Minimizing Detail Data in Data Warehouses

M. O. Akinde, O. G. Jensen, and M. H. Böhlen *

Department of Computer Science, Aalborg University,
Frederik Bajers Vej 7E, DK–9220 Aalborg Øst, Denmark,
e-mail: <strategy | guttorm | boehlen>@@cs.auc.dk

Abstract. Data warehouses collect and maintain large amounts of data from several distributed and heterogeneous data sources. Because of security reasons, operational requirements, and technical feasibility it is often impossible for data warehouses to access the data sources directly. Instead data warehouses have to replicate legacy information as detail data in order to be able to maintain their summary data.
In this paper we investigate how to minimize the amount of detail data stored in a data warehouse. More specifically, we identify the minimal amount of data that has to be replicated in order to maintain, either incrementally or by recomputation, summary data defined in terms of *generalized project-select-join* (GPSJ) views. We show how to minimize the number of tuples and attributes in the current detail tables and even aggregate them where possible. The amount of data to be stored in current detail tables is minimized by exploiting smart duplicate compression in addition to local and join reductions. We identify situations where it becomes possible to omit the typically huge fact table and prove that these techniques in concert ensure that the current detail data is minimal in the sense that no subset of it permits to accurately maintain the same summary data. Finally, we sketch how existing maintenance methods can be adapted to use the minimal detail tables we propose.

1 Introduction

A data warehouse materializes summarized data in order to provide fast access to data integrated from several distributed and heterogeneous data sources [10]. At an abstract level, summarized data can be considered as materialized views over base tables in the data sources. Particularly important are materialized views that involve aggregation because data warehouse clients often, if not always, require summarized data [12, 15].

When source data is modified, summarized data must be updated eventually to reflect the changes. This is accomplished either by recomputing summary data or by incrementally maintaining it. While incrementally maintaining summary data is substantially cheaper than recomputing it this is not always possible [6, 13]. In addition to the performance penalty recomputation imposes, data

* This research was supported in part by the Danish Technical Research Council through grant 9700780 and Nykredit, Inc.

warehouses have to face a potentially worse problem. It is hardly ever possible to directly access the data sources and thus the base tables needed to recompute summary data. In other words, recomputation might not even be feasible because of the lack of base data. The typical solution is to maintain detail data in data warehouses, which accounts for or at least aggravates the storage problems of data warehouses [11, 12].

Figure 1 illustrates the structure of a data warehouse and its interaction with the operational data store [10]. We assume that the current detail data mirrors the (current) data of the data sources. Summarized data aggregates current (and old) detail data in terms of GPSJ views. GPSJ views are project-select-join views extended with aggregation and grouping. They represent the single most important class of SQL statements used in data warehousing [12, p.14].

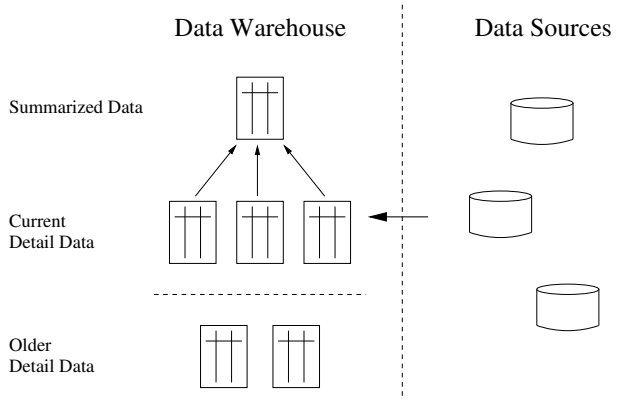

Fig. 1. The Basic Data Warehouse Framework

This paper addresses the problem of minimizing the amount of current detail data such that the summarized data (materialized GPSJ views) is still accurately maintainable. The set of current detail tables is minimal in the sense that only tables needed for maintaining the summarized data are included and no extraneous tuples or attributes are included in these tables; i.e. the summarized data and the set of minimal detail data is self-maintainable.

Given a GPSJ view, we apply local and join reductions to derive the minimal set of detail data that is sufficient to maintain the view. We show how smart duplicate compression has to be exploited to minimize the amount of detail data. More precisely, we compress duplicates by aggregating the current detail data, while making sure they keep being sufficient to maintain the summary table. For star schemas this technique results in enormous storage space savings as illustrated below.

Each base table referenced in a materialized GPSJ view is associated with an auxiliary view. However, sometimes an auxiliary view is not needed to maintain the materialized view, and thus does not need to be stored in the data warehouse.

We give the conditions for eliminating an auxiliary view, and as a side-effect show how key constraints and group-by attributes can be utilized to reduce the cost of incremental maintenance of the summary data.

1.1 Motivating Example

As a running example, we consider a data warehouse of retail sales for a grocery chain. The sales data is stored in the `sale` fact table with the schema

```
sale(ID, timeID, productID, storeID, price).
```

The warehouse also stores the following three dimension tables, which contain information about dates, products, and stores, respectively.

```
time(ID, day, month, year)
product(ID, brand, category)
store(ID, street_address, city, country, manager)
```

We assume referential integrity constraints from (1) `sale.productID` to `product.ID`, (2) from `sale.timeID` to `time.ID`, and (3) from `sale.storeID` to `store.ID`. As an example of a GPSJ view consider the following query, which, for each month, retrieves the total number of products sold by the grocery chain along with the total price of these sales and the number of different brands sold.

```
CREATE VIEW product_sales AS
SELECT time.month, SUM(price) AS TotalPrice, COUNT(*) AS TotalCount,
       COUNT(DISTINCT brand) AS DifferentBrands
FROM sale, time, product
WHERE time.year = 1997 AND
      sale.timeID = time.ID AND
      sale.productID = product.ID
GROUP BY time.month
```

The current detail data necessary to maintain the `product_sales` view consists of three auxiliary views.

```
CREATE VIEW timeDTL AS
SELECT ID, month
FROM time
WHERE year = 1997
```

```
CREATE VIEW productDTL AS
SELECT ID, brand
FROM product
```

```
CREATE VIEW saleDTL AS
SELECT timeID, productID, SUM(price) AS SalePrice,
       COUNT(*) AS SaleCount
FROM sale
WHERE timeID IN (SELECT ID FROM timeDTL) AND
      productID IN (SELECT ID FROM productDTL)
GROUP BY timeID, productID
```

The product_sales view can now be reconstructed from these three auxiliary views without ever accessing the original fact and dimension tables.

```
CREATE VIEW product_sales AS
SELECT timeDTL.month, SUM(saleDTL.SalePrice) AS TotalPrice,
       SUM(SaleCount) AS TotalCount,
       COUNT(DISTINCT productDTL.brand) AS DifferentBrands
FROM saleDTL, timeDTL, productDTL
WHERE saleDTL.timeID = timeDTL.ID AND
      saleDTL.productID = productDTL.ID
GROUP BY timeDTL.month
```

In order to illustrate the savings in terms of storage space we compare the size of auxiliary views to the size of the original fact and dimension tables. Consider the following numbers based on real-life case studies of data warehouses [12, p.46-47,62]. We only give the numbers for the fact table because the size of dimension tables is insignificant in comparison to the fact table.

```
Time dimension: 2 years × 365 days = 730 days.
Store dimension: 300 stores, reporting sales each day.
Product dimension: 30,000 products in each store, 3,000 sell each
     day in a given store.
Transactions per product: 20

Number of tuples in fact table: 730 × 300 × 3000 × 20
     = 13,140,000,000
Fact table size: 13,140,000,000 × 5 fields × 4 bytes = 245 GBytes
```

The product_sales view only retrieves products from 1997, so the time dimension is halved assuming an even distribution of product sales over the two years. Further, the store dimension can be ignored, as it is not referenced in the view. If all 30,000 different products in the grocery chain are sold each day, which is the worst case for smart duplicate compression, then the auxiliary view saleDTL has the following size.

```
Number of tuples in the auxiliary view: 365 × 30000 = 10,950,000
Auxiliary view size: 10,950,000 × 4 fields × 4 bytes = 167 MBytes
```

In other words, in order to maintain the product_sales summary table we can reduce the size of the fact table a data warehouse has to maintain from 245 GBytes to 167 MBytes.

1.2 Related Work

Many incremental view maintenance algorithms have been developed, but most of them have been developed for traditional, centralized database environments where the view maintenance system is assumed to have full control and access to the database [3, 6–8, 15]. Segev et al. [16–18] study materialized views in distributed systems. However, they only consider views over a single base table.

Maintenance algorithms for views with aggregation have received little attention only [6, 7, 15]. A recent paper [13] proposes a method for efficiently maintaining materialized views with aggregation and how to maintain a large set of summary tables defined over the same base tables. Common to all these approaches is the fact that, in the general case, they have to resort to recomputations and queries involving base tables. These may not always be accessible, as with the case of legacy systems or highly secure data. In addition, [13] requires the presence of referential integrity constraints between base tables.

It is possible to avoid recomputation by making views self-maintainable [2, 5]. Determining the minimum amount of extra information required to make a given view self-maintainable is an open research problem [19]. Hull and Zhou [9] make views self-maintainable by pushing down projections and selections to the base tables and storing these at the data warehouse. Quass et al. [14] present an algorithm for making views self-maintainable using key and referential integrity constraints. However, their algorithm is limited to handling *project-select-join* (PSJ) views; that is, views consisting of a projection followed by a selection followed by a cross-product over a set of base tables.

In this paper, we extend the framework in [14] to handle *generalized project-select-join* (GPSJ) views, that is, views consisting of a generalized projection, i.e., a projection enhanced with aggregation and grouping, followed by a selection followed by a cross-product over a set of base tables. We consider all the SQL aggregates, as well as the use of the `DISTINCT` keyword. In addition to the use of key and referential integrity constraints, we show how the duplicate-eliminating property of generalized projection and group-by attributes can be exploited to minimize the amount of data stored in the data warehouse.

1.3 Paper Outline

The paper proceeds as follows. Section 2 presents notation and assumptions along with a brief explanation of the basic framework for deriving auxiliary views. Section 3 presents an algorithm for deriving a set of auxiliary views that is sufficient to maintain GPSJ views. Conclusions are given in Section 4.

2 Preliminaries

In Section 2.1 we present the notation used in this paper along with some basic assumptions. Section 2.2 introduces the framework for deriving auxiliary views.

2.1 Notation and Assumptions

We denote the *materialized GPSJ view* by V, its set of *auxiliary views* by $\mathcal{X}$, and the set of *base tables* referenced in V by $\mathcal{R} = \{R_1, R_2, \ldots, R_n\}$. We assume that no base table contains null values. For simplicity, we also assume that each base table R_i contains a single attribute key.

We use the *generalized projection* operator, Π_A, to represent aggregation [4]. Generalized projection is an extension of duplicate-eliminating projection, where the schema A can include both aggregates and regular attributes. For simplicity, all aggregates are assumed to be on single attributes. An *aggregate* $f(a_i)$ denotes the application of f to the attribute a_i, where $f \in \{\texttt{MIN,MAX,COUNT,SUM,AVG}\}$. An expression list is a list of attributes and/or aggregates. Regular attributes in A become *group-by attributes* for the aggregates and are denoted $GB(A)$. We say that that an attribute is *preserved* in V if it appears in A, either as a regular attribute or in an aggregate.

GPSJ views are relational algebra expressions of the form $V = \Pi_A \sigma_S (R_1 \bowtie_{C_1} R_2 \bowtie_{C_2} \ldots \bowtie_{C_{n-1}} R_n)$ where A is the schema of V, S is the set of conjunctive selection conditions on $\mathcal{R}$, and C_1 to C_{n-1} are join conditions. C_i is the join condition $R_i.b = R_j.a$, where a is a key of R_j. While we limit ourselves to views joining on keys in this paper, our techniques can be extented to a broader class of GPSJ views.

We assume insertions, deletions and updates of base tables. We say that a base table R_i has *exposed updates* if updates can change values of attributes involved in selection or join conditions. Exposed updates are propagated as deletions followed by insertions. We assume that no superfluous aggregates [1] are used in the view.

2.2 Framework for Deriving Auxiliary Views

Given a view V defined over a set of base tables $\mathcal{R}$ we want to derive a set of auxiliary views $\mathcal{X}$ such that $\{V\} \cup \mathcal{X}$ is self-maintainable (i.e., can be maintained upon changes to $\mathcal{R}$ without requiring access to the base tables). The set of base tables $\mathcal{R}$ constitutes one such set of auxiliary views. It is possible to reduce the size of this set of auxiliary views using *local* and *join reductions* on $\mathcal{R}$.

Local Reductions. Local reductions result from pushing down projections and local conditions (i.e., selection conditions involving attributes from a single table as opposed to those involving attributes from different tables, which are called join conditions) to each base table $R_i \in \mathcal{R}$ [9, 14].

Projections for GPSJ views store only those attributes of R_i which are preserved in V or are involved in join conditions. Note that, unlike for PSJ views, we do not explicitly require the storage of keys in order to maintain the auxiliary views.

We refer to the use of projections and local conditions to reduce the number of attributes and tuples in an auxiliary view as local reductions.

Join Reductions. Join reductions result from exploiting key and referential integrity constraints [14]. Consider a join $R_i \bowtie_{R_i.b=R_j.a} R_j$, where $R_j.a$ is the

[1] An aggregate function $f(a_i)$ is superfluous, if it can be replaced by a_i without changing the semantics of the statement.

key of R_j and there is a referential integrity constraint from $R_i.b$ to $R_j.a$. This means that each tuple in R_i joins with exactly one tuple in R_j, and that there is no tuple in R_i not joinable with a tuple in R_j. As a result insertions to R_j can never join with already existing tuples in R_i.

We say that R_i *depends* on R_j if V contains a join condition $R_i.b = R_j.a$, where a is a key of R_j, referential integrity exists between R_i and R_j, and R_j does not have exposed updates [14]. Join reductions limit the number of tuples in the auxiliary view X_{R_i} to those that can join with tuples in other auxiliary views of $\mathcal{X}$. More precisely, if R_i depends on R_j we perform a semijoin, i.e., a join reduction, of X_{R_j} on R_i.

Recollect that exposed updates change the values of attributes involved in local or join conditions. If we attempted the join reduction $X_{R_i} = R_i \ltimes X_{R_j}$ in the presence of exposed updates on R_j, then it would be possible for an updated tuple in R_j to join with an existing tuple in R_i, where the old value did not pass local selection conditions and hence was not in X_{R_j}.

3 Deriving Auxiliary Views

In this section we present an algorithm that, given a GPSJ view V, derives a set of self-maintainable auxiliary views sufficient to maintain V. In Section 3.1 we examine how different types of aggregates in a GPSJ view can influence the amount of auxiliary data needed to maintain the view, and we define the concept of a completely self-maintainable aggregate set. Section 3.2 describes how auxiliary views are minimized, and explains how duplicate compression can be used to further reduce auxiliary views. In Section 3.3 we present the conditions for omitting an auxiliary view. The algorithm for deriving auxiliary views is found in Section 3.4.

3.1 Classification of Aggregates

Aggregates can be divided into two classes depending on whether or not they are *self-maintainable*. An aggregate $f(a_i)$, where a_i is an attribute of the base table R_i, is self-maintainable, if $f(a_i)$ can be incrementally maintained upon changes to R_i. If $f(a_i)$ is not incrementally maintainable, then it must be recomputed from the base table upon changes to R_i, and, consequently, an auxiliary view for R_i is required. Thus, $f(a_i)$ is said to be a *self-maintainable aggregate* (SMA), if the new value of $f(a_i)$ can be computed solely from the old value of the aggregate and from the change to the base table R_i. Furthermore, a *self-maintainable aggregate set* (SMAS) is a set of aggregates, where the new values of the aggregates can be computed solely from the old values of the aggregates and from the changes to the base tables. The five SQL aggregates are categorized in Table 3.1 [13].

Aggregates, which are SMASs with respect to both insertions and deletions do not require auxiliary data to be maintained upon any change to their respective base tables. Therefore we define a *completely self-maintainable aggregate set* (CSMAS) as:

Table 1. Classification of SQL aggregates. An aggregate can be a SMA (or SMAS) with respect to insertion($\triangle$) and deletion(∇) respectively

Aggregate	SMA	SMAS
COUNT	$\triangle/\nabla$	$\triangle/\nabla$
SUM	$\triangle$	$\triangle/\nabla$, if COUNT is included
AVG	Not a SMA	$\triangle/\nabla$, if COUNT and SUM are included
MAX/MIN	$\triangle$	$\triangle$

Definition 1 (Completely self-maintainable aggregate set). *A set of aggregates is completely self-maintainable if the new values of the aggregates can be computed solely from the old values of the aggregates and from the changes to the base tables in response to both insertions and deletions.*

Table 2 categorizes each of the five SQL aggregates as either CSMASs or non-CSMASs. Note that if the `DISTINCT` keyword is used, then the aggregate is always a non-CSMAS and is not replaced. This is so because an aggregate must be distributive[2] or be replaceable by distributive aggregates in order to be self-maintainable, and the `DISTINCT` keyword makes an aggregate non-distributive.

Table 2. Classification of SQL aggregates

Aggregate	Replaced By	Class
COUNT	COUNT(*)	CSMAS
SUM	SUM, COUNT(*)	CSMAS
AVG	SUM, COUNT(*)	CSMAS
MAX/MIN	Not replaced	non-CSMAS

Henceforth we assume that aggregates appearing in view definitions are replaced according to Table 2. Note that because null-values are not considered any `COUNT` can be replaced by a `COUNT(*)`.

3.2 Minimizing Auxiliary Views

The algorithm presented in Section 3.4 derives a set of auxiliary views $\mathcal{X}$, where each auxiliary view $X_{R_i} \in \mathcal{X}$ is an expression on the following form:

[2] Distributive aggregates can be computed by partitioning their input into disjoint sets, aggregating each set individually, and then further aggregating the result from each set into the final result. The SQL aggregates, `COUNT`, `SUM`, `MIN`, and `MAX`, are distributive.

$X_{R_i} = (\Pi_{A_{R_i}} \sigma_S R_i) \ltimes_{C_1} X_{R_{j1}} \ltimes_{C_2} X_{R_{j2}} \ltimes \ldots \ltimes_{C_n} X_{R_{jn}}$, where

A_{R_i} is the expression list defined over the set of attributes in R_i preserved in V or involved in join conditions after applying smart duplicate compression.

S is the local condition on R_i.

$R_{j1}, R_{j2}, \ldots, R_{jn}$ is the set of tables R_i depends on.

C_{j_k} is the join condition $R_i.b = R_{j_k}.a$, where a is the key of R_{j_k} referenced by $R_i.b$.

Each auxiliary view X_{R_i} is a selection and a generalized projection on a base table R_i followed by zero or more semi-joins with other auxiliary views. Local and join reductions are applied as discussed in Section 2.2.

Smart Duplicate Compression. Smart duplicate compression exploits the duplicate-eliminating property of the generalized projection to minimize auxiliary views while ensuring that they are still self-maintainable.

Consider an auxiliary view `sale′` for the `sale` fact table that is required by the `product_sales` view defined in Section 1.1. Assume the `sale′` auxiliary view is a result of local reductions and generalized projection. As duplicate elimination occurs, we are unable to determine the number of tuples in each group, which means that the `product_sales` view can not be maintained. Therefore we include a `COUNT(*)` in the auxiliary view. This results in the `sale′` auxiliary view becoming self-maintainable, as `COUNT(*)` is a CSMAS. Table 3 shows an example instance of the `sale′` auxiliary view.

Table 3. An example instance of the `sale′` auxiliary view after adding `COUNT(*)`

timeID	productID	price	COUNT(*)
1	1	20	15
1	2	40	9
1	2	50	2
2	1	15	5
2	1	25	8

The `product_sales` view retrieves the total sales price, `SUM(sale.price)`. `SUM` is a CSMAS, and all CSMASs can be replaced by a set of distributive aggregates. Distributive aggregates can be computed by partitioning their input into disjoint sets, aggregating each set individually, and then further aggregating the result from each set into the final result. This means that we can aggregate the `sale′` auxiliary view and still maintain the `product_sales` view. Table 4

shows an instance of the `sale′` auxiliary view after local reductions and smart duplicate compression.

Table 4. The `sale′` auxiliary view after smart duplicate compression

timeID	productID	SUM(price)	COUNT(*)
1	1	300	15
1	2	460	11
2	1	275	13

We formalize smart duplicate compression on an auxiliary view X_{R_i} with the schema $A_{X_{R_i}}$ after local reductions in the following steps.

Algorithm 3.1 Smart duplicate compression for auxiliary views.

1. Include a `COUNT(*)` in X_{R_i} unless this is superfluous.
2. For each attribute $a_i \in A_{X_{R_i}}$
   If a_i is not used in non-CSMASs, join conditions, or group-by clauses,
   Then replace all CSMASs over a_i by the appropriate set of aggregates in Table 2.

Note that if an aggregate is superfluous, there is no need to replace the attribute with it. An example of this occurs when an auxiliary view includes the key of its base table, and results in the auxiliary view degenerating into a PSJ view.

Maintainance Issues under Duplicate Compression. The theory of deriving maintenance expressions for GPSJ views in general is beyond the scope of this paper and the interested reader is referred to [15] instead. However, the presence of compressed duplicates requires changes to the maintenance of the views which we discuss below. If aggregates are not limited to the root table[3] or if non-CSMASs are present in the view, it may become necessary to compute the values of an aggregate from tuples in the auxiliary views.

Recall that CSMASs can be computed by partitioning their inputs into disjoint sets. This means that the value of a CSMASs can be computed from aggregated values in the auxiliary view. Thus we compute a `COUNT(*)` in V by summing up the counts in the auxiliary view of the root table. Similarly, a `SUM` can be recomputed by adding the appropriate aggregates in the auxiliary view.

However, when computing a CSMAS from an attribute in the auxiliary view (as may happen if an attribute of a base table is involved in both a CSMAS and

[3] The root table is the base table at the root of the extended join graph (see Section 3.3). In a star schema, this would be the fact table.

a non-CSMAS or the attribute is involved in a CSMAS not on the root table), it becomes necessary to use the `COUNT(*)` on the root table (referred to as cnt_0) to account for duplicates. Thus, for any aggregate $f(a)$ in A, where a is an attribute in R_i which is not maintained by an aggregate in $\mathcal{X}$, we compute the value of f as $f(a * cnt_0)$.

Assume that we define a view `product_sales_max` as follows:

```
CREATE VIEW product_sales_max AS
SELECT sale.productID, MAX(sale.price) AS MaxPrice, SUM(sale.price)
       AS TotalPrice, COUNT(*) AS TotalCount,
FROM sale
GROUP BY sale.productID
```

The auxiliary view for `product_sales_max` would then be:

```
CREATE VIEW saleDTL AS
SELECT productID, price, COUNT(*) AS SaleCount
FROM sale
GROUP BY productID
```

To recompute the value of `SUM(sales.price)` from `saleDTL`, one could then create the following view:

```
CREATE VIEW product_sales_max AS
SELECT productID, MAX(price) AS MaxPrice, SUM(price*SaleCount) AS
       TotalPrice, SUM(SaleCount) AS TotalCount,
FROM saleDTL
GROUP BY productID
```

Note that this approach only holds true for CSMASs. `MAX` and `MIN` and aggregates using the `DISTINCT` keyword already ignore duplicates and can be recomputed directly from the auxiliary views.

3.3 Eliminating an Auxiliary View

Under certain circumstances an auxiliary view X_{R_i} for a base table R_i is not required for propagating insertions, deletions, and updates to the materialized GPSJ view V or the other auxiliary views. When this happens, it becomes unnecessary to materialize the auxiliary view for that base table.

For PSJ views it is sufficient that R_i transitively depends on all other base tables in $\mathcal{R}$, and that it is not in the $Need$ set of any other base table $R_j \in \mathcal{R}$ for X_{R_i} to be eliminated [14]. Informally, the $Need$ set of a base table R_i is the minimal set of base tables with which R_i must join, so that the unique set of tuples in V associated with any given tuple in R_i can be identified. In general, if R_j is in the $Need$ set of R_i, then X_{R_j} is required to propagate the effect of deletions and protected updates to R_i on V.

Due to the presence of aggregates in GPSJ views, we must also require that attributes in R_i are not involved in non-CSMASs. By definition non-CSMASs cannot be incrementally maintained for both insertions and deletions. As a result,

it may sometimes be necessary to recompute the values of the aggregates from the auxiliary views, thereby preventing the elimination.

Join graphs are used to define the $Need$ functions associated with PSJ views [14]. Due to the complexities added by generalized projection, we extend this concept and define the *extended join graph* $G(V)$.

Definition 2 (Extended Join Graph). *Given a GPSJ view V, the extended join graph $G(V)$ is a directed graph $\langle \mathcal{R}, \varepsilon \rangle$ where $\mathcal{R}$ is the set of base tables referenced in V and forms the vertices of the graph. There exists a directed edge $e(R_i, R_j) \in \varepsilon$ from R_i to R_j if V contains a join condition $R_i.b = R_j.a$ and a is a key of R_j. A vertex $R_i \in \mathcal{R}$ is annotated with g, if R_i contains attributes involved in group-by clauses in V. If one of the attributes is a key of R_i, it is annotated with k instead.*

For the purpose of this paper, we assume that the graph is a tree (i.e., there is at most one edge leading into any vertex and no cycles), and that it has no self-joins. As both star schemas and snowflake structures have tree graphs, this assumption still allows us to handle a broad class of views occurring in practice. The base table at the root of the tree is referred to as the *root table*, R_0. Figure 2 shows the extended join graph for the `product_sales` view.

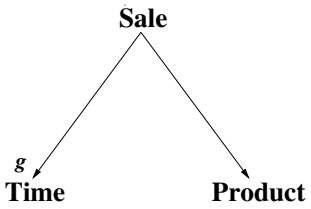

Fig. 2. The extended join graph for the `product_sales` view

The following definition is used to determine the $Need$ set of a base table R_i. Unlike for PSJ views [14], we are not required to join with all other base tables, if the key of R_i is not preserved in V. Instead we use the $Need_0$ function to find a set of group-by attributes that forms a combined key to V.

Definition 3.

$$Need(R_i, G(V)) = \begin{cases} \emptyset & \text{if } R_i \text{ is a vertex annotated with } k, \\ \{R_j\} \cup Need(R_j, G(V)) & \text{if } R_i \text{ is not a vertex annotated with } k \\ & \text{and there exists an } R_j \text{ such that} \\ & e(R_j, R_i) \in G(V) \text{ and } i \neq 0, \\ Need_0(R_0, G(V)) & \text{otherwise.} \end{cases}$$

$Need_0(R_0, G(V))$ performs a depth-first traversal of the extended join graph, in order to find the minimal set of base tables, whose group-by attributes form a

combined key to V. If none of the keys of the base tables are preserved in V, then $Need_0(R_0, G(V))$ includes all base tables, but R_0, containing attributes involved in group-by clauses in V, annotated $\mathcal{R}_g$, and the base tables between the root table and $\mathcal{R}_g$ in the extended join graph. As we require all group-by attributes to be projected in the view, these always form a combined key to the view. However, if a key of a base table R_i is preserved in V, then it is not necessary to include the base tables appearing in the subtree of R_i in $Need_0(R_0, G(V))$. The reason is that each tuple in R_i joins with exactly one tuple in each of the base tables in the subtree of R_i. This means that when we group on the key of R_i, any group-bys on attributes of base tables in the subtree of R_i can not increase the total number of groups in V, and therefore are not needed in the combined key to the view. $Need_0(R_i, G(V))$ is defined as follows:

Definition 4.

$$Need_0(R_i, G(V)) = \begin{cases} \bigcup_{R_j \in G(V)} \{R_j\} \cup Need_0(R_j, G(V)) & \textit{if there exists an edge } e(R_i, R_j) \textit{ and } R_i \textit{ is not} \\ & \textit{vertex annotated with } k \textit{ and there exists an } R_k \\ & \textit{annotated with } k \textit{ or } g \textit{ in the subtree of } R_j, \\ \emptyset & \textit{otherwise.} \end{cases}$$

Note that *Need* functions define the minimal set of tables with which R_i must join to maintain the view. This can be exploited in view maintenance [1].

3.4 Algorithm for Deriving Auxiliary Views

Our algorithm employs the concept of completely self-maintainable aggregates along with local/join reduction and smart duplicate compression to derive the minimum set of auxiliary views necessary to maintain a specified generalized project-select-join view.

Algorithm 3.2 Creation of minimum auxiliary views for generalized project-select-join views.

1. Construct the extended join graph $G(V)$.
2. For each base table $R_i \in \mathcal{R}$ calculate $Need(R_i, G(V))$ and check whether R_i transitively depends on all other base tables in $\mathcal{R}$.
   If this is the case, and R_i is not in the *Need* set of any other base table in $\mathcal{R}$, and none of the attributes of R_i are involved in non-CSMASs, then X_{R_i} can be omitted.
   Else $X_{R_i} = (\Pi_{A_{R_i}} \sigma_S R_i) \ltimes_{C_1} X_{R_{j1}} \ltimes_{C_2} X_{R_{j2}} \ltimes \ldots \ltimes_{C_n} X_{R_{jn}}$, where A_{R_i} is an expression list over the attributes of R_i after local reduction and smart duplicate compression. C_j is the join condition $R_i.b = R_j.a$, where a is a key of R_i and R_i is dependent on $R_{j1}, R_{j2}, \ldots, R_{jn}$.

We state a theorem regarding the correctness and minimality of the auxiliary views derived by the above algorithm.

Theorem 1. *Let V be a view defined by a generalized project-select-join view with a tree-structured join graph. The set of auxiliary views $\mathcal{X}$ derived by Algorithm 3.2 is the unique minimal set of views $\mathcal{X}$ that can be added to V such that $\mathcal{X} \cup \{V\}$ is self-maintainable under insertions, deletions, and updates to the base tables $\mathcal{R}$ referenced in V.*

We say that the auxiliary views $\mathcal{X}$ derived by the algorithm are minimal in the sense that no subset of it permits us to accurately maintain V. By this we mean that neither an auxiliary view nor any attributes or tuples in an auxiliary view can be removed without sacrificing the maintainability of $\mathcal{X} \cup V$. The proof involves three steps. First, we show that each auxiliary view $X_{R_i} \in \mathcal{X}$ is minimal. Secondly, we show that each auxiliary view X_{R_i} is necessary to maintain V. Finally, we prove that $\mathcal{X}$ is self-maintable. The complete proof had to be omitted due to space limitations. It can be found in [1].

4 Conclusions

This paper presents an algorithm for making GPSJ views, the single most important class of views in data warehouses, self-maintainable by materializing a set of auxiliary views such that the original view and the auxiliary views taken together are self-maintainable.

We complete local and join reductions [14] with smart duplicate compression, which is inherent to GPSJ views with aggregation, and show that this allows to significantly reduce the amount of detail data that has to be stored in a data warehouse.

The work suggests several lines of future research.

Our approach could be extended to old detail data. Old detail data is often append-only data. This makes it possible to relax the definition of CSMA because only insertions have to be considered. This implies that old detail data can be reduced even further and it should also be possible to simplify (and speed up) the incremental maintenance.

Another future research direction is the generalization of GPSJ views. While GPSJ views are appropriate for data warehouses (they fit the template structure of data warehouse query tools such as Star Tracker [12, p.321]) it can still be useful to generalize them to include restrictions on groups (the `HAVING` clause in SQL), nested subqueries, and general expressions in the select clause.

An automation of the proposed techniques should start out with the definition of an abstract notation for classes of summary data. Our algorithm should then be extended to determine the minimal set of detail data for classes of summary data.

Finally, it could also be attractive to trade accuracy for space. It is one of our assumptions that summary data shall be maintained accurately. It is interesting to investigate frameworks where summary data is not accurate but a (good) approximation. Such an approach might allow further reductions of the size of the detail data.

References

1. M. O. Akinde, O. G. Jensen, and M. H. Böhlen. Minimizing Detail Data in Data Warehouses. R-98-5002, Aalborg University, 1998.
2. J. A. Blakely, N. Coburn, and P. A. Larson. Updating Derived Relations: Detecting Irrevelant and Autonomously Computable Updates. In *ACM Transactions on Database Systems*, pages 14(3):369–400. Los Alamitos, USA, September 1989.
3. S. Ceri and J. Widom. Deriving Production Rules for Incremental View Maintenance. In *Proceedings of the Seventeenth International Conference on Very Large Databases*, pages 577–589. Barcelona, Spain, September 1991.
4. A. Gupta, V. Harinarayan, and D. Quass. Aggregate-Query Processing in Data Warehousing Environments. In Umeshwar Dayal, Peter M. D. Gray, and Shojiro Nishio, editors, *Proceedings of the Twenty-first International Conference on Very large Databases*. Zurich, Switzerland, September 1995.
5. A. Gupta, H. V. Jagadish, and I. S. Mumick. Data Integration using Self Maintainable Views. Technical report, AT&T Bell Laboratories, November 1994.
6. T. Griffin and L. Libkin. Incremental Maintenance of Views with Duplicates. In M. Carey and D. Schneider, editors, *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 328–339. San Jose, CA, USA, May 1995.
7. A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining Views Incrementally. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 157–166. Washington D.C., USA, May 1993.
8. J. V. Harrison and S. W. Dietrich. Maintenance of Materialized Views in a Deductive Database: An Update Propagation Approach. In *Proceedings of the Sixth International Conference of Data Enginerring*, pages 56–65, 1992.
9. R. Hull and G. Zhou. A Framework for Supporting Data Integration using the Materialized and Virtual Approaches. In *Proceedings of the ACM SIGMOD Conference on Management of Data*. Montreal, Quebec, Canada, June 1996.
10. W. H. Inmon, C. Imhoff, and G. Battas. *Building the Operational Data Store*. John Wiley & Sons, Inc., 1996.
11. W. H. Inmon. *Building the Data Warehouse*. John Wiley & Sons, Inc., 1992.
12. R. Kimball. *The Data Warehouse Toolkit*. John Wiley & Sons, Inc., 1996.
13. I. S. Mumick, D. Quass, and B. S. Mumick. Maintenance of Data Cubes and Summary Tables in a Warehouse. In *Proceedings of the ACM SIGMOD Conference on Management of Data*. Tuscon, Arizona, USA, May 1997.
14. D. Quass, A. Gupta, I. S. Mumick, and J. Widom. Making Views Self-Maintainable for Data Warehousing. In *Proceedings of the Conference on Parallel and Distributed Information Systems*. Miami Beach, Florida, USA, December 1996.
15. D. Quass. Maintenance Expressions for Views with Aggregation. In *ACM Workshop on Materialized Views: Techniques and Applications*. Montreal, Canada, June 1996.
16. A. Segev and W. Fang. Currency-based Updates to Distributed Materialized Views. In *Proceedings of the Sixth International Conference of Data Enginerring*, pages 512–520. Los Alamitos, USA, 1990.
17. A. Segev and W. Fang. Optimal Update Policies for Distributed Materialized Views. In *Management Science*, pages 37(7):851–870, July 1991.
18. A. Segev and J. Park. Updating Distributed Materialized Views. In *IEEE Transactions on Knowledge and Data Engineering*, pages 1(2):173–184, 1989.
19. J. Widom. Research Problems in Data Warehousing. In *Proceedings of the Fourth International Conference on Information and Knowledge Management*, November 1995.