\ qualification,\ salesQty)$ for employee with employee number, 1. Using the sample data warehouse in Section 3, the view would contain the following information:

emp#	prod#	qualification	salesQty
1	10	HSC	10
1	20	HSC	8
1	30	HSC	6
1	10	BSc	10
1	20	BSc	8
1	30	BSc	6
1	10	MSc	10
1	20	MSc	8
1	30	MSc	6

The total sales calculated from view V is 72 whereas the total sales for *emp#* 1 calculated from the Fact table is 24. The problem is that the *salesQty* has been replicated for each *qualification* of the employee. □

4.6 Horizontal versus Vertical Partitioning

The size of attributes in a view should be taken into account when selecting views. If there are some attributes that are small and some that are very large, then when the smaller attributes are accessed, the whole tuple including the very large attributes must be read into temporary storage. A better design is to partition the large attributes from the other attributes, using vertical partitioning. Another related scenario where vertical partitioning is effective is where a view has many attributes and those attributes are never accessed together. The access speed is negatively affected by the extra attributes in the view.

Example 10. Consider a view $V(prod\#,\ categoryDesc,\ \Sigma salesQty_{prod\#})$ where *categoryDesc* is a very large attribute containing a text description of the category of the product, and $\Sigma salesQty_{prod\#}$ is the sum of *salesQty* grouped by *prod#*. If we frequently access only attribute $\Sigma salesQty_{prod\#}$, then including *categoryDesc* in the view slows down access. It is better to partition the view vertically into $V_1(prod\#,\ categoryDesc)$ and $V_2(prod\#,\ \Sigma salesQty_{prod\#})$. □

If a view is very large and queries are often based on particular values for an attribute, the table can be horizontally partitioned on that attribute to improve performance. For example, consider a large view with an attribute *emp#*, where access frequently involves individual employees, then performance will be improved if the view is horizontally partitioned based on the *emp#*.

4.7 Surrogate key

If there is a set of views where one table has a large key and that key is used as a foreign key in another view then a surrogate key can be introduced and the surrogate key can replace the foreign key.

Example 11. Let $\mathbf{av}salesQty_{category}$ be the average sales for the products in each category. Consider a materialized view $V_1(categoryDesc,\ \mathbf{av}salesQty_{category})$ with *categoryDesc* as key, and a view $V_2(prod\#,\ categoryDesc)$ where *categoryDesc* is a foreign key. The view V_2 is large because the foreign key is large. The view V_1 is replaced by $V_1'(c\_id,\ categoryDesc,\ \mathbf{av}salesQty_{category})$ and the surrogate key in V_1' , c_id, can be used as the foreign key in V_2, such that V_2 is replaced by $V_2'(prod\#,\ c\_id)$. □

4.8 Frequency of key values

Consider joining (part of) a fact table and (part of) a dimension table to form a view. If on average the key of the dimension table occurs infrequently in the fact table, then (part of) the dimension table can be joined to (part of) the fact table to form a view, otherwise it isn't worthwhile forming a view.

Example 12. Consider the view $V(emp\#,\ prod\#,\ pname,\ category,\ price)$ formed by joining part of the fact table with part of the product table. View V can be formed if on average each $prod\#$ value occurs infrequently in the fact table. If on the other hand, on average each $prod\#$ value occurs frequently then forming view V wastes a lot of space and the product information is harder to maintain, so no view should be formed.

4.9 Aggregate tables

The way aggregates are stored for a dimension hierarchy is dependent on the frequency of updates, selectivity of attributes, and the access profile as demonstrated in Example 13. Recall that when you are decomposing a table while normalizing a traditional database, it is important that no information is lost. When you are selecting views for data warehouses, this constraint no longer applies because the underlying tables remain.

Example 13. Let $\Sigma salesQty_{prod\#}$ be the sum of $salesQty$ grouped by $prod\#$, and $\Sigma salesQty_{category}$ be the sum of $\Sigma salesQty_{prod\#}$ grouped by category. Consider the view $V(prod\#,\ pname,\ category,\ \Sigma salesQty_{prod\#})$. An alternative set of views that stores salesQty grouped both by $prod\#$ and $category$ is $V_1(prod\#,\ pname,\ \Sigma salesQty_{prod\#})$ and $V_2(category,\ \Sigma salesQty_{category})$. Another alternative set of views is $V_3(prod\#,\ pname,\ category,\ \Sigma salesQty_{prod\#})$ and $V_4(category,\ \Sigma salesQty_{category})$, duplicating category. The view V is preferable over the other sets of views under the following conditions.

- View V is preferable if there are a small number of product names ($pname$) per category, because decomposing V to the other alternatives would in the worst case double the space that is used to store the views without providing any real gains.
- View V is preferable if there are a large number of different categories, for the same reason as the above condition. Namely, a lot of space is used with little gain.
- View V is preferable if the number of sales is updated often. If either of the alternative sets of views are used, when the sales quantity is updated, the aggregation of the sales quantity must also be recalculated.

- View V is preferable if there are very few queries asking for sales by category, because only then is it worthwhile computing the aggregation at the time of the query.
- Otherwise, one of the other sets of views is preferable.

The set of views with V_1 and V_2 is preferable if there are few queries involving the relationship between *pname* and *category* otherwise the set of views with V_3 and V_4 is preferable. □

5 Design Heuristics

The following heuristics can be used to avoid the problems described in Section 4. The aims of the heuristics are to make maintenance easier, and to reduce the access cost, without introducing anomalous data. Traditionally in database systems, the ease of maintenance would be increased by ensuring that each view is in some normal form (like 3NF). In [7], the authors introduce relaxed, replicated, and relaxed-replicated normal forms that improve access speed without introducing new redundancies. The heuristics in Section 5.1 are based on these new normal forms. We use the weak and strong functional dependencies introduced in Section 2.

Traditionally in database systems the query cost is reduced by choices made during physical database design. The heuristics in Section 5.2 are based on physical database design principles, like those in [2], and adapted for views in data warehouses.

It is important that no anomalous data is introduced when views are selected. The heuristics in Section 5.3 guard against this. The anomalous data problem arises where there is aggregation or summarization of the underlying data, and so views in data warehouses in particular are susceptible to this problem.

The selection of views to materialize are different for data warehouses than they are for traditional databases because data warehouses store huge volumes of data, there are different access patterns and the views are typically more complex.

5.1 Reduce access cost using data dependencies

To use the following heuristics, it is necessary to know the data dependencies that exist within the data. The heuristics in this section eliminate the problems described in Sections 4.1, 4.2, 4.3 and 4.4. Heuristics (a1) and (b1) relate to the problem in Section 4.1, heuristic (b2) relates to Section 4.3, heuristic (b3) to Section 4.2, and heuristics (a2) and (b4) to Section 4.4.

OK Rule

a1. The view $V(A,\ B,\ C,\ D)$ is OK if
   - $A \rightarrow \{B,\ C,\ D\}$, or
   - $A \rightarrow \{B,\ D\}$ and $B \xrightarrow{S} C$.

a2. The view $V(A,\ B,\ C)$ is OK, if there is a non-trivial multi-valued dependency $A \twoheadrightarrow B$, on average there are not many values for B and C, and the attributes B and C are frequently accessed together.

Not OK Rule

b1. If $A \rightarrow \{B,\ D\}$ and $B \rightarrow C$ but $B \stackrel{S}{\not\rightarrow} C$, then the view $V(A,\ B,\ C,\ D)$ should be decomposed into $V_1(A,\ B,\ D)$ and $V_2(B,\ C)$.

b2. If there is a non-trivial multi-valued dependency $A \twoheadrightarrow B$ and $A \stackrel{W}{\rightarrow} B$ and $A \stackrel{W}{\rightarrow} C$, the view $V(A,\ B,\ C)$ is replaced by $V_1(A,\ B,\ C)$, $V_{bOverflow}(A,\ B)$ and $V_{cOverflow}(A,\ C)$.

b3. Let there be a fact table, R(A,E,F) where $\{A, E\} \rightarrow F$, and $\{A, E\} \stackrel{S}{\not\rightarrow} F$. Let B be the aggregation of F grouped on A (i.e. ΣF_A) and C be an aggregate that is computed using B (e.g. sum of B, average of B), then view $V(A,\ B,\ C)$ should be replaced by $V_1(A,\ B,\ D)$, where D is another attribute that is needed when computing C from B (see Example 10).

b4. If there is a non trivial multi-valued dependency $A \twoheadrightarrow B$ and $A \stackrel{W}{\not\rightarrow} B$, on average there are many values for B and C, and the attributes B and C are not frequently accessed together the view $V(A,\ B,\ C)$ should be decomposed into $V_1(A,\ B)$ and $V_2(A,\ C)$.

5.2 Reduce access cost using physical database principles

To use the following heuristics, it is necessary to know the approximate size of attributes and the expected access patterns. The heuristics in this section eliminate the problems described in Sections 4.6, 4.7, 4.8 and 4.9. Heuristics (c1), (d1), (d2) and (d6) relate to the problem in Section 4.6, heuristics (c2) and (d3) to Section 4.7, heuristics (c3) and (d4) to Section 4.8, and heuristics (c4) and (d5) to Section 4.9.

OK Rule

c1. A view $V(A, B, C)$ is OK if $A \rightarrow \{B, C\}$ and each of the attributes is a "reasonable" size. It is up to the designer to judge what a reasonable size is.

c2. A view is OK, if it has a large key but the key is not used as a foreign key in another view.

c3. A view that is created by joining (part of) a fact table with (part of) a dimension table is OK if
  - the key of the dimension table is part of the key of the fact table, and
  - each value of the key of the dimension table occurs infrequently, on average, in the fact table.

c4. Consider a view $V(A,\ B,\ C)$ where A and B are two levels in a dimension hierarchy and C is a measurement for A. For example, A could be a product number, B a category and C the quantity of the product sold. The view V is OK if

- there are a small number of As per Bs,
- the attribute C is updated often, or
- there are few queries that aggregate C grouping by B.

Not OK Rule

d1. If there is a view $V(A,\ B,\ C)$ where $A \rightarrow \{B, C\}$ and attribute C is very large then decompose the view into $V_1(A,\ B)$ and $V_2(A,\ C)$.

d2. If there is a view $V(A,\ B,\ C)$ where $A \rightarrow \{B, C\}$ and queries are frequently asked for a particular value of attribute B then V can be horizontally partitioned on B.

d3. If the key in a view V_1 is large and the key is being used as a foreign key in another view V_2 then introduce a surrogate key into V_1 and replace the foreign key in V_2 by the surrogate key.

d4. A view $V(A,\ B,\ C,\ E,\ F,\ G)$ that is created by joining (part of) a fact table $R(A,\ B,\ C)$ (with key $\{A,\ B\}$) with (part of) a dimension table $R_1(B,\ E,\ F,\ G)$ should not be formed if each value of the key of the dimension table occurs frequently, on average, in the fact table.

d5. Consider a view $V(A,\ B,\ C)$ where A and B are two levels in a dimension hierarchy and C is a measurement of A. The view V should be decomposed into $V_1(A,\ C)$ and $V_2(B, \Sigma C_B)$ if
   - there are not many different values for attribute B,
   - the attribute C is not updated often,
   - there are few queries that include the relationship between A and B,
   - there are frequent queries that aggregate C grouping by B.

   If the above properties hold but there are frequent queries that include the relationship between A and B then replace V with $V_3(A, B, C)$ and $V_4(B, \Sigma C_B)$.

d6. If there is a view $V(A,\ B,\ C,\ D,\ E,\ F)$ with key A and there are frequent queries that access $A,\ B,\ C$ and frequent queries that access $D,\ E,\ F$ then vertically partition V into $V_1(A,\ B,\ C)$ and $V_2(A,\ D,\ E,\ F)$.

5.3 Do not introduce anomalous data

Views are built from underlying tables using select, project, join, and aggregation operations. As we demonstrated in Example 9 anomalous data may be introduced if there is a measurement that is likely to be aggregated or a measurement that is aggregated, and the keys of the underlying tables are not the same. The heuristics in this section eliminate the problems described in Section 4.5.

Let there be a table $R_1(A, B, C)$ and another table $R_2(A, D, E)$. We write $\mathcal{F}C_A$ to denote that function $\mathcal{F}$ is likely to be or has been applied to attribute C after grouping on A.

OK Rule

e1. It is OK to create a view as the result of joining a temporary table $R_T(A, \mathcal{F}C_A)$ to any table that has key A.

Not OK Rule

f1. If there is a table (or temporary table) $R_T(A, \mathcal{F}C_A)$ and a table $R_2(A, D, E)$ with a composite key e.g. $\{A, D\}$ then the tables should not be joined.

Example 14. Consider a fact table $F(A,\ B,\ C)$ with key $\{A,\ B\}$ and a dimension table $D(A,\ E,\ F)$ with key $\{A,\ E\}$. Anomalous data will be introduced if the view $V(A,\ E,\ \Sigma C_A)$ is created. The view can be decomposed to $V_1(A,\ E)$ and $V_2(A,\ \Sigma C_A)$. □

5.4 Notes

In summary, we note the following points:

- The replication of data can increase access speed but makes maintenance more difficult. This is only a problem if the data is updated frequently or if an attribute is an aggregation of data that is updated frequently. We distinguish between attributes that are updated frequently and attributes that are seldom updated. Our heuristics attempt to balance speed of access against updatability of attributes.
- We have made estimates e.g. the most appropriate level of aggregation, that should further be supported by experimentation.
- Small changes to the data warehouse schema or to access patterns will not effect the selection of materialized views very much. If there are large changes to either the data warehouse schema or the access patterns then a new set of views must be selected. These could be selected from scratch or the heuristics can be used to find which set of views is better than the existing ones given the new situation.
- This is an obvious approach for selecting views in a data warehouse. It is based on knowledge that the data warehouse designers already have and the heuristics are easy to apply, as we will demonstrate in the following section.

6 Demonstrating the heuristics

In this section we demonstrate how the design heuristics described in Section 5 are used to select views for the example data warehouse in Section 3.

We follow three steps to select views:

S1. We consider the typical access patterns and select the attributes that we require in the set of views. We do this by creating a view for each query.
S2. We then eliminate redundancy by joining related views. This minimizes the space that will be needed to store the materialized views and makes updating the views less error prone. The groupings can be formed by grouping all attributes that belong to each entity together.
S3. Finally select the best set of views by decomposing the views found in the previous step, using the heuristics in Section 5.

6.1 Select attributes

In considering only the access patterns, we create a view for each of the queries presented in Section 3. For simplicity we use the query number (from Section 3) as the name of the matching view.

$Q1(\underline{emp\#}, \ \Sigma salesQty_{emp\#})$
$Q2(\underline{prod\#}, \ pname, \ \Sigma salesQty_{prod\#})$
$Q3(\underline{category}, \ categoryDesc, \ \Sigma salesQty_{category})$
$Q4(\underline{emp\#}, \ \Sigma(price \ \times \ salesQty)_{emp\#})$
$Q5(\underline{prod\#}, \ pname, \ \Sigma(price \ \times \ salesQty)_{prod\#})$
$Q6(\underline{category}, \ categoryDesc, \ \Sigma(price \ \times \ salesQty)_{category})$
$Q7(\underline{emp\#}, \ \mathbf{av} salesQty_{emp\#})$
$Q8(\underline{prod\#}, \ pname, \ \mathbf{av} salesQty_{prod\#})$
$Q9(\underline{prod\#, \ emp\#, \ phone}, \ pname, \ category, \ categoryDesc, \ price)$
$Q10(\underline{emp\#}, \ \Sigma salesQty_{emp\#}, \ \mathbf{count} qualification_{emp\#})$
$Q11(\underline{emp\#, \ previousEmp})$

Both $Q9$ and $Q11$ involve selecting top performing employees. These queries could present a case for horizontally partitioning the data on top performing employees per product and the 3 top performing employees overall, respectively (see heuristic d2) but because the tuples in the partition would change often, we do not perform the partitioning.

6.2 Eliminate redundancy

In considering redundancy, we join related views together. The groupings are formed by grouping all attributes that belong to each entity together. For example, we group all attributes that belong to the *employee* entity in one view.

The resulting views are

$VT_{emp\#}(\underline{emp\#, \ previousEmp, \ phone}, \ \Sigma salesQty_{emp\#}, \mathbf{av} salesQty_{emp\#}, \ \Sigma(price \ \times \ salesQty)_{emp\#}, \ \mathbf{count} qualification_{emp\#})$,
formed from $Q_1 \ \bowtie \ Q4 \ \bowtie \ Q_7 \ \bowtie \ \pi_{emp\#, \ phone}(Q_9) \ \bowtie \ Q_{10} \ \bowtie \ Q_{11}$,
$VT_{prod\#}(\underline{prod\#}, \ pname, \ category, \ \Sigma salesQty_{prod\#}, \ \Sigma(price \times salesQty)_{prod\#}, \ \mathbf{av} salesQty_{prod\#}, \ price)$,
formed from $Q_2 \ \bowtie \ Q_5 \ \bowtie \ Q_8 \ \bowtie \ \pi_{prod\#, \ pname, \ category, \ price}(Q9)$,
$VT_{category}(\underline{category}, \ \Sigma salesQty_{category}, \ \Sigma(price \ \times \ salesQty)_{category}, \ categoryDesc)$,
formed from $Q_3 \ \bowtie \ Q_6 \ \bowtie \ \pi_{category, \ categoryDesc}(Q9)$.

6.3 Use heuristics

Now we use the heuristics to improve the views produced in Section 6.2. We consider $VT_{emp\#}$, $VT_{prod\#}$ and $VT_{category}$, and then the resulting views together. Consider Figure 5 that shows the decomposition of $VT_{emp\#}$. Heuristics (b2), (b3) and (b4) are used in the decomposition. The resulting views are $V_{overflow}$, V_3 and V_4. Consider Figure 6 that shows the decomposition of $VT_{prod\#}$. The resulting

$VT_{emp\#}$ $(\underline{emp\#,\ phone,\ previousEmp},\ \Sigma salesQty_{emp\#},\ \Sigma(price \times salesQty)_{emp\#},$
$\mathbf{av}\,salesQty_{emp\#},\ \mathbf{count}\,qualification_{emp\#})$

$\Downarrow$ b2 ($emp\# \xrightarrow{W} phone$)
$VT_{emp\#}$ is replaced by $V_{overflow}$ and $VT_{emp\#}$

$V_{overflow}(\underline{emp\#,\ phone})$
$VT_{emp\#}(\underline{emp\#,\ previousEmp},\ phone,\ \Sigma salesQty_{emp\#},\ \Sigma(price \times salesQty)_{emp\#},$
$\mathbf{av}\,salesQty_{emp\#},\ \mathbf{count}\,qualification_{emp\#})$

$\Downarrow$ b3 (aggregates based on $\Sigma salesQty_{emp\#}$)
$VT_{emp\#}$ is replaced by VT_2

$V_{overflow}(\underline{emp\#,\ phone})$
$VT_2(\underline{emp\#,\ previousEmp},\ phone,\ \Sigma salesQty_{emp\#},\ \mathbf{count}\,qualification_{emp\#})$

$\Downarrow$ b4 ($emp\# \twoheadrightarrow previousEmp$)
VT_2 is replaced by V_3 and V_4

$V_{overflow}(\underline{emp\#,\ phone})$
$V_3(\underline{emp\#,\ previousEmp})$
$V_4(\underline{emp\#},\ phone,\ \Sigma salesQty_{emp\#},\ \mathbf{count}\,qualification_{emp\#})$

Fig. 5. Views resulting from $VT_{emp\#}$

view is VT_5. Consider Figure 7 that shows the decomposition of $VT_{category}$. The resulting views are V_7 and VT_8. Next we consider the resulting views together.

$VT_{prod\#}$ $(\underline{prod\#},\ pname,\ category,\ \Sigma salesQty_{prod\#},\ \Sigma(price \times salesQty)_{prod\#},$
$\mathbf{av}\,salesQty_{prod\#},\ price)$

$\Downarrow$ b3 (aggregates based on $\Sigma salesQty_{prod\#}$)
$VT_{prod\#}$ is replaced by VT_5

$VT_5(\underline{prod\#},\ pname,\ category,\ \Sigma salesQty_{prod\#},\ price)$

Fig. 6. Views resulting from $VT_{prod\#}$

Using heuristic c4, we consider the views that store the product-category dimension hierarchy. See Figure 8. In this case, the overriding factor is that salesQty is updated often so we chose to replace VT_5 and VT_8 by V_9. Based on the other heuristics, this set of views we have selected are the best for this schema, with the given access pattern. In summary, the resulting views using the heuristics are $V_{overflow}$, V_3, V_4, V_7 and V_9.

$VT_{category}$ ($\underline{category}$, $\Sigma salesQty_{category}$, $\Sigma(price \times salesQty)_{category}$, categoryDesc)

$\Downarrow$ b3 (aggregates based on $\Sigma salesQty_{category}$)
$VT_{category}$ is replaced by VT_6

$VT_6(\underline{category}$, $categoryDesc$, $\Sigma salesQty_{category})$

$\Downarrow$ d1 ($categoryDesc$ is large field)
VT_6 is replaced by V_7 and VT_8

$V_7(\underline{category}$, $categoryDesc)$
$VT_8(\underline{category}$, $\Sigma salesQty_{category})$.

Fig. 7. Views resulting from $VT_{category}$

$VT_5(\underline{prod\#}$, $pname$, $category$, $\Sigma salesQty_{prod\#}$, $price)$
$VT_8(\underline{category}$, $\Sigma salesQty_{category})$

$\Downarrow$ c4 ($salesQty$ updated often)
VT_5 and VT_8 are replaced by V_9

$V_9(\underline{prod\#, category}$, $pname$, $\Sigma salesQty_{prod\#}$, $price)$.

Fig. 8. Views resulting from VT_5 and VT_8

7 Related work

Materialized views in traditional database systems have been widely researched, both from the selection and maintenance perspectives. There are different demands on data warehouses, and these demands influence the selection of materialized views for data warehouses. There is an excellent survey covering the maintenance of materialized views in [5], with a subsection serving the research in maintaining materialized views in data warehouses. But there has been only a little interest in how the views are selected in data warehouses.

Most research to date has treated the selection of materialized views as an optimization problem with respect to the cost of view maintenance and/or with respect to the cost of queries using general cost models [1, 3, 4, 9, 11, 12]. Our work is very different from this kind of research. We have taken a very practical approach to selecting views for data warehouses, taking the physical properties of the data into account.

The authors of [8] suggest that functional dependencies could be used when selecting views for data warehouses but they do not suggest how functional dependencies could be used. They suggest the development of a new type of functional dependency, called an aggregate functional dependency. In contrast, we have shown how existing data dependencies and heuristics from physical database design can be used in view design. Our work provides a basis for view design with other functional dependencies, like aggregate functional dependencies.

8 Conclusions

A data warehouse stores data from heterogeneous sources and enables efficient analysis of the data. Materialized views can improve access speeds to the large volumes of data stored. However, the selection of views is crucial. To date, most of the research in the area uses general cost models to find the optimal or near optimal set of views, without considering the physical properties of the data.

The star and snowflake schemas, that are widely recognized as the way to model the data in a data warehouse, are based on overly simplistic assumptions. It is difficult to model realistic data warehouses using these schema. For example, it is assumed that the attributes in a dimension table are single-valued, the keys of the dimension table are single attributes, and that the relationship between levels of the dimension hierarchy are 1-to-n. This is often not the case in practice.

In this paper, we discuss problems that may arise when selecting which views to materialize for a data warehouse. Our main contribution is a set of heuristics that can be used to improve a set of materialized views, or judge if one set of views is better than another set of views. The heuristics are based on physical properties of the data such as how likely it is that the value of an attribute will change, how often a multi-valued attribute will have more than one value, access patterns, selectivity of an attribute, and size of an attribute. We have provided a foundation on which further practical work, involving the selection of materialized views in data warehouses, can be based.

There are many directions we could take from here. We list only some of them below. We will start by formalizing the heuristics presented and further investigate the heuristics. We need to find answers to the following questions: Does one heuristic make something that was OK, not OK or vice versa? Do any of the rules conflict? Is the result different if the rules are applied in a different order? We believe that some of the heuristics could be replaced by first ensuring that the views are in relax-replicated 4NF (see [7]). Following this line of investigation would make the distinction between traditional database design and data warehouse view design more obvious. Investigating this distinction then leads on to investigating the kinds of indexes that are best for materialized views in data warehouses, from a practical perspective.

The database group at Stanford University consider a data warehouse itself as a materialized view of the underlying data sources. We believe that if we take this approach, the heuristics presented in this paper could also be used in the design of data warehouses. We intend to investigate this approach to data warehouse design.

Another question that naturally follows from this work is whether view maintenance algorithms can be improved by taking the physical properties of the data into account. This question was listed among the open problems in [5], where they suggested that functional dependencies and other constraints may be used.

We propose that the properties identified in this paper may also be used to improve view maintenance algorithms.

References

1. Elena Baralis, Stefano Paraboschi and Ernest Teniente. Materialized Views Selection in a Multidimensional Database. In *Proceedings of 23rd International Conference on Very Large Data Bases (VLDB'97)*, 1997.
2. Rob Gillette, Dean Muench and Jean Tabaka. *Physical Database Design for SYBASE SQL Server*. Prentice Hall PTR, 1995.
3. Himanshu Gupta. Selection of Views to Materialize in a Data Warehouse. In *6th International Conference on Database Theory (ICDT)*, 1997, pages 98–112, Springer-Verlag LNCS 1186.
4. Himanshu Gupta and Inderpal Singh Mumick. Selection of Views to Materialize Under a Maintenance Cost Constraint. In *Database Theory - ICDT '99, 7th International Conference on Database Theory (ICDT)*, 1999, pages 453–470, Springer-Verlag LNCS 1540.
5. Ashish Gupta and Inderpal Singh Mumick. Maintenance of Materialized Views: Problems, Techniques, and Applications. In *Data Engineering Bulletin*, 18(2), pages 3–18, 1995.
6. R. Kimball. *The data warehouse toolkit*. John Wiley and Sons, 1996.
7. Tok Wang Ling, Cheng Hian Goh and Mong Li Lee. Extending Classical Functional Dependencies for Physical Database Design. In *Information and Software Technology*, pages 601-608, vol 38 (1996), Elsevier Science.
8. M. Mohania, K. Karlapalem and Y. Kambayashi. Data Warehouse Design and Maintenance through View Normalization. In *Proceedings of the 10th International Database and Expert Systems Applications Conference, DEXA '99*, pages 747–750. Springer-Verlag LNCS 1677.
9. Kenneth A. Ross, Divesh Srivastava and S. Sudarshan. Materialized View Maintenance and Integrity Constraint Checking: Trading Space for Time. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 447-458.
10. Amit Shukla, Prasad Deshpande and Jeffrey F. Naughton. Materialized View Selection for Multidimensional Datasets. In *Proceedings of 24th International Conference on Very Large Data Bases, (VLDB'98)*, 1998, pages 488-499.
11. Dimitri Theodoratos, Spyros Ligoudistianos, and Timos Sellis. Designing the Global Data Warehouse with SPJ Views. In *Proceedings of 11th International Conference on Advanced Information Systems Engineering, (CAiSE'99)*, 1999, Springer-Verlag LNCS 1626.
12. Jian Yang, Kamalakar Karlapalem and Qing Li. Algorithms for Materialized View Design in Data Warehousing Environment. In *Proceedings of 23rd International Conference on Very Large Data Bases, (VLDB'97)*, 1997, pages 136-145.