Wright State University
CORE Scholar

Kno.e.sis Publications

The Ohio Center of Excellence in Knowledge-Enabled Computing (Kno.e.sis)

12-1996

Algorithms for Adapting Materialised Views in Data Warehouses

Mukesh Mohania

Guozhu Dong
Wright State University - Main Campus, guozhu.dong@wright.edu

Follow this and additional works at: https://corescholar.libraries.wright.edu/knoesis


Part of the Bioinformatics Commons, Communication Technology and New Media Commons, Databases and Information Systems Commons, OS and Networks Commons, and the Science and Technology Studies Commons

Repository Citation

Mohania, M., & Dong, G. (1996). Algorithms for Adapting Materialised Views in Data Warehouses. *Proceedings of the International Symposium on Cooperative Database Systems for Advanced Applications, Kyoto, Japan Dec. 5-7, 1996*, 62-69.
https://corescholar.libraries.wright.edu/knoesis/346

This Article is brought to you for free and open access by the The Ohio Center of Excellence in Knowledge-Enabled Computing (Kno.e.sis) at CORE Scholar. It has been accepted for inclusion in Kno.e.sis Publications by an authorized administrator of CORE Scholar. For more information, please contact corescholar@www.libraries.wright.edu, library-corescholar@wright.edu.

Algorithms for Adapting Materialised Views in Data Warehouses

Mukesh Mohania* Guozhu Dong†

Abstract

In this paper we consider the problem of materialised view adaptation in data warehouses. Materialised views are important in data warehousing where they are used to speed up query processing on large amounts of data. User requirements change over time, which may change the definitions of views dynamically. For such situations, the question arises whether the materialised views should be recomputed from scratch for every change in the definition or they should be obtained by adapting old materialised views. Changes to a view definition may be expensive, if the view is recomputed from scratch. Therefore, it is worthwhile to examine ways of performing changes to the materialised view without recomputing the entire view which has undergone a change in definition. We present adaptation algorithms for adapting views when the changes are made in each SELECT, FROM, and WHERE clause. The main idea of our algorithms is to augment the schemas of base relations by adding 'join-count' attributes to them, and augment the schemas of views by keeping 'derive-count' attributes on them as extra information.

1 Introduction

Materialised views are required in data warehouses when rapid access of derived data is needed or recomputing of the entire view from scratch is expensive. User requirements change over time, which may change the definitions of views dynamically in data warehouses. For such situations, the question arises whether the materialised views should be recomputed from scratch for every change in the definition or they should be obtained by adapting old materialised views. Recomputing views from scratch would be prohibitive, especially when the definitions of views involve base data from multiple sites. Therefore, it would be beneficial to adapt views so that the amount of information which needs to be communicated between the sites can be minimised while recomputing the materialised views. Thus, a good solution of view adaptation problem is important because of the need of views in data warehouses.

In this paper we consider the problem of materialised view adaptation in data warehouses, which is a variant of the view maintenance problem [1, 2, 8, 12]. (In the view maintenance problem, materialised views are maintained incrementally whenever there are changes in the base data. In the view adaptation problem, old materialised views are adapted, as far as possible, in order to obtain new materialised views when old views are redefined.) We assume that a data warehouse is implemented as a distributed database system and a data warehouse itself can use a database management system, where the underlying data sources and the warehouse subscribe to a single relational data model. We assume that duplicates are not retained in the materialised views and the redefinition of a view can be expressed as a sequence of local changes. We propose view adaptation algorithms for changes in views defined by Select-Project-Join (SPJ) queries. Our primary objective is to minimise the total communication cost while adapting views. The idea behind these algorithms is to avoid sending data that is irrelevant to the final value of the view. It is achieved by associating a 'join-count' attribute for each join on base relations and a 'derive-count' attribute on each view as extra information. The 'join-count' on a base relation indicates how many times a tuple joins with tuples of the other relation. A tuple with a 'join-count' zero means that it cannot join with any tuple in the other relation. To illustrate this, let the schema of R be AB, and the schema of S be BC. Let $\langle a, b\rangle$ be a tuple in R and its 'join-count' value be 4. It means there are 4 tuples in S whose B-value is b. The 'derive-count' on a view indicates the number of derivations of each view tuple. By associating this extra information to base relations and views, which takes very little extra space and can be maintained ef-

*Advanced Computing Research Centre, School of Computer and Information Science, University of South Australia, The Levels 5095, Adelaide, Australia, Email: mohania@cis.unisa.edu.au. Part work done while working at the University of Melbourne and supported by the Australian research Council through research grants.

†Dept. of Computer Science, University of Melbourne, Parkville 3052, Australia, Email: dong@cs.mu.oz.au

ficiently, the total communication cost can be reduced significantly.

Gupta et. al. [9] present view adaptation methods for changes to SPJ queries in centralised databases. These methods handle only a limited number of changes, and do not consider communication cost, which becomes an important issue when data warehouse systems are considered. The view adaptation problem is related to the problem of answering queries using materialised views [11, 3]. The query can be considered to be a redefinition of the view and it may be recomputed by changing the materialization of the view. However in the query approach it is assumed that the old view must remain in storage, while in view adaptation, the old view is replaced with the new. Many view maintenance methods have been proposed for centralised databases [2, 5, 6, 8, 7, 10] and for distributed databases [1, 4, 13, 14]. However, the solutions proposed cannot be extended to view adaptation. This is because view maintenance involves only propagation of updates on base relations to views, where as view adaptation involves processing of changes to view definitions, and computation of the resulting changes to view instances. In this paper, we either extend the work of [9] for data warehouses or propose more efficient methods.

The outline of the rest of the paper is as follows. Firstly we introduce some notations, definitions, and our motivating example in section 2. We describe the view adaptation algorithm for changing the SELECT clause in section 3. In 4, we discuss the algorithms for the situation where the new view is obtained from the old view by changing the condition in WHERE clause. Next, in section 5, we discuss the algorithm for adapting the old view when changes are made in the FROM clause. Finally, we draw conclusions from what has been presented and propose topics for further investigation in section 6.

2 Definitions

To deal with the view adaptation problem, we consider SELECT-PROJECT-JOIN views where there are at most two relations in the FROM clause. We call these views as 2-way SPJ views. If there are n relations in the FROM clause, they are called n-way SPJ views. We use R, S and T to denote base relation names and their corresponding current instances. We denote $\bar{A}$, $\bar{B}$ etc for sequences of attributes, $\bar{a}$, $\bar{b}$ etc for the corresponding attribute values. We denote $\mathcal{C}$ and $\mathcal{C}'$ for conditions in the WHERE clause. Let the schema of R be $(\bar{A}\bar{B})$ and the schema of S be $(\bar{B}\bar{C}\bar{D})$. We write t for tuples, and $\langle \bar{a}, \bar{b} \rangle$ for tuples when we need to specify the contents of the tuples. By $t[\bar{A}]$ we mean the attribute values of t for the attributes in $\bar{A}$. Let $p_R = j_R/|R|$, and called the join participation rate, where $j_R = |\{t \in R \mid t[\bar{B}] \in \pi_{\bar{B}}(S)\}|$, p_S for S being similar.

Let V_a be an augmented view, and let V_a^o and V_a^n be the value of view V_a before and after a change in V_a. An augmented view has some extra information in the view V. This extra information consists of either more attributes and/or more tuples in the view. Indeed, the value of the final view can be obtained by taking the selection and/or projection on the augmented view and it is denoted by V'. The process of view adaptation is defined when the extent of the new view can be obtained by the extent of the old view. For example, V_a^o can be adapted provided the extent of V_a^n is obtained from the extent of V_a^o. Thus, the augmented view is useful for adapting the view in response to change in SELECT, PROJECT, and JOIN clauses. When V_a^n is not obtained by V_a^o, but obtained by evaluating the definition of view V, the process is called recomputing view V.

2.1 Example

In this section we give an example to motivate view adaptation methods. We shall use it as a running example.

Example 1 Consider a data warehouse, implemented as a distributed database system, with two sites, namely, *Melbourne site* and *Adelaide site*. At *Melbourne site* there are three relations:

Melb_Com_Director	*Comp*	*Location*	*Director*	*Phone*	
Melb_Manufacture	*Comp*	*P*	*Type*	*Manager*	*Rm#*
Melb_Customer	*Cust*	*P*	*Type*		

At *Adelaide site* there are three relations:

Adel_Com_Director	*Comp*	*Location*	*Director*	*Phone*	
Adel_Manufacture	*Comp*	*P*	*Type*	*Manager*	*Rm#*
Adel_Customer	*Cust*	*P*	*Type*		

Here, the company's director relation stores information regarding "who is the director of a company

and where it is located". The manufacturer relation stores information regarding "who is producing what products", and the customer relation stores information regarding "who is interested in buying what major products". (By major products we mean appliances, cars, building materials and so on.)

A tuple $\langle \bar{c}, \bar{l}, \bar{d}, \bar{v} \rangle$ in the *Melb_Com_Director* relation means that a company $\bar{c}$ is located at location $\bar{l}$ and its director is $\bar{d}$ whose phone number is $\bar{v}$. A tuple $\langle c, p, \tau, m, r \rangle$ in the *Melb_Manufacture* relation means that a Melbourne company c produces product p of type τ in room r and product manager is m. A tuple $\langle c', p', \tau' \rangle$ in the *Melb_Customer* relation means that a Adelaide customer c' intends to buy product p' of type τ'.

To help selling products produced in Melbourne to customers from Adelaide, the following query is frequently posed:

> "List those pairs of companies in Melbourne and customers in Adelaide where the customer wishes to buy products produced by the company and having a type code less than 3."

Because this query is needed frequently, it is profitable to use a corresponding materialised view, called V, at Adelaide data warehouse. It can be written in SQL as follows:

CREATE VIEW V **AS**
SELECT *Comp*, *Cust*
FROM *Melb_Manufacture*, *Adel_Customer*
WHERE *Melb_Manufacture.P* = *Adel_Customer.P* and *Melb_Manufacture.Type* = *Adel_Customer.Type* and *Melb_Manufacture.Type* < '03'.

A small sample state of the relevant relations and the view V is shown below.

Part of the Melbourne Database

Melb_Manufacture

Comp	*P*	*Type*	*Manager*	*Rm#*
c_1	p_1	1	John	L1.2
c_1	p_1	2	Mark	G0.4
c_1	p_2	2	Jim	L2.4
c_3	p_2	1	Lang	L1.3

Part of The Adelaide Database

Adel_Customer

Cust	*P*	*Type*
d_1	p_1	1
d_1	p_2	3
d_2	p_1	1
d_1	p_2	2
d_3	p_2	4
d_3	p_2	2

V

Comp	*Cust*
c_1	d_1
c_1	d_2
c_1	d_3

3 Changing the SELECT Clause

In this section we discuss view adaptation method for the situations where the new view is obtained from the old view by changing the SELECT clause. In [9], the authors have discussed a method of adapting old materialised view in centralised databases when an attribute in the SELECT clause is added in the new definition. Their method is limited to those cases where foreign keys are available in the database schema and they participate in joining the base relations. The main idea in their method is to augment the view with foreign key attributes in order to adapt it. They have considered this problem as an update to the view and proposed an update strategy. Since in their method base relations are joined on key attributes, a subquery (in their update command) returns only one value for each tuple of the old view, which is not true in general. That is, in their method the number of tuples in the old view is equal to the number of tuples in the new view. If base relations are not joined on key attributes, their method is not applicable for view adaptation.

We discuss a view adaptation algorithm for the situation where attributes are added or deleted in the SELECT clause. This algorithm is general for 2-way SPJ queries in that it allows base relations to be joined on any attributes. (Note that this algorithm can also be used for n-way SPJ queries).

3.1 Adaptation Algorithm

Deleting attributes from the SELECT clause is straightforward; the old materialised view V can be adapted by taking the projection. Adding attributes to a view may increase the cardinality of the old view. Therefore, it is important to get all required tuples of new materialised view V' after the adaptation of the old view. There could be many solutions for adapting V to get V', including:

1. Keep more or all attributes of participating relations (in the join) in the old materialised view V as extra information and project out all those attributes whenever they are needed in the V'.

   This solution would be inappropriate when there are large number of attributes in the base relations.

2. Keep join attributes in the old materialised view V as extra information besides the attributes of the view. Also, augment a 'join-count' attribute to each relation for each join, indicating how many times a tuple joins with tuples of other relation. Basically, this extra information helps in getting all relevant tuples of V' while adapting V and do not incur any overheads, except those due to storage.

   We discuss the algorithm based on this idea.

Let $R(\bar{A}\bar{B})$ and $S(\bar{B}\bar{C}\bar{D})$ be stored at different sites. Let the view expression V be defined as

CREATE VIEW V **AS**
SELECT $\bar{A}$, $\bar{C}$
FROM R, S
WHERE $R.\bar{B} = S.\bar{B}$ and $\mathcal{C}$.

Here, values of attributes may or may not be distinct and $\mathcal{C}$ is a condition on either R or S or both.

Let attributes $\bar{D}$ be added in the SELECT clause of V and the new view V' be defined as

CREATE VIEW V' **AS**
SELECT $\bar{A}$, $\bar{C}$, $\bar{D}$
FROM R, S
WHERE $R.\bar{B} = S.\bar{B}$ and $\mathcal{C}$.

To adapt V in order to get V', we keep join attributes $\bar{B}$ in V; however, the schema of the resulting augmented view V_a^o will be $(\bar{A}\bar{B}\bar{C})$. We also augment the schemas of R and S by adding a 'join-count' attribute to each. We define the 'join-count' of R, the 'join-count' of S being likewise. A tuple $\langle \bar{a}, \bar{b} \rangle$ in R will have the 'join-count' value 3 if $\bar{b}$ occurs three times in $\pi_{\bar{B}}(S)$, and the value 0 if there is no occurrence of $\bar{b}$ in $\pi_{\bar{B}}(S)$.

Example 2 The 'join-count' augmented relations of *Melb_Manufacture* and *Adel_Customer* are now shown next.

Melb_Manufacture

Comp	*P*	*Type*	*Manager*	*Rm#*	Count
c_1	p_1	1	John	L1.2	2
c_1	p_1	2	Mark	G0.4	0
c_1	p_2	2	Jim	L2.4	2
c_3	p_2	1	Lang	L1.3	0

Adel_Customer

Cust	*P*	*Type*	Count
d_1	p_1	1	1
d_1	p_2	3	0
d_2	p_1	1	1
d_1	p_2	2	1
d_3	p_2	4	0
d_3	p_2	2	1

In our running example, *Melb_Manufacture* and *Adel_Customer* are joined on the attributes P and $Type$. Therefore, we augment the view V with these attributes, obtaining view V_a^o, as shown below.

V_a^o

Comp	*Cust*	*Manager*	*P*	*Type*
c_1	d_1	John	p_1	1
c_1	d_2	John	p_1	1
c_1	d_1	Jim	p_2	2
c_1	d_3	Jim	p_2	2

We now present an adaptation algorithm to calculate the value of V' by adapting the value of V when both V and V' are at same site. The performance of the algorithm depends on the join participation rate of R. Recall that its value is calculated as $p_R = j_R/|R|$, where $j_R = |\{t \in R \mid t[\bar{B}] \in \pi_{\bar{B}}(S)\}|$. From this formula, we can observe that the value of p_R can be maintained together with the maintenance of the 'join-counts' of tuples in R.

Algorithm 1 Algorithm for changing SELECT clause

1. At the site of S,
   - let $I = \{t \in S |$ the 'join-count' of t in S is $>0\}$;
   - send I to the site of V;

2. At the site of V,
   - perform
     $V_a^n = \pi_{\bar{A}\bar{B}\bar{C}\bar{D}}\sigma_{\mathcal{C}}(V_a^o \bowtie_{V_a^o.\bar{B}=I.\bar{B}, V_a^o.\bar{C}=I.\bar{C}} I)$;
   - perform $V' = \pi_{\bar{A}\bar{C}\bar{D}}(V_a^n)$;

The communication cost of the above algorithm is zero provided V is stored at the site of S, otherwise

$|I|$, which is $p_S * |S|$ under uniform distribution. The communication cost of the computing-from-scratch algorithm (without 'join-count' attribute on base relations) will be $|S|$. Note that when the value of p_S is 1, then the above algorithm behaves as a computing-from-scratch algorithm.

3.2 Example

Example 3 Here we illustrate Algorithm 1 using our running example. Suppose we wish to add the *Manager* attribute to the view V defined in our example. The new view V' is expressed as:

CREATE VIEW V' **AS**
SELECT *Comp*, *Cust*, *Manager*
FROM *Melb_Manufacture*, *Adel_Customer*
WHERE *Melb_Manufacture.P* = *Adel_Customer.P* and *Melb_Manufacture.Type* = *Adel_Customer.Type* and *Melb_Manufacture.Type* < '03'.

The intermediate steps and the value of V' are shown below.

1. Derive I at Melbourne.

I

Comp	*P*	*Type*	*Manager*	*Rm#*
c_1	p_1	1	John	L1.2
c_1	p_2	2	Jim	L2.4

V'

Comp	*Cust*	*Manager*
c_1	d_1	John
c_1	d_2	John
c_1	d_1	Jim
c_1	d_3	Jim

2. Send I to Adelaide, and join I with V_a^o on *Comp*, *P*, *Type* attributes.

3. Project *Comp*, *Cust*, *Manager* attributes on the join of I and V_a^o. The results are shown in V'.

4 Changing the WHERE Clause

In this section we discuss view adaptation methods for the situation where the new view is obtained from the old view by changing the conditions in the WHERE clause.

An adaptation algorithm for changing the WHERE clause in centralised databases has been discussed in [9]. There, the authors assume that the attribute of the condition, which is going to be updated, is either an attribute of the view or of a wider augmented stored view. If that attribute is not in the view, they first add the attribute to the view and then apply their algorithm. Thus, their algorithm is a two phase algorithm. If the change leads to the insertion of tuples to the view, then they compute these tuples by using some rewritings of the view definition; however, for most cases, these rewritings seem to be as expensive as the computing the view from scratch. When the change leads to deletion of tuples, they perform selection on the augmented view.

Since change in the WHERE clause may either insert tuples to the view or delete tuples from the view, two efficient algorithms are required, one for insertion and the other for deletion. In this section we present two such algorithms for view adaptation and they are one phase algorithms.

4.1 Adaptation Algorithms

Let the schemas of R and S be $(\bar{A}\bar{B})$ and $(\bar{B}\bar{C}\bar{D})$, respectively. Suppose R and S are stored at different sites. The view V is defined as

CREATE VIEW V **AS**
SELECT $\bar{A}$, $\bar{C}$
FROM R, S
WHERE $R.\bar{B} = S.\bar{B}$ and C_1 and C_2 and $\cdots$ C_m

Let the new view V' be defined as

CREATE VIEW V' **AS**
SELECT $\bar{A}$, $\bar{C}$
FROM R, S
WHERE $R.\bar{B} = S.\bar{B}$ and C_1' and C_2 and $\cdots$ C_m

Here we assume that the attributes mentioned by C_1 and C_1' are in S and $C_2, \cdots, C_m$ are conditions formulated either on attributes of R or S or both (we assume that C_1 is not redundant in V.) (Note that when a new condition is added in the V', we can express this as a change of condition by adding a tautologically true condition to the old view V definition. When a condition is deleted from the view, it is equivalent to replace the condition by a tautologically true condition.)

Now we present the two view adaptation algorithms. Both algorithms will maintain a 'derive-count' attribute C_V on view V as extra information and its value indicates how many times a view tuple has been derived in the view. This count will be important

for the efficiency of the deletion algorithm. We also augment the schemas of R and S with 'join-count' attributes. We augment the schema of R with a 'join-count' attribute $C_{\bar{B}}$ for $\bar{B}$ and similarly in S. We assume that the view contains at least one attribute from all its participating relations. In these algorithms, we first find those tuples which are affected by the changed condition and then treat them as if they are going to be deleted (inserted) from (to) the base relation. The insertion or deletion case can be found as follows:
If the condition C_1' implies C_1 then tuples will be inserted into the view; for this case algorithm 2 will be used. Otherwise, tuples will be deleted from the view, and algorithm 3 will be used.

Algorithm 2 Counting insertion algorithm

1. At the site of S,
   - executing the following query
     SELECT * INTO I **FROM** S **WHERE** $C_{\bar{B}} > 0$ and *not* C_1 and C_1'
   - send I to the site of R;
2. At the site of R,
   - let $K = \pi_{\bar{A}\bar{C}}\sigma_{C_1',C_2,\cdots,C_m}(I \bowtie_{I.\bar{B}=R.\bar{B}} R)$; also keep the 'derive-count' $C_{\bar{K}}$ for each tuple in K.
   - send K to the site of V;
3. At the site of V,
   - $V' = V \cup K$; increase C_V of each tuple of V by the corresponding C_K of K. If a tuple of K is not in V, include it in V.

The above algorithm is more efficient when the view V is at the site of R. In this case, the communication cost is $|I|$, which is $p_S * |S|$ under uniform distribution. If the view V is at site S or at other site, then the cost is $|I| + |K|$. In contrast, the cost to adapt the view by applying the computing-from-scratch algorithm will be $|S|$ provided V is at the site of R, $|R|$ if V is the site of S, otherwise $|R| + |S|$.

Algorithm 3 Counting deletion algorithm

1. At the site of V,
   - if the attribute of condition is in the view, then execute the following query
     DELETE * FROM V **WHERE** *not* C_1' & C_1
   - terminate the algorithm.
2. At the site of S,
   - if the attribute of condition is not in the view, then execute the following query
     SELECT * INTO I **FROM** S **WHERE** $C_{\bar{B}} > 0$ and C_1 and *not* C_1'
   - send I to site R;
3. At the site of R,
   - let $K = \pi_{\bar{A}\bar{C}}\sigma_{C_1',C_2,\cdots,C_m}(I \bowtie_{I.\bar{B}=R.\bar{B}} R)$; also keep C_K for each tuple in K.
   - send K to site V;
4. At the site of V,
   - decrease C_V of each tuple of V by the corresponding C_K of K. A tuple with a C_V of zero in V will be deleted from the V.

The communication cost of the above algorithm is given below.

- If the attribute of condition is in the view, the the cost will be zero.
- If the attribute of condition is not in the view and view is at the site of R, then the cost will be $|I|$, which is $p_S * |S|$ under uniform distribution. When the view V is at the site of S or at other site, then the cost is $|I| + |K|$.

In contrast, the cost to adapt the view by applying the computing-from-scratch algorithm (without 'join-count' attribute on base relations) will be zero provided the attribute of condition is present in the view. When the attribute is not present in the view, then the cost is $|S|$ if V is at the site of R, $|R|$ if V is the site of S, otherwise $|R| + |S|$.

5 Changing the FROM clause

In this section we discuss view adaptation method for the situation where the new view is obtained from the old view by deleting (or inserting) a relation from (to) the FROM clause.

Reference [9] proposed a method for such view adaptation for centralised databases. Their method for the situation when a relation is added can be easily modified to be applicable in data warehousing. However, their method for the situation when a relation is deleted is very restricted; in fact it works only when duplicates of tuples are maintained and dangling tuples are allowed in the view.

We now propose a view adaptation algorithm for the situation where a relation is deleted from the FROM clause. This algorithm does not have strict restrictions as needed by the algorithm of [9]. The communication cost of this algorithm is low, and this is achieved by keeping "join-count" for tuples in base relations and "derive-count" for tuples in the view.

5.1 Algorithm for Deleting a Relation

Suppose the schemas of R, S, and T are $(\bar{A}\bar{B})$, $(\bar{B}\bar{C}\bar{D})$, and $(\bar{C}\bar{E})$, respectively, and suppose these relations are stored at different sites. Let the old view V be defined as

CREATE VIEW V **AS**
SELECT $\bar{A}$,$\bar{C}$,$\bar{E}$
FROM R, S, T
WHERE $R.\bar{B} = S.\bar{B}$ and $S.\bar{C} = T.\bar{C}$
and $\mathcal{C}$ and $\mathcal{C}'$

where, $\mathcal{C}'$ is a condition involving attributes of T and $\mathcal{C}$ is the remainder of the conditions. Suppose V' is obtained by deleting T from the FROM clause and hence deleting every reference to T:

CREATE VIEW V' **AS**
SELECT $\bar{A}$,$\bar{C}$
FROM R, S
WHERE $R.\bar{B} = S.\bar{B}$ and $\mathcal{C}$

Observe that $\pi_{\bar{A}\bar{C}}\sigma_{\mathcal{C}}(R \bowtie_{R.\bar{B}=S.\bar{B}} S)$ may not be completely contained in $\pi_{\bar{A}\bar{C}}(V)$. Indeed, there can be tuples, say $\bar{u}$, generated from tuples in $R \bowtie_{R.\bar{B}=S.\bar{B}} S$ which do not join with any tuples of T. Therefore, we have to get all such tuples $\bar{u}$. To accomplish this, we augment the schema of R with a 'join-count' attribute $\mathrm{C}_{\bar{B}}$ for $\bar{B}$, the schema of S with two 'join-count' attributes $\mathrm{C}_{\bar{B}}$ & $\mathrm{C}_{\bar{C}}$ (one for $\bar{B}$ and other for $\bar{C}$), and the schema of T with a 'join-count' attribute $\mathrm{C}_{\bar{C}}$ for $\bar{C}$. We also augment a 'derive-count' attribute C_V in V for indicating the number of derivations of each tuple in the view.

The algorithm is described below.

Algorithm 4 Deletion Algorithm

1. At the site of V,
   - perform
     SELECT $\bar{A}$,$\bar{C}$,C_V **INTO** I **FROM** V.
2. At the site of S,
   - find all those tuples in S whose 'join-count' value C_C is zero and C_B is nonzero, and store them in relation J.
   - send J to the site of R.
3. At the site of R,
   - let $K = \pi_{\bar{A}\bar{C}}\sigma_{\mathcal{C}}(J \bowtie_{J.\bar{B}=R.\bar{B}} R)$; also keep the 'derive-count' of the number of derivations for each tuple in K.
   - send K to the site of V.
4. At the site of V,
   - perform $V' = I \bigcup K$; also increase C_V of each tuple of I by the 'derive-count' of corresponding tuple of K.

The above algorithm is more efficient when view V is stored at site R. In this case, the communication cost is $|J|$, which is $p_S * |S|$ under uniform distribution. If the view V is at other site, then the cost is $|J| + |K|$. A better estimate of $|J|$ under the uniform distribution would be $p_{SR} * (1 - p_{ST}) * |S|$, where $p_{SR} = (|R \bowtie_{R.\bar{B}=S.\bar{B}} S|)/(|R| * |S|)$ and $p_{ST} = (|S \bowtie_{S.\bar{C}=T.\bar{C}} T|)/(|S| * |T|)$. In contrast, the cost to adapt the view by applying the computing-from-scratch algorithm (without 'join-count' attribute on base relations) is $|S|$, if V is at the site of R, $|R|$ if V is the site of S, otherwise $|R| + |S|$.

6 Conclusions

In this paper, we have discussed the view adaptation problem in data warehouses. In the view adaptation problem, old materialised views are adapted, as far as possible, in order to obtain new materialised

views when old views are redefined. Our objective is to minimise the communication cost while adapting the views. A good solution to this problem is important for both centralised and distributed database applications, such as data visualisation, mobile computing, data warehousing, in-home digital services etc. We have proposed view adaptation algorithms for changes in each SELECT, FROM, and WHERE clause in the SPJ queries. Our objective is to minimise the communication cost while adapting the views. The main idea behind these algorithms is to associate a 'join-count' attribute for each join on base relations and a 'derive-count' attribute on each view as extra information. By associating this extra information to base relations and views, the total communication cost is reduced significantly. In future work, we plan to investigate algorithms for adapting views having complicated definitions.

References

[1] J. Bailey, G. Dong, M. Mohania, and X. Sean Wang. Distributed view maintenance by incremental semijoin and tagging. *Distributed and Parallel Databases*, 6(3):287–309, 1998.

[2] J. A. Blakeley, P.-A. Larson, and F. W. Tompa. Efficiently updating materialized views. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 61–71, 1986.

[3] S. Chaudhuri, Ravi Krishnamurthy, Spyros Potamianos, and Kyuseok Shim. Optimizing queries with materialized views. In *Proc. Int. Conf. on Data Engineering*, 1995.

[4] G. Dong and M. Mohania. Algorithms for view maintenance in mobile databases. In *1^{st} Australian Workshop on Mobile Computing and Databases, Monash University*, 1996.

[5] G. Dong and R. Topor. Incremental evaluation of datalog queries. In *Proc. Int'l Conference on Database Theory*, pages 282–296, Berlin, Germany, October 1992.

[6] T. Griffin and L. Libkin. Incremental maintenance of views with duplicates. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 1995.

[7] A. Gupta and I. S. Mumick. Maintenance of materialized views: problems, techniques, and applications. *IEEE Data Engineering Bulletin, Special Issue on Materialized Views and Warehousing*, 18(2), 1995.

[8] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 157–166, 1993.

[9] A. Gupta, I.S. Mumick, and K.A. Ross. Adapting materialized views after redefinitions. In *Proc. ACM SIGMOD International Conference on Management of Data, San Jose, USA*, 1995.

[10] S. Konomi, T. Furukawa, and Y. Kambayashi. Super-key classes for updating materialized derived classes in object bases. In *Proc. 3rd Int'l Conf. on Deductive and Object-Oriented Databases*, volume 760 of *Lecture Notes in Computer Science, Springer-Verlag*, pages 310–326. Springer-Verlag, 1993.

[11] P. Larson and H. Z. Yang. Computing queries from derived relations. In *Proc. Int. Conf. on Very Large Data Bases*, pages 259–69, 1985.

[12] K.A. Ross, D. Srivastava, and Sudarshan S. Materialized view maintenance and integrity constraint checking: Trading space for time. In *Proc. ACM SIGMOD International Conference on Management of Data, Montreal, Canada*, 1996.

[13] A. Segev and J. Park. Maintaining materialised views in distributed databases. In *Proceedings of the IEEE International Conference on Data Engineering*, 1989.

[14] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 316–327, 1995.