# Efficient Update of Data Warehouse Views with Generalised Referential Integrity Differential Files

Carson Kai-Sang Leung[1] and Wookey Lee[2]

[1] The University of Manitoba, Winnipeg, MB, Canada
kleung@cs.umanitoba.ca
[2] Sungkyul University, Anyang, Korea
wook@sungkyul.ac.kr

**Abstract.** Data warehouse (DW) views provide an efficient access to information integrated from source data. When changes are made to the source data, the corresponding views may be outdated. Thus, the maintenance of DW views is crucial for the currency of information. Recently, a method was proposed to use referential integrity differential files (RIDFs) to self-maintain DW views that contain select-project-joins over relations modelled in a star schema. However, it is not uncommon for applications to have relations that are modelled in other schemas such as a snowflake schema or a galaxy schema. In this paper, we generalise the concept of RIDFs; we propose a method that uses *generalised RIDFs* to self-maintain the DW views that contain joins over relations modelled in the star schema as well as non-star schemas. Our method computes new views by using only the old materialised views and files that keep the truly relevant tuples in the "delta". Consequently, it avoids accessing the underlying source data, and hence leads to efficient update of DW views.

**Keywords:** Data warehousing, view maintenance, referential integrity constraints, snowflake schema, galaxy schema, self-maintainability.

## 1 Introduction

A data warehouse (DW) is a subject-oriented, integrated, time-variant, and non-volatile collection of data organised in such a way that it supports the decision making process of management [8]. In general, DW views provide a fast access to integrated source data. As changes can be made to the source data, the corresponding views may be outdated. Thus, the maintenance of views is crucial for the currency of information. In other words, views need to be periodically refreshed so as to reflect those updates that have been made to the source data. In response to the changes to the source data, many existing DW views are refreshed by recomputing the contents from scratch (i.e., computing the new views from the updated source data), while some other views are incrementally maintained by accessing the source data. However, these approaches can be costly. Moreover, in many real-life situations, it is not uncommon that only a

D. Bell and J. Hong (Eds.): BNCOD 2006, LNCS 4042, pp. 199–211, 2006.

tiny fraction of some huge source data gets changed. The above approaches require an access to a huge amount of source data. Consequently, both CPU and I/O costs of these approaches can be extremely high. A better approach is to incrementally maintain views *without* accessing the source data. This calls for efficient view maintenance approaches.

Over the past decade, many approaches (e.g., [1-7, 9-13, 15-19]) have been proposed. However, some of these works focused mainly on conventional database views than DW views, and some did not fully exploit *referential integrity (RI) constraints* for view maintenance. For those that maintain DW views based on RI, they used auxiliary structures (e.g., auxiliary views [17], auxiliary relations [15], auxiliary data [9], complements [11]), which can be costly to build in some situations.

Recently, an efficient method, which incurred a lower cost, was proposed to exploit the RI constraints imposed on relations in the source data [13]. Specifically, the method used *referential integrity differential files (RIDFs)* to keep all and only those tuples that are relevant to the maintenance of views. By so doing, DW views can be self-maintained in the sense that the new views (reflecting the changes that were made to the source data) can be formed by using only the old materialised views, differential files (DFs—i.e., files containing the inserted or the deleted tuples), and RIDFs. In other words, the update of views avoided accessing the underlying databases. Such a method was designed for, and worked well on, self-maintaining views that contain a select-project-join (SPJ) over multiple relations modelled in a *star schema*. However, there exist many situations where relations are *not* modelled in the star schema (e.g., a snowflake schema in Example 1, a galaxy schema in Example 2).

*Example 1.* Consider a DW for shipment consists of the following tables:

- `Item (itemID, name, description)`;
- `Location (locCode, city, county, region)`;
- `Shipper (shipperID, locCode, shipperName)`
  `where locCode references Location`; and
- `Shipment (shipmentID, itemID, shipperID)`
  `where itemID references Item, shipperID references Shipper`.

Here, information about shippers and their locations is stored in two different dimension tables, namely `Shipper` and `Location`, due to normalisation. Relations/tables in this DW are modelled in a *snowflake schema*, where the fact table `Shipment` references dimension tables `Shipper` (which, in turn, references the dimension table `Location`) and `Item`. In this model, view $v_1 \equiv \pi_{\texttt{itemID,shipperName}}\sigma_{\texttt{city=Belfast}}(\texttt{Shipment} \bowtie \texttt{Shipper} \bowtie \texttt{Location})$, which finds IDs of items and names of shippers for those shippers located in Belfast, contains an SPJ over three relations modelled in a snowflake schema. □

*Example 2.* Let us add the following table to the above shipment DW:

- `Sales (invoiceID, itemID, locCode, price)`
  `where itemID references Item, locCode references Location`.

Then, in the resulting model, relations/tables are in a *galaxy schema* consisting of a collection of stars and snowflakes. Here, two fact tables `Sales` and `Shipment` share the dimension table `Item`. In this model, view $v_2 \equiv \pi_{\texttt{invoiceID,shipmentID}}\sigma_{\texttt{description=book}}(\texttt{Sales} \bowtie \texttt{Shipment} \bowtie \texttt{Item})$, which finds invoice IDs and shipment IDs for all books, contains an SPJ over three relations modelled in a galaxy schema. □

Given that there exist situations where relations can be modelled in a non-star schema, some natural questions are: How to handle these situations? Can we use RIDFs in these situations? Can RIDFs be helpful? Can we self-maintain the views using just the old materialised views, DFs and RIDFs? If not, what else is needed?

In this paper, we study these questions. Our ***key contribution*** is the development of a novel method, which exploits RI constraints and generalises the ideas of RIDFs, for self-maintaining DW views. Our method uses *generalised RIDFs (GRIDFs)*, which keep all and only those tuples that are (directly or indirectly) relevant to the updates of views. By so doing, the method efficiently self-maintains the views modelled in the star schema as well as non-star schemas (e.g., the snowflake schema, the galaxy schema). With our method, new views can be formed by using *only* (i) the old materialised views, (ii) DFs, (iii) RIDFs, and (iv) *GRIDFs*. In other words, the method avoids accessing any underlying databases to form the new views.

The outline of this paper is as follows. Section 2 gives background. Section 3 describes how we generalise the concepts of RIDFs for self-maintaining DW views involving joins over relations modelled in a snowflake or a galaxy schema. Section 4 discusses further generalisation and potential improvements of GRIDFs. Experimental results are given in Section 5. Finally, conclusions are presented in Section 6.

## 2 Background

In this section, we present some background materials about RI constraints and RIDFs, which are relevant to the rest of this paper.

### 2.1 Referential Integrity Constraints

A referential integrity (RI) constraint can be specified between relations in database and data warehousing environments; it is used to maintain consistency among tuples in the relations. Informally, the constraint states that a tuple $r$ in a relation $R$ (called the **referencing relation**) that refers to another relation $S$ (called the **referenced relation**) must refer to an existing tuple $s$ in $S$. More formally, the foreign key of $R$ (denoted as $R.fk$) must "match" a candidate key of $S$ (denoted as $S.ck$), that is, they must have the same domain and $R.fk = S.ck$.[1] Without loss of generality, we assume in this paper that all relations in the DW are "linked" by RI constraints.

Whenever there is a change to a relation in an underlying database, the corresponding views need to be updated to reflect the change. This can be done using either an immediate mode or a deferred mode. For the former, the views are

[1] A *candidate key* of a relation is a minimal set of attributes whose values uniquely identify each tuple in the relation. A *foreign key* is a set of attributes (in a referencing relation $R$) that either refers to a candidate key of the referenced relation $S$ or is NULL.

refreshed immediately; for the latter, all the changes are first recorded in some *differential files (DFs)*, and the views are then updated periodically using these DFs. Whenever a tuple is inserted into, or deleted from, a referencing relation $R$ or a referenced relation $S$, appropriate actions need to be taken as described below. (i) When a tuple $r$ is **inserted into a referencing relation** $R$, a look-up in $S$ is required to ensure the presence of a tuple $s \in S$ where $s.ck = r.fk$. If $s$ is present, then $r$ is inserted into $R$ as well as the differential file $\Delta R$;[2] otherwise, RI is violated. (ii) When a tuple $r$ is **deleted from a referencing relation** $R$, the tuple $r$ is recorded in the differential file $\nabla R$. (iii) When a tuple $s$ is **inserted into a referenced relation** $S$, the tuple $s$ is recorded in the differential file $\Delta S$. (iv) When a tuple $s$ is **deleted from a referenced relation** $S$, a (reverse) look-up in $R$ is required (for the default mode of "`on delete no action`") to ensure the absence of a tuple $r \in R$ satisfying $r.fk = s.ck$. If $r$ is absent, then $s$ is safely removed from $S$; otherwise (i.e., $r$ exists in $R$), RI is violated and the deletion is rejected. It is interesting to note that the insertion into, or deletion from, one relation ($R$ or $S$) does not affect another one.

### 2.2 Referential Integrity Differential Files

Let us consider view $v_3 \equiv \pi_{\texttt{itemID,city}}\sigma_{\texttt{price}>50}(\texttt{Sales} \bowtie \texttt{Item} \bowtie \texttt{Location})$, which finds item IDs and cities of those sales with price > \$50. This view contains an SPJ over three relations modelled in a *star schema*, where the fact table `Sales` references dimension tables `Item` and `Location`. This view can be expressed—in abstract terms—as $\pi_A\sigma_C(v_4)$, where $v_4 \equiv (F \bowtie D_1 \bowtie D_2)$ such that $F.fk_1$ references $D_1.ck$ and $F.fk_2$ references $D_2.ck$. In subsequent expressions, let us focus on how to efficiently update the join component because it dominates the SPJ operations.

When changes are made to the source data, a **naïve method** to update a DW view—whenever its underlying relations (e.g., $F, D_1, D_2$) of a view are updated—is to ignore the old view and to compute the new view from scratch (e.g., $v_4' = (F' \bowtie D_1' \bowtie D_2')$). However, this method can be very costly, especially when updates are made very frequently or when only a tiny fraction of underlying relations is updated.

It is well-known that an updated relation $R'$ can be expressed as $R' = R - \nabla R \cup \Delta R$, where $R$ is the old relation (e.g., $F, D_1$, or $D_2$), $\Delta R$ is its insertion, and $\nabla R$ is its deletion. So, an **improved method** is to obtain the new view $v_4'$ from the old view $v_4 \equiv (F \bowtie D_1 \bowtie D_2)$, DFs (e.g., $\Delta F, \nabla D_1, \ldots$), and source relations (e.g., $F, D_1, D_2$), as follows:

$$\begin{aligned} v_4' &= (F - \nabla F \cup \Delta F) \bowtie (D_1 - \nabla D_1 \cup \Delta D_1) \bowtie (D_2 - \nabla D_2 \cup \Delta D_2) \\ &= (F \bowtie D_1 \bowtie D_2) - (F \bowtie D_1 \bowtie \nabla D_2) \cup (F \bowtie D_1 \bowtie \Delta D_2) \cup \ldots \\ &\quad - (\Delta F \bowtie \Delta D_1 \bowtie \nabla D_2) \cup (\Delta F \bowtie \Delta D_1 \bowtie \Delta D_2). \end{aligned} \quad (1)$$

[2] Since the views can be updated using the deferred mode, it is more precise to state the following. An insertion of a tuple $r$ into $R$ requires a look-up in the "current" referenced relation ($S - \nabla S \cup \Delta S$). If there exists a tuple $s \in (S - \nabla S \cup \Delta S)$ such that $s.ck = r.fk$, then $r$ is inserted into $R$ as well as $\Delta R$.

Note that, among the $3^3 = 27$ terms in Equation (1), the first term $(F \bowtie D_1 \bowtie D_2)$ is the old view $v_4$. However, many of the remaining 26 terms (e.g., $(F \bowtie D_1 \bowtie \Delta D_2)$) involve source relations.

To avoid accessing the source relations and to efficiently self-maintain DW views, we proposed in BNCOD 2005 a **self-maintainable method with referential integrity differential files (RIDFs)** [13]. By exploiting the properties of RI constraints as well as the nature of the expression for view updates and by using RIDFs, Equation (1) can be simplified to become the following:

$$\begin{aligned} v_4' = v_4 &\cup (\Delta F \bowtie RIDF_F(D_1) \bowtie RIDF_F(D_2)) \\ &\cup (\Delta F \bowtie RIDF_F(D_1) \bowtie \Delta D_2) \cup (\Delta F \bowtie \Delta D_1 \bowtie RIDF_F(D_2)) \\ &\cup (\Delta F \bowtie \Delta D_1 \bowtie \Delta D_2) \ominus \nabla F \ominus \nabla D_1 \ominus \nabla D_2. \end{aligned} \tag{2}$$

Such a simplification is possible because of the following:

- The term $(F \bowtie D_1 \bowtie D_2)$ in Equation (1) represents the old view $v_4$.
- Any terms with $(F \bowtie \Delta D_1)$ give empty relations. Because of RI constraints, for all $f \in F$, there must exist $d \in D_1$ such that $d.ck = f.fk_1$. In other words, there does not exist a tuple $d' \in \Delta D_1$ satisfying $d'.ck = f.fk_1$. Similar comments apply for all terms with $(F \bowtie \Delta D_2)$.
- All the terms involving $\nabla F$ can be grouped together (denoted as $\ominus \nabla F$) as they basically represent the action that all the tuples containing $f \in \nabla F$ can be deleted. Similar comments apply for all terms involving $\nabla D_1$ as well as $\nabla D_2$.
- The term $(\Delta F \bowtie \Delta D_1 \bowtie \Delta D_2)$ involves only three differential files ($\Delta F, \Delta D_1$ and $\Delta D_2$). In other words, no access to the source data is required.
- The remaining three terms—namely, $(\Delta F \bowtie RIDF_F(D_1) \bowtie RIDF_F(D_2))$, $(\Delta F \bowtie RIDF_F(D_1) \bowtie \Delta D_2)$ and $(\Delta F \bowtie \Delta D_1 \bowtie RIDF_F(D_2))$—all use RIDFs. Recall from Section 2.1 that when a tuple $f$ is inserted into $F$, we check if there exists a tuple $d \in D_i$ such that $d.ck = f.fk$. If such $d$ exists, the insertion is successful and $f$ is then recorded in $\Delta F$. Given that the search and check has been performed, one can record the tuple $f$ in a file called *RIDF*. By so doing, the RIDF contains all those tuples ($d$) that are related to the tuples in $\Delta F$. In other words, the RIDF contains all and only those tuples that could be joined with $\Delta F$ in the term $(\Delta F \bowtie D_i)$. Therefore, with the RIDF, the term $(\Delta F \bowtie D_i)$ can be rewritten as $(\Delta F \bowtie RIDF_F(D_i))$, which no longer requires an access to the source data.

While more details can be found in our BNCOD 2005 paper [13], it is important to note that the self-maintenance of DW views with RIDFs was designed and worked well on joins over relations that are modelled in a *star schema*. In the current BNCOD 2006 paper, we extend and generalise the RIDFs to handle situations where relations are modelled in a *non-star schema* (as well as in a star schema).

## 3 Our New Method: Self-maintenance of DW Views with Generalised RIDFs (GRIDFs)

In this section, we start describing our *new* method that *uses our generalised RIDFs to self-maintain DW views*. Like the method with RIDFs, this new one with GRIDFs also exploits referential integrity. However, unlike the method with RIDFs, the method with GRIDFs can be applicable for updating views involving relations that are modelled in non-star schemas (as well as in a star schema).

Here, we start with the base case where views involve an SPJ over three relations; then, in Section 4, we give the general case for $k$ relations. For three relations $R, S$ and $T$, there are various ways in which these relations reference others. For example, $R$ may reference both $S$ and $T$ (e.g., views $v_3, v_4$ shown above). Alternatively, $R$ may also reference $S$, which in turn references $T$ (e.g., view $v_1$ in Example 1). As the third way, it may be the case where both $R$ and $S$ reference $T$ (e.g., view $v_2$ in Example 2).

### 3.1 "Forward-Linked" Generalised RIDFs (fGRIDFs)

Let us consider view $v_1 \equiv \pi_{\texttt{itemID,shipperName}}\sigma_{\texttt{city=Belfast}}(\texttt{Shipment} \bowtie \texttt{Shipper} \bowtie \texttt{Location})$ in Example 1. How to self-maintain $v_1$? Or, a more general question is: How to compute a new view of the form $\pi_A\sigma_C(v_5)$, where $v_5 \equiv (F \bowtie D_1 \bowtie D_2)$ such that $F.fk$ references $D_1.ck$ and $D_1.fk$ references $D_2.ck$?

Learned from Section 2.2, we know the disadvantages of using the naïve method (i.e., start from scratch) and the improved method (i.e., using the old view, DFs, and source relations). Specifically, the former can be very costly, whereas the latter requires accesses to source relations. So, we exploit the RI constraints and obtain the following expression:

$$\begin{aligned} v_5' = v_5 \cup (\Delta F \bowtie RIDF_F(D_1) \bowtie D_2) \cup (\Delta F \bowtie \Delta D_1 \bowtie RIDF_{D_1}(D_2)) \\ \cup (\Delta F \bowtie \Delta D_1 \bowtie \Delta D_2) \ominus \nabla F \ominus \nabla D_1 \ominus \nabla D_2. \end{aligned} \quad (3)$$

Observed from Equation (3), we can easily spot that one of the terms (i.e., the term $(\Delta F \bowtie RIDF_F(D_1) \bowtie D_2)$) still involves $D_2$ (the underlying database). Hence, even if we could use RIDFs, the new view $v_5'$ could not be computed without accessing the source data.

On the surface, it may appear that this is the best we could do. However, a careful study reveals that we could do better. Specifically, are we required to access $D_2$? Do we need to join with the entire $D_2$? The answer in each case is no. We can avoid accessing $D_2$ by using some "files" similar to RIDFs. The "files" store only truly relevant tuples. To elaborate, we extend and generalise the concept of RIDFs, and we come up with *generalised RIDFs (GRIDFs)*. Specifically, our idea can be described as follows. When a tuple $f$ is inserted into $F$, we check if there exists a tuple $d_1 \in D_1$ such that $d_1.ck = f.fk$. If such $d_1$ exists, the insertion is successful. Then, (i) $f$ is recorded in $\Delta F$ and (ii) $d_1$ is recorded in $RIDF_F(D_1)$. Up to this point, the procedure sounds familiar as it is the same as the creation of the RIDF. However, next step is different: The

insertion of $d_1$ into $RIDF_F(D_1)$ then triggers a look-up of $d_2 \in D_2$ such that $d_2.ck = d_1.fk$. Due to the RI constraint, we know that there exists such $d_2 \in D_2$. So, $d_2$ is then inserted into a "file" called the *"forward-linked" generalised RIDF (fGRIDF)*. See the definition below.

**Definition 1 ("Forward-linked" generalised referential integrity differential file (fGRIDF)).** *Let (i) an SPJ view $\pi_A\sigma_C(F \bowtie D_1 \bowtie D_2)$ be created in terms of three relations $F, D_1$ and $D_2$; (ii) a RI constraint be imposed on $F$ and $D_1$ such that $F.fk = D_1.ck$, where $F.fk$ denotes the foreign key of the referencing relation $F$ and $D_1.ck$ denotes a candidate key of the referenced relation $D_1$; and (iii) a RI constraint be imposed on $D_1$ and $D_2$ such that $D_1.fk = D_2.ck$, where $D_1.fk$ denotes the foreign key of the referencing relation $D_1$ and $D_2.ck$ denotes a candidate key of the referenced relation $D_2$. (Note that $D_1$ plays two different roles: It is a referenced relation with respect to $F$, but a referencing relation with respect to $D_2$.) Then, when a tuple $f$ is successfully inserted into $F$ (i.e., $f$ is put in $\Delta F$), the corresponding tuple $d_1$ (where $d_1.ck = f.fk$) is then inserted into $D_1$. This triggers the insertion of $d_2$ into a* **"forward-linked" generalised referential integrity differential file**, *denoted as $fGRIDF_{D_1}(D_2)$, which keeps all and only those tuples (in $D_2$) that are truly relevant to the update of the view. Precisely, for each tuple $d_1 \in RIDF_F(D_1)$, its corresponding $d_2 \in D_2$ (such that $d_2.ck = d_1.fk$) is kept in $fGRIDF_{D_1}(D_2)$.*

There are some nice properties of $fGRIDF_{D_1}(D_2)$. First, $fGRIDF_{D_1}(D_2)$ keeps all and only those tuples (in $D_2$) that are truly relevant to the join $(\Delta F \bowtie RIDF_F(D_1) \bowtie D_2)$. Thus, the number of tuples in $fGRIDF_{D_1}(D_2)$ is bounded above by the number of tuples in $D_2$ (i.e., $|fGRIDF_{D_1}(D_2)| \leq |D_2|$). Second, for each candidate key of $D_2$, the number of tuples in $fGRIDF_{D_1}(D_2)$ is bounded above by the number of tuples in $RIDF_F(D_1)$. This is due to RI constraints. More specifically, because $d_1.fk = d_2.ck$, many $d_1 \in RIDF_F(D_1)$ can reference one $d_2$ (but each $d_1$ can only reference one $d_2$). Hence, if $D_2$ only has one candidate key (which is quite common for dimension tables), then $|fGRIDF_{D_1}(D_2)| \leq |RIDF_F(D_1)|$. Third, since $|RIDF_F(D_1)| \leq |\Delta F|$, we have $|fGRIDF_{D_1}(D_2)| \leq |RIDF_F(D_1)| \leq |\Delta F|$. Therefore, by exploiting properties of RI constraints and using $fGRIDF_{D_1}(D_2)$, Equation (3) can be simplified to become the following (i.e., the new view can be computed as follows):

$$\begin{aligned} v_5' = v_5 &\cup (\Delta F \bowtie RIDF_F(D_1) \bowtie fGRIDF_{D_1}(D_2)) \\ &\cup (\Delta F \bowtie \Delta D_1 \bowtie RIDF_{D_1}(D_2)) \\ &\cup (\Delta F \bowtie \Delta D_1 \bowtie \Delta D_2) \ominus \nabla F \ominus \nabla D_1 \ominus \nabla D_2. \end{aligned} \tag{4}$$

It is important to note that, with this self-maintainable method with GRIDFs, we no longer require accesses to the source data. The new view $v_5'$ can be computed using only (i) the old view $v_5$, (ii) DFs, (iii) RIDFs, and (iv) GRIDFs (i.e., fGRIDFs). See the following example.

*Example 3.* Let us consider the self-maintenance of view $v_6 \equiv$ (`Shipment` $\bowtie$ `Shipper` $\bowtie$ `Location`), which is modelled in a snowflake schema. (Note that view $v_1$ in Example 1

can be expressed as $\pi_{\texttt{itemID,shipperName}}\sigma_{\texttt{city=Belfast}}(v_6)$.) When the underlying databases are updated, the new view $v_6'$ can be computed as follows:

$$\begin{aligned} v_6' = v_6 &\cup (\Delta\texttt{Shipment} \bowtie RIDF_{\texttt{Shipment}}(\texttt{Shipper}) \bowtie fGRIDF_{\texttt{Shipper}}(\texttt{Location})) \\ &\cup (\Delta\texttt{Shipment} \bowtie \Delta\texttt{Shipper} \bowtie RIDF_{\texttt{Shipper}}(\texttt{Location})) \\ &\cup (\Delta\texttt{Shipment} \bowtie \Delta\texttt{Shipper} \bowtie \Delta\texttt{Location}) \\ &\ominus \nabla\texttt{Shipment} \ominus \nabla\texttt{Shipper} \ominus \nabla\texttt{Location}. \end{aligned}$$ □

### 3.2 "Backward-Linked" Generalised RIDFs (bGRIDFs)

Next, let us consider view $v_2 \equiv \pi_{\texttt{invoiceID,shipmentID}}\,\sigma_{\texttt{description=book}}(\texttt{Sales} \bowtie \texttt{Shipment} \bowtie \texttt{Item})$ in Example 2. How to self-maintain $v_2$? Or, a more general question is: How to compute a new view of the form $\pi_A\sigma_C(v_7)$, where $v_7 \equiv (F_1 \bowtie F_2 \bowtie D)$ such that $F_1.fk$ references $D.ck_1$ and $F_2.fk$ references $D.ck_2$?

By applying the method with RIDFs [13] (or applying our proposed method with fGRIDFs described in Section 3.1), we obtain the following expression:

$$\begin{aligned} v_7' = v_7 &\cup (F_1 \bowtie \Delta F_2 \bowtie RIDF_{F_2}(D)) \cup (\Delta F_1 \bowtie F_2 \bowtie RIDF_{F_1}(D)) \\ &\cup (\Delta F_1 \bowtie \Delta F_2 \bowtie [RIDF_{F_1}(D) \cap RIDF_{F_2}(D)]) \\ &\cup (\Delta F_1 \bowtie \Delta F_2 \bowtie \Delta D) \ominus \nabla F_1 \ominus \nabla F_2 \ominus \nabla D. \end{aligned} \quad (5)$$

Observe that two of the terms (i.e., the second and the third terms) still involve those underlying databases (i.e., $F_1$ and $F_2$). So, what we need is another type of generalised RIDF, which we call the *"backward-linked" generalised RIDF (bGRIDF)*. Specifically, the idea can be described as follows. When a tuple $f_i$ is inserted into $F_i$, we check if there exists a tuple $d \in D$ such that $d.ck = f_i.fk$. If such $d$ exists, the insertion is successful. Then, (i) $f_i$ is recorded in $\Delta F_i$ and (ii) $d$ is recorded in $RIDF_{F_i}(D)$. Again, the extra/new step is as follows: The insertion of $d$ into $RIDF_{F_i}(D)$ triggers a reverse look-up of $f_j \in F_j$ (where $j \neq i$) such that $f_j.fk = d.ck$. Due to the RI constraint, we know that there could be no such $f_j \in F_j$. However, if one exists, it is then inserted into a "backward-linked" generalised RIDF (i.e., $bGRIDF_D(F_j)$). See the definition below.

**Definition 2 ("Backward-linked" generalised referential integrity differential file (bGRIDF)).** *Let (i) an SPJ view $\pi_A\sigma_C(F_1 \bowtie F_2 \bowtie D)$ be created in terms of three relations $F_1, F_2$ and $D$; and (ii) a RI constraint be imposed on $F_i$ and $D$ such that $F_i.fk = D.ck$ where $F_i.fk$ denotes the foreign key of the referencing relation $F_i$ (for $i = 1, 2$) and $D.ck$ denotes a candidate key of the referenced relation $D$. Then, when a tuple $f_i$ is successfully inserted into $F_i$ (i.e., $f_i$ is put in $\Delta F_i$), the corresponding tuple $d$ (where $d.ck = f_i.fk$) is then inserted into $D$. This triggers the insertion of $f_j$ (where $j \neq i$) into a* **"backward-linked" generalised referential integrity differential file**, *denoted as $bGRIDF_D(F_j)$, which keeps all and only those tuples (in $F_j$) that are truly relevant to the update of the view. Precisely, for each tuple*

*$d \in RIDF_{F_i}(D)$, its corresponding $f_j \in F_j$ (such that $f_j.fk = d.ck$) is kept in $bGRIDF_D(F_j)$.*

A nice property of the $bGRIDF_D(F_j)$ is that it keeps all and only those tuples (in $F_j$) that are truly relevant to the join ($\Delta F_i \bowtie F_j \bowtie RIDF_{F_i}(D)$). Thus, the number of tuples in $bGRIDF_D(F_j)$ is bounded above by the number of tuples in $F_j$ (i.e., $|bGRIDF_D(F_j)| \leq |F_j|$). Another property is that, for a given $d \in RIDF_{F_i}(D)$, there could be no $f_j \in F_j$ (where $f_j.fk = d.ck$) referencing it. This potentially reduces the size of $bGRIDF_D(F_j)$. Therefore, by exploiting properties of RI constraints and using $bGRIDF_D(F_i)$, Equation (5) can be simplified to become the following:

$$\begin{aligned} v_7' = v_7 &\cup (bGRIDF_D(F_1) \bowtie \Delta F_2 \bowtie RIDF_{F_2}(D)) \\ &\cup (\Delta F_1 \bowtie bGRIDF_D(F_2) \bowtie RIDF_{F_1}(D)) \\ &\cup (\Delta F_1 \bowtie \Delta F_2 \bowtie [RIDF_{F_1}(D) \cap RIDF_{F_2}(D)]) \\ &\cup (\Delta F_1 \bowtie \Delta F_2 \bowtie \Delta D) \ominus \nabla F_1 \ominus \nabla F_2 \ominus \nabla D. \end{aligned} \tag{6}$$

It is important to note that, with this self-maintainable method with GRIDFs, we no longer require accesses to the source data. The new view $v_7'$ can be computed using (i) the old view $v_7$, (ii) DFs, (iii) RIDFs, and (iv) GRIDFs (e.g., bGRIDFs). See the following example.

*Example 4.* Let us consider the self-maintenance of view $v_8 \equiv$ (`Sales` $\bowtie$ `Shipment` $\bowtie$ `Item`), which is modelled in a galaxy schema. (Note that view $v_2$ in Example 2 can be expressed as $\pi_{\texttt{invoiceID,shipmentID}}\sigma_{\texttt{description=book}}(v_8)$.) When the underlying databases are updated, the new view $v_8'$ can be computed as follows:

$$\begin{aligned} v_8' = v_8 &\cup (bGRIDF_{\texttt{Item}}(\texttt{Sales}) \bowtie \Delta\texttt{Shipment} \bowtie RIDF_{\texttt{Shipment}}(\texttt{Item})) \\ &\cup (\Delta\texttt{Sales} \bowtie bGRIDF_{\texttt{Item}}(\texttt{Shipment}) \bowtie RIDF_{\texttt{Sales}}(\texttt{Item})) \\ &\cup (\Delta\texttt{Sales} \bowtie \Delta\texttt{Shipment} \bowtie [RIDF_{\texttt{Sales}}(\texttt{Item}) \cap RIDF_{\texttt{Shipment}}(\texttt{Item})]) \\ &\cup (\Delta\texttt{Sales} \bowtie \Delta\texttt{Shipment} \bowtie \Delta\texttt{Item}) \ominus \nabla\texttt{Sales} \ominus \nabla\texttt{Shipment} \ominus \nabla\texttt{Item}. \end{aligned}$$ □

## 4 Discussion: Generalisation to Multiple Relations

So far, we have described and explained our proposed efficient method with GRIDFs for self-maintaining DW views involve an SPJ over three relations: (i) the use of fGRIDFs for an SPJ over a chain of one fact table and two dimension tables (Section 3.1), and (ii) the use of bGRIDFs for an SPJ over two fact tables that share a dimension table (Section 3.2). As expected, our method is not confined to just three relations. It can be further generalised to handle multiple relations by exploiting RI constraints. For example, a new view containing an SPJ over a chain of a fact table $F$ and $k$ levels of dimension tables $D_1, \ldots, D_k$ can be computed using $2k + 3$ terms as follows:

$$\begin{aligned} v' &= F' \bowtie D_1' \bowtie \cdots \bowtie D_k' \\ &= v \ \cup (\Delta F \bowtie RIDF_F(R_1) \bowtie fGRIDF_{R_1}(R_2) \bowtie \cdots \bowtie fGRIDF_{R_1}(R_k)) \end{aligned}$$

$$\cup [\bigcup_{j=2}^{k-1} (\Delta F \bowtie \Delta R_1 \bowtie \cdots \bowtie \Delta R_{j-1} \bowtie RIDF_{R_{j-1}}(R_j) \bowtie fGRIDF_{R_j}(R_{j+1}) \bowtie \cdots \bowtie fGRIDF_{R_j}(R_k))]$$
$$\cup (\Delta F \bowtie \Delta R_1 \bowtie \cdots \bowtie \Delta R_{k-1} \bowtie RIDF_{R_{k-1}}(R_k))$$
$$\cup (\Delta F \bowtie \Delta R_1 \bowtie \cdots \bowtie \Delta R_k) \ominus \nabla F \ominus \nabla R_1 \cdots \ominus \nabla R_k. \quad (7)$$

Similarly, a new view containing an SPJ over $k$ fact tables $F_1, \ldots, F_k$ that share a dimension table $D$ can be computed using $2^k + k + 2$ terms as follows:

$$v' = F_1' \bowtie \cdots F_k' \bowtie D'$$
$$= v \cup \left( \bigcup \Delta F_i \bowtie bGRIDF_D(F_j) \bowtie [\cap_i RIDF_{F_i}(D)] \right)$$
$$\cup (\Delta F_1 \bowtie \cdots \bowtie \Delta F_k \bowtie \Delta D) \ominus \nabla F_1 \cdots \ominus \nabla F_k \ominus \nabla D, \quad (8)$$

where $1 \leq i, j \leq k$. With this generalisation, one would be able to efficiently compute the new DW views that contain an SPJ over different numbers of relations modelled in various schemas (e.g., star, snowflake, or galaxy schemas). One does not need to access the underlying databases during the update. All it needs is the old view, DFs, RIDFs, and our proposed GRIDFs (i.e., fGRIDFs and/or bGRIDFs).

Our proposed GRIDFs can further be improved by keeping only relevant *attributes* of the relevant tuples. Any attributes that do not contribute to the update of DW views can be discarded. Moreover, any tuples that do not contribute to the selection operator (of the SPJ) can also be discarded.

## 5 Experimental Results

We ran several experiments on various DWs. The results cited below are based on a DW that consists of a fact table (with 6,000,000 tuples) and multiple dimension tables (each with 800,000 tuples) that are modelled in a snowflake schema. In the experiments, we compared the results of the following four implemented methods:

- The **naïve method**, which recomputes new views from scratch.
- The **improved method**, which uses *old views, DFs*, and *source relations* to update the views.
- The self-maintainable **method with RIDFs** [13], which uses only old views, DFs, and *RIDFs*.
- Our efficient self-maintainable **method with GRIDFs**, which uses only old views, DFs, RIDFs, and *GRIDFs*. This method avoids accessing source relations even for those that are modelled in a non-star schema (e.g., the snowflake or galaxy schema).

In the first experiment, we fixed the number of dimension tables to 2 (i.e., the view $(F \bowtie D_1 \bowtie D_2)$ that contains an SPJ over a fact table $F$ and two dimension tables $D_1$ & $D_2$. We varied the percentage of tuples being updated/changed

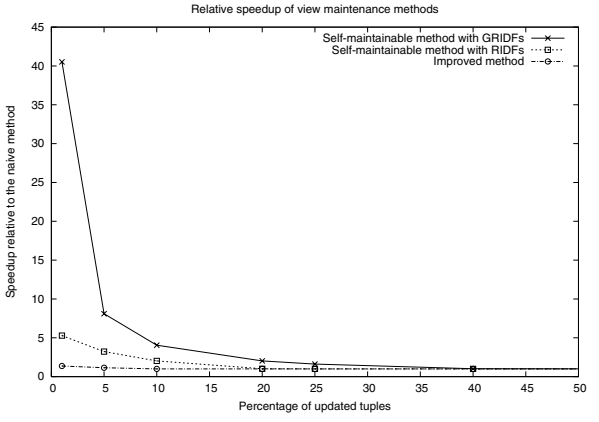


**Fig. 1.** Relative speedup of view maintenance methods

from 1% to 50%. The x-axis of Fig. 1 shows the percentage of updated tuples; the y-axis shows the speedup of the improved method, the method with RIDFs, and our self-maintainable method with GRIDFs against the naïve method. As observed from Fig. 1, the lower the percentage of updated tuples, the higher is the benefit of using our method. For example, the speedup of our method is above 40 times when 1% of tuples are updated. A much higher speedup is expected when the percentage of updated tuples is lower (e.g., 0.1%). Note that a low percentage of updated tuples is not uncommon. In many real-life applications, DW views need to be refreshed frequently (which usually leads to a low percentage of tuples being updated between each refresh) so as to facilitate accurate decision making.

While Fig. 1 shows the relative speedup, the table below gives some samples of the total runtime (i.e., both CPU and I/O time) for updating view.

| % updated tuples | Naïve | Improved | RIDFs | GRIDFs |
|---|---|---|---|---|
| 1% | 714 mins | 520 mins | 135 mins | 17.6 mins |
| 10% | 714 mins | 621 mins | 222 mins | 88.1 mins |

Note that our proposed method with GRIDFs requires a much shorter runtime than the other three methods. The reason is that our method uses GRIDFs; it does not need to access source relations. In contrast, the method with RIDFs, which uses DFs and RIDFs, needs to access the source relation $D_2$. The improved method, which uses DFs but not RIDFs, needs to access more source relations (both $D_1$ and $D_2$).

Next, we varied the number of dimension tables. The results show that increasing the number of dimension tables increases the speedup of our method and increases the runtime gaps. Note that when there are $k$ dimension tables, the improved method and the method with RIDFs need to access $k$ and $(k-1)$ source relations respectively. In contrast, our proposed method with GRIDFs does not need to access any source relations.

Then, let us count the numbers of tuples in the source relations, the "delta", RIDFs, and GRIDFs. It was observed that the number of tuples needed to be stored in a GRIDF is bounded above by the numbers of tuples in its corresponding source relations, RIDFs, and "delta" (e.g., $|fGRIDF_F(D_2)| \leq \min\{|D_2|, |RIDF_F(D_1)|, |\Delta F|\}$).

To summarise, the experimental results show the effectiveness of our proposed self-maintainable method with GRIDFs. Since the results on various DWs were consistent (and for lack of space), we do not show all the results here. For more details, please refer to our technical report [14].

## 6 Conclusions

Data warehouse (DW) views provide an efficient access to integrated data. As changes are made to the source data, the corresponding views may be outdated. Hence, the maintenance of views is crucial for the currency of information. In this paper, we proposed a novel method to efficiently self-maintain the DW views that contain a select-project-join (SPJ) over multiple relations. Specifically, we exploit the RI constraints imposed on the relations in the source data, and generalise the referential integrity differential files (RIDFs). The *generalised RIDFs (GRIDFs)*, proposed in this paper, keep the truly relevant tuples in the "delta"; they avoid accessing the underlying databases. Consequently, our method can update DW views by using only the old views, differential files (e.g., the insertion file $\Delta R$ and the deletion file $\nabla R$), RIDFs, and *GRIDFs*. The method is applicable to the efficient self-maintenance of views that contain an SPJ over relations modelled in various schemas in data warehousing environments.

**Acknowledgements.** This project is partially sponsored by Science and Engineering Research Canada (NSERC) and The University of Manitoba, as well as Korea Science and Engineering Foundation (KOSEF) through Advanced Information Technology Research Centre (AITrc), in the form of research grants.

## References

1. Blakeley, J.A., Larson, P.-Å., Tompa, F.W.: Efficiently updating materialized views. In: Proc. SIGMOD 1986. 61–71
2. Bruckner, R.M., Tjoa, A.M.: Managing time consistency for active data warehouse environments. In: Proc. DaWaK 2001. 254–263
3. Engström, H., Lings, B.: Evaluating maintenance policies for externally materialised multi-source views. In: Proc. BNCOD 2003. 140–156
4. Fišer, B., Onan, U., Elsayed, I., Brezany, P., Tjoa, A.M.: On-line analytical processing on large databases managed by computational grids. In: Proc. DEXA Workshop (GLOBE) 2004. 556–560
5. Griffin, T., Libkin, L., Trickey, H.: An improved algorithm for the incremental recomputation of active relational expressions. IEEE TKDE **9** (1997) 508–511
6. Gupta, H., Mumick, I.S.: Selection of views to materialize in a data warehouse. IEEE TKDE **17** (2005) 24–43

7. Hyun, N.: Multiple-view self-maintenance in data warehousing environments. In: Proc. VLDB 1997. 26–35
8. Inmon, W.H.: Building the Data Warehouse. John Wiley & Sons (1996)
9. Khan, S., Mott, P.: LeedsCQ: a scalable continual queries system. In: Proc. DEXA 2002. 607–617
10. Kotidis, Y., Roussopoulos, N.: A case for dynamic view management. ACM TODS **26** (2001) 388–423
11. Laurent, D., Lechtenbörger, J., Spyratos, N., Vossen, G.: Monotonic complements for independent data warehouses. VLDB Journal **10** (2001) 295–315
12. Lee, W.: On the independence of data warehouse from databases in maintaining join views. In: Proc. DaWaK 1999. 86–95
13. Leung, C.K.-S., Lee, W.: Exploitation of referential integrity constraints for efficient update of data warehouse views. In: Proc. BNCOD 2005. 98–110
14. Leung, C.K.-S., Lee, W.: GRIDFs: generalised referential integrity differential files for maintaining data warehouse views. Technical report TR 06/02, Department of Computer Science, The University of Manitoba, Canada (2006)
15. Mohania, M., Kambayashi, Y.: Making aggregate views self-maintainable. DKE **32** (2000) 87–109
16. Qian, X., Wiederhold, G.: Incremental recomputation of active relational expressions. IEEE TKDE **3** (1991) 337–341
17. Quass, D., Gupta, A., Mumick, I., Widom, J.: Making views self-maintainable for data warehousing. In: Proc. PDIS 1996. 158–169
18. Theodoratos, D., Xu, W.: Constructing search spaces for materialized view selection. In: Proc. DOLAP 2004. 112–121
19. Zhuge, Y., Garcia-Molina, H., Hammer, J., Widom, J.: View maintenance in a warehousing environment. In: Proc. SIGMOD 1995. 316–327