경영정보학연구
제14권 제3호
2004년 9월

다중 데이터 원천을 가지는 데이터웨어하우스 뷰의 자율갱신*

이 우 기**

Self Maintainable Data Warehouse Views for Multiple Data Sources

Wookey Lee

Self-maintainability of data warehouse (DW) views is an ability to maintain the DW views without requiring an access to (i) any underlying databases or (ii) any information beyond the DW views and the delta of the databases. With our proposed method, DW views can be updated by using only the old views and the differential files such as different files, referential integrity differential files, linked differential files, and backward-linked differential files that keep the truly relevant tuples in the delta. This method avoids accessing the underlying databases in that the method achieves self-maintainability even in preparing auxiliary information. We showed that out method can be applicable to the DW views that contain joins over relations in a star schema, a snowflake schema, or a galaxy schema.

Keywords : Data Warehouses, Self-Maintainability, Referential Integrity, DW Schema

* This work was supported by Korea Science and Engineering Foundation (KOSEF), through Advanced Information Technology Research Center (AITrc).

** 성결대학교 컴퓨터학부

Ⅰ. Introduction

A data warehouse (DW) is a subject-oriented, integrated, time-variant, and non-volatile collection of data organized in such a way that it supports management decision making. DW views need to be periodically refreshed to reflect the updates to the source data. In response to the changes in the source data, many existing DW views are refreshed by recomputing from scratch (i.e., recomputing the new DW views from the updated source data). Alternatively, some existing DW views are incrementally maintained by accessing the source data. However, either of these approaches can be costly. It is not uncommon that only a tiny fraction of some huge source data is changed. The above approaches require an access to the huge source data, which may even be in a remote site. As a result, both CPU and I/O costs of these approaches can be extremely high. A better approach is to incrementally maintain the DW views without accessing the source data.

In this paper, we develop a novel method for self-maintaining views for relations modeled in various schema e.g., a star schema, a snowflake schema, or a galaxy schema.

Ⅱ. Background

Recently Data warehouses are highlighted in large-scale business environments such as OLAP [Trujillo et al., 2003; Sifer, 2003; Lakshmanan et al., 2003], Decision Support System [Nematia et al., 2002; Berndt et al., 2003; Chari, 2003], Data Mining [Zhuang et al., 2003], BI(Business Intelligence) [Wixom and Watson, 2004], and E-commerce [Piccinelli et al., 2003], Continual Queries [Khan and Mott, 2002], Data Streaming [Babcock et al., 2002], and Web applications [Babcock et al., 2002; Lee and Geller, 2004]. Maintaining materialized views under source updates in a DW environment is one of the important issues of data warehousing [Ceri et al., 2000; Lechtenborger, 2003]. In this section, we review existing methods of view maintenance by categorizing several corresponding issues.

Self-maintenance is a notion that can be defined as maintaining views by materializing supplementary data so that the warehouse view can be maintained without (or at least mostly without) accessing base relations. The notion was originally introduced by Blakeley et al. [Blakeley et al., 1986]. The main idea is based on a Boolean expression with sufficient and necessary conditions on the view definition for autonomously computable updates that can be called self-maintainable views. Such studies focus on the conventional database views, not on the DW view. Theodoratos et al. [2001] summarize issues extensively related to self-maintainability, and suggest a view selection approach based on a DAG method. Several notable articles that deal with self-maintenance aim to develop algorithms related to the integration and the maintenance of information extracted from heterogeneous and autonomous sources[Hyun, 1997; Quass et al., 1996].

Algebraic approaches in maintaining DW views are discussed in [Quian and Widerhold, 1991; Griffin and Libkin, 1995; Hyun, 1997; Gupta and Mumick, 1999; Theodoratos et al., 2001]. Quian and Widerhold[1991] present an algorithm for incremental view maintenance based on finite differencing techniques. The al-

gorithm derives the minimal incremental changes in an arbitrary relational expression for a view modification by replacing the original relational algebraic expression with an efficient and incremental re-computation. However, the algorithm uses source relations and thus it lacks the self-maintenance notion. Griffin & Libkin in [Griffin and Libkin, 1995] extend the techniques in [Quian and Widerhold, 1991]. Hyun [1997] proposes to include functional dependencies. Gupta and Mumick[1999] integrate outer joins. These references do not consider the concepts of referential integrity for the maintenance of DW views. In this paper, some of the common notations (mainly from [Quian and Widerhold, 1991; Griffin and Libkin, 1995]) are extended to present some propagation rules for DW views based on referential integrity constraints.

There has been some research that considers the database system as a rule system [Ceri et al., 2000] that presents a comprehensive survey on the roles played in DW views. In that paper, the rule is classified as a constraint or a trigger in that the constraint is descriptive while the trigger is procedural. (However, in this paper we use the term 'constraint' interchangeably with 'rule trigger.') There are several works [Markowitz, 1991; Lechtenborger and Vossen, 2003] corresponding to this method that have the potential to extend referential integrity constraints to the maintenance of database views. When a referential integrity rule invokes cascade among database rules in the DBMS, [Markowitz, 1991] presents the run time execution problem and the safeness condition respectively. Lechtenborger and Vossen [2003] investigate the view maintenance problem with inclusion dependency but no referential integrity rules.

Database rules, including referential integrity constraints, are utilized in maintaining materialized views in several articles such as [Mohania and Kambayashi, 2000; Quass et al., 1996]. Quass et al.[1996] use the referential integrity constraint to determine whether a base relation is participating in the views, and [Quass et al., 1996] extends the works of Quian and Widerhold [1991] and Griffin and Libkin[1995] to transform change propagation equations into more efficient ones. They use an 'auxiliary view' in [Mohania and Kambayashi, 2000] in order to maintain a select-project-join (SPJ) view without accessing base relations at the sources, which is similarily called as 'auxiliary relation' in [Kahn and Mott, 2002], and 'complements' in [Lechtenborger and Vossen, 2003]. However, the validity and the performance of these methods are strongly dependent upon query types, as long as the view conditions can screen the corresponding base relation. This is discussed in the motivational examples in Section 2.

Even though the auxiliary view is neither a simple view (e.g., to join several tables) nor does the view have any selection condition (e.g., group-by condition), then it still has to depend on the base relations. In that case, the auxiliary view would be the base relation per se or the whole relation should be replicated as auxiliary information. Maintaining the replica of base relations itself invokes a serious problem, in addition to synchronization and concurrency control problem [Zhuge et al., 1995; Ceri et al., 2002]. Therefore, the method is not truly self-maintainable or is just a nominal self-maintainable method. In other words, it is not practical.

A referential integrity constraint is one of the most fundamental constraints that any relational databases should satisfy. It can be specified between two relations in a database, and used to maintain consistency among tuples in the two 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.

To define the referential integrity constraints more formally, let us remind the reader the concepts of *candidate keys and foreign keys*. 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 refers to a candidate key of the referenced relation S. The foreign key of R must "match" the candidate key of S, that is, they must have the same domains and $R.fk = S.ck$ (where $R.fk$ denotes a foreign key of R and $S.ck$ denotes a candidate key of S). Without loss of generality, we assume, in this paper, that all relations in the database, which is an organized collection of related data, are "linked" by referential integrity 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 either in an immediate mode or a deferred mode. In the former, the views are refreshed immediately; in the latter, all the changes are first recorded in some *differential files* and the views are then updated periodically using these differential files.

The following describes the appropriate actions to be taken when a tuple is inserted into, or deleted from, a referencing or referenced relation:

Case 1: An insertion into a referencing relation.

Referential integrity can be violated if the value of the foreign key of a referencing relation R does not exist in any candidate key of the referenced relation S. Hence, an insertion of a tuple r into a referencing relation R requires a look-up in the referenced relation S. If there exists a tuple $s \in S$ such that $s.ck = r.fk$, then r is inserted into R. Since the views can be updated using the deferred mode, it is more precise to say the following. An insertion of a tuple r into a referencing relation R requires a look-up in the "current" referenced relation $(S - \nabla S \cup \triangle S)$. If there exists a tuple $s \in (S - \nabla S \cup \triangle S)$ such that $s.ck = r.fk$, then r is inserted into R. Consequently, the tuple r is recorded in the differential file $\triangle R$. It can be easily observed that such an insertion into the referencing relation R does not affect the referenced relation S.

Case 2 : A deletion from a referencing relation.

When a tuple r is deleted from a referencing relation R, it can be recorded in the differential file ∇R. Such a deletion from the referencing relation R does not affect any referenced relation S.

Case 3 : An insertion into a referenced relation.

When a tuple s is inserted into a referenced relation S, it can be recorded in the differential file $\triangle S$. Such an insertion into the referenced relation S does not affect any referencing relation R

Case 4 : A deletion from a referenced relation.

Referential integrity can be violated when a

tuple s is deleted from a referenced relation *S*. There are several options, as shown below, if such a deletion causes a violation:

(a) Reject the deletion (i.e., **"on delete restrict"**): This is the default option. With this option, the deletion of *s* from *S* requires a look-up in the referencing relation *R*. If there exists a tuple $r \in R$ such that *r.fk* = *s.ck*, then such a deletion is rejected. Similar to Case 1, as the views can be updated using the deferred mode, it is more precise to say the following. A deletion of *s* from *S* requires a look-up in the "current" referencing relation ($R - \nabla R \cup \triangle R$). If there exists a tuple $r \in (R - \nabla R \cup \triangle R)$ such that *r.fk* = *s.ck*, then such a deletion is rejected. Similar comments applied to Cases 4(b) and 4(c). No change to both the referencing relation *R* and the referenced relation *S*.

(b) Attempt to cascade or propagate the deletion (i.e., **"on delete cascade"**): With this option, the deletion of *s* from *S* requires a look-up in the referencing relation *R*. If there exists a tuple $r \in R$ such that *r.fk* = *s.ck*, then the database system attempts to delete *r* from *R*. If such an attempt is successful, the deleted tuples are recorded in the appropriate differential files (e.g., *s* and *r* are recorded in ∇S and ∇R, respectively). Otherwise, no change to both the referencing relation *R* and the referenced relation *S*

(c) Modify the referencing attribute values to NULL or the default values (i.e., **"on delete set NULL"** or **"on delete set default"**): With this option, the deletion of *s* from *S* requires a look-up in the referencing relation *R*. If there exists a tuple $r \in R$ such that *r.fk* = *s.ck*, then the database system attempts to modify the value of *r.fk* (provided that such an action does not violate any integrity constraints). If such an attempt is successful, then *s* is deleted and *r* is modified. Consequently, *s* is recorded in ∇S, and *r* is recorded in both ∇R and $\triangle R$ (because a modification can be considered as a deletion-and-insertion pair).

Ⅲ. Maintenance of DW Views Involving Two Relations

In this section, we discuss the situation where a DW view contains a join over two relations *R* and *S*. Without loss of generality, let us assume that *R* references *S*. For example, *R* represents the employee relation (**Emp**) that contains employee information, and *S* represents the department relation (**Dept**) that contains department information as described below:

- Dept (deptID, deptName, budget)
- Emp (empID, empName, salary, deptID), where deptID references Dept.deptID

The view $\pi_{\text{empName,deptName}}\, \sigma_{\text{salary>50k}}$ (Emp ⋈ Dept) finds the employee name and department name for those whose salary is over $50K.

3.1 A Naïve Approach: Recompute DW Views from Scratch

Consider a select-project-join (SPJ) view $\pi_A \sigma_C (R \bowtie S)$ where: (i) *R* is a referencing relation, (ii) *R* is a referenced relation, (iii) σ_C is the (usual) selection based on a Boolean condition *C*, and (iv) π_A is the (usual) projection on a list of attributes *A*. When the underlying relations (namely, *R* and *S*) of the view are updated, we

need to update the view to preserve consistency. A naïve approach is to recompute $\pi_A\sigma_C(R' \bowtie S')$ from scratch, where R' is the new/updated R and S' is new/updated S. However, this approach can be very costly, especially when only a tiny fraction of R or S is updated.

3.2 A Basic Problem Representation

A more efficient approach is to obtain the new view from the old view, differential files, and source relations. It is well-known that the new/updated referencing relation R' can be expressed in terms of the old relation R, its insertion $\triangle R$, and its deletion ∇R, i.e., $R' = R - \nabla R \cup \triangle R$. Similarly, the new/updated referenced relation S' can be expressed as $S' = S - \nabla D \cup \triangle S$. Therefore, the new view $\pi_A\sigma_C(R' \bowtie S')$ can be expressed as follows. To simplify our presentation, in the remainder of this paper, we omit the select and project components, and only show the join component:

$$v' = (R - \nabla R \cup \triangle R) \bowtie (S - \nabla S \cup \triangle S) \quad (1)$$

$$= (R \bowtie S) \cup (R \bowtie \triangle S) - (R \bowtie \nabla S) \cup (\triangle R \bowtie S) \cup (\triangle R \bowtie \triangle S) - (\triangle R \bowtie \nabla S) - (\nabla R \bowtie S) - (\nabla R \bowtie \triangle S) - (\nabla R \bowtie \nabla S) \quad (2)$$

Among the $3^2 = 9$ terms in Equation(2), the first term $(R \bowtie S)$ is the old view. Hence, we do not need to compute the new view entirely from scratch; we can compute the new view by combining the old view with the results from the other eight terms. However, many of these eight terms involve not only the differential files (e.g., $\triangle R$, $\triangle S$, ∇R, ∇S), but the source relations (i.e., the relations in the underlying database). Given that source relations are required, the DW view is not self-maintainable with this approach.

3.3 A Self-Maintainable Approach

Equation(2) can be simplified by exploiting the properties of referential integrity constraints and the nature of the nine terms (e.g., by applying the propagation rules):

- In the Equation, the term $(R \bowtie S)$ represents the old DW view.
- The term $(R \bowtie \triangle S)$ gives an empty relation. Because of the referential integrity constraint, for all $r \in R$, there exists $s \in S$ such that $r.fk = s.ck$. In other words, there does not exist a tuple $s' \in \triangle S$ satisfying $r.fk = s'.ck$.
- All the terms involving ∇S can be grouped together because they basically represent the action that all the tuples containing $s \in \nabla S$ can be deleted.
- Similarly, all the terms involving ∇R can be grouped together because they represent the action that all the tuples containing $r \in \nabla R$ can be deleted.
- The term $(\triangle R \bowtie \triangle S)$ involves only the two differential files $\triangle R$ and $\triangle S$. In other words, no access to the source data is required.
- The term $(\triangle R \bowtie S)$ involves two files, namely the differential file $\triangle R$ and the source relation S. To self-maintain DW views, we need to avoid accessing any source relations.

Recall from Section 2 that when a tuple r is inserted into R, the database checks if there exists a tuple $s \triangle$ such that $s.ck = r.fk$. If such s exists, the insertion is successful and r is then recorded in $\triangle R$. Given that the search has been performed, one can record the tuple s in a file called *RefFile*. By so doing, the *RefFile* contains all those tuples that are related to tuples in $\triangle R$. In other words, the *RefFile* contains all and only those tu-

ples that could be joined with $\triangle R$ in the term ($\triangle R \bowtie S$). Therefore, with this *RefFile*, the term ($\triangle R \bowtie S$) can be rewritten as ($\triangle R \bowtie RefFile_R(S)$), which no longer requires an access to the source data.

DEFINTION 1. *Consider the situation where (i) a SPJ view* $\pi_A\sigma_C(R \bowtie S)$ *is created in terms of two relations R and S, and (ii) a referential integrity constraint is imposed on R and S such that R.fk = S.ck where R.fk denotes the foreign key of the referencing relation R and S.ck denotes the candidate key of the referenced relation S. Then, when a tuple r is successfully inserted into R (i.e., r is put in* $\triangle R$*), a RefFile is created to keep all and only those tuples (in S) that are truly relevant to the update of the view.* □

The following are some nice properties of $RefFile_R(S)$:

- $RefFile_R(S)$ contains all and only those tuples (*in S*) that are truly relevant to joins containing ($\triangle R \bowtie S$).
- $RefFile_R(S)$ can be created without any extra/significant cost (e.g., in searching *S*). One can consider $RefFile_R(S)$ as a "by-product" of checking referential integrity.
- The number of tuples in $RefFile_R(S)$ is bounded above by the number of tuples in $\triangle R$. This is due to the referential integrity constraints. More specifically, because *r.fk = s.ck*, many *r* can reference one *s*. (Of course, each *r* can only reference one *s*.)

With $RefFile_R(S)$, Equation (2) can be simplified to become the following:

$$v' = (R' \bowtie S') = v \cup (\triangle R \bowtie RefFile_R(S)) \cup (\triangle R \bowtie \triangle S) - \nabla R - \nabla S \quad (3)$$

where $v = (R \bowtie S)$ is the old view. Here, for simplicity and readability, we abuse the notion of "set difference" (-): The fourth and the fifth terms ($-\nabla R$) and ($-\nabla S$) represent the deletion of all the tuples containing $r \in \nabla R$ and $s \in \nabla S$, respectively.

It is important to note that, with this self-maintainable approach, we are no longer to access the source data. To elaborate, let us consider all the terms listed in Equation (3). The first term ($R \bowtie S$) is the old DW view. The second term ($\triangle R \bowtie RefFile_R(S)$) uses two differential files, namely $\triangle R$ and $RefFile_R(S)$, where the former contains all insertions into *R* and the latter is a referential integrity differential file (*RefFile*) containing the tuples in *S* that are relevant to $\triangle R$. Regarding the other three terms ($\triangle R \bowtie \triangle S$), ∇R and ∇S, they do not involve the source data; all they need are the differential files, namely $\triangle$R and $\triangle$S (which contain the inserted tuples) as well as ∇R and ∇S (which contain the deleted tuples). To summarize, all the terms in the Equation do not require an access to source data (e.g., *R* or *S*). Consequently, with this approach, DW views can be self-maintained.

EXAMPLE 1. *Consider two relations R and S:* $R(\underline{A}, B) = \{<a1, b1> <a2, b2>\}$, $S(\underline{B}, C) = \{<b1, c1>, <b2, c2>, <b3, c3>\}$, $\nabla R = \{<a2, b2>\}$, $\nabla S = \{<b3, c3>\}$, $\triangle R = \{<a3, b1>\}$, $\triangle S = \{<b4, c4>\}$ and $RefFile_R(S) = \{<b1, c1>\}$. *In this example, when the tuple* <a3, b1> *is inserted into R, its corresponding tuple in S (namely,* <b1, c1>*) is recorded in* $RefFile_R(S)$. *Note that* ($\triangle R \bowtie RefFile_R(S)$) *gives the same result as* ($\triangle R \bowtie S$)*; so, by keeping* $RefFile_R(S)$ *one can compute the join more efficient. To a greater extent, one can efficiently compute the new view* ($R' \bowtie S'$) *by using the old view* ($R \bowtie S$) *with these differential files* ($\triangle R$, $\triangle S$, ∇R, ∇S *and* $RefFile_R(S)$) *according to Equation* (3).

Ⅳ. DW Views Involving Multiple Relations

In the previous section, we described how to self-maintain a DW view involving two relations R and S. Specifically, we showed how a new DW view can be obtained by using only the old view and differential files ($\triangle R$, ∇R, $\triangle S$, ∇S and $RefFile_R(S)$) that is, without accessing the source data. Obviously, self-maintainability is not confined to just two relations. In this section, we show how to self-maintain a DW involving multiple relations.

Here, let us start with a data warehouse view involving three relations (R, S and T). More specifically, we study the following three cases:

- **Case A :** A foreign key of S references both the candidate keys of R and T
- **Case B :** A foreign key of R references a candidate key of S, and a foreign key of S references a candidate key of T
- **Case C :** The foreign keys of R and T reference a candidate key of S

4.1 Data Warehouse Views Involving Three Relations (Case A: R ← S → T)

In this section, we show how a new data warehouse view containing a join over three relations (say, R, S, and T) where a foreign key of S references both candidate keys of R and T can be computed from the old view and the "delta" of the corresponding relations (i.e., without accessing base relations).

As we discussed in Section 3, a new/updated relation R' can be expressed in terms of the old relation R, its insertion $\triangle R$, and its deletion ∇R (i.e., $R' = R - \nabla R \cup \triangle R$). Similar comments can be applied to S and T. So, $(R' \bowtie S' \bowtie T')$ can be expanded as follows:

$$v' = (R' \bowtie S' \bowtie T') = (R - \nabla R \cup \triangle R) \bowtie (S - \nabla S \cup \triangle S) \bowtie (T - \nabla T \cup \triangle T) \quad (4)$$

This equation can be factored into $3^3 = 27$ terms. Fortunately, we can reduce the number of terms in the expression by grouping all terms involving ∇R (and, similarly for those terms involving ∇S as well as ∇T), as we did in Section 3.3. The resulting expression is as follows:

$$\begin{aligned} v' = {} & (R \bowtie S \bowtie T) \cup (R \bowtie S \bowtie \triangle T) \cup (R \bowtie \triangle S \bowtie T) \\ & \cup (R \bowtie \triangle S \bowtie \triangle T) \cup (\triangle R \bowtie S \bowtie T) \\ & \cup (\triangle R \bowtie S \bowtie \triangle T) \cup (\triangle R \bowtie \triangle S \bowtie T) \\ & \cup (\triangle R \bowtie \triangle S \bowtie \triangle T) - \nabla R - \nabla S - \nabla T \end{aligned} \quad (5)$$

Among these $2^3+3 = 11$ terms, some of the joins can be eliminated. For example, the joins $(R \bowtie S \bowtie \triangle T)$, $(\triangle R \bowtie S \bowtie T)$ and $(\triangle R \bowtie S \bowtie \triangle T)$ are not necessary because any joins involving $(\triangle R \bowtie S)$ or $(S \bowtie \triangle T)$ would result in an empty relation. This is due to the referential integrity constraints.

Because of referential integrity constraints, whenever we insert a tuple s into the referencing relation S, we check to see if there exists a corresponding tuple in both referenced relations R and T. So, we keep $\triangle S$, $RefFile_S(R)$ and $RefFile_S(T)$. By using these files, self-maintainability is achieved. More explicitly, Equation (5) can be rewritten as follows:

$$\begin{aligned} v' = {} & (R' \bowtie S' \bowtie T') = v \cup (RefFile_S(R) \bowtie \triangle S \bowtie RefFile_S(T)) \\ & \cup (RefFile_S(R) \bowtie \triangle S \bowtie \triangle T) \cup (\triangle R \bowtie \triangle S \bowtie RefFile_S(T)) \cup (\triangle R \bowtie \triangle S \bowtie \triangle T) - \nabla R - \nabla S - \nabla T \end{aligned} \quad (6)$$

where view = $(R \bowtie S \bowtie T)$ is the old view. It is important to note the following. We do not need

to access the source data, and the new DW view can be recomputed by using only differential files (insertion files, deletion files and *RefFiles*). To elaborate, the first term in Equation (6) is view = ($R \bowtie S \bowtie T$) is the old view. the next four terms use only differential DFs (i.e., $\triangle R$, $\triangle S$ & $\triangle T$ as well as $RefFiles_S(R)$ & $RefFiles_S(T)$). The remaining three terms are deletion files, and they do not require accessing the source data either.

EXAMPLE 2. Consider three relations *R*, *S* and *T*: R (B, D) ={<b1, d1>, <b2, d2>, <b3, d3>}, S (A, B, C) ={<a1, b1, c1>, <a2, b2, c2>}, T (C, E) = {<c1, e1>, <c2, e2>, <c3, e3>}, ∇R = {<b3, d3>}, $\triangle$R = {<b4, d4>}, ∇S = {<a2, b2, c2>}, $\triangle S$ = {<a3, b1, c2>}, ∇T = {<c3, e3>}, and $\triangle$T= {<c5, e5>}, $RefFiles_S(R)$ = {<b1, d1>} and $RefFiles_S(T)$ = {<c2, e2>}. *In this example, when the tuple* <a3, b1, c2> *is inserted into S, its corresponding tuple in R (namely,* <b1, d1>*) is kept in* $RefFiles_S$(T). *Similarly, the tuple in* <c2, e2> $\in$ T *is kept in* $RefFiles_S$(T). *Note that the join* ($RefFiles_S$(R) $\bowtie \triangle$S) *gives the same result as the join* ($R \bowtie \triangle S$)*; the join* ($\triangle S \bowtie RefFiles_S(T)$) *gives the same result as the join* ($\triangle S \bowtie T$). *So, by keeping* $RefFiles_S$(R) *and* $RefFiles_S$ (T), *one can compute the joins more efficient. As a result, one can self-maintain the DW views containing* (R' $\bowtie$ S' $\bowtie$ T') □

4.2 Self-maintenance for a Star Schema

Case A can be further generalized to handle the self-maintenance of DW views containing joins over several relations modeled in the form of a *star schema*. In general, a *star schema*, which is the most common modeling paradigm in the data warehousing environment, contains a *fact table* and several *dimension tables*. These tables are connected in such a way that for each dimension table D_i, there exists a foreign key of the fact table *F* referencing a candidate key of D_i.

It is not difficult to observe that Case A described in Section 4.1 is just a special of this star schema where *S* is the fact table and both *R* and *T* are the dimension tables.

Given a star schema consists of many dimension tables (say, *m* dimension tables), a DW view may contain a join on only *some* (but not all) of these dimension tables. Without loss of generality, let us assume that the DW view contain a join on *n* dimension tables $D_1, \cdots, D_n$ (where $n \leq m$). Then, the new DW view ($F' \bowtie D'_1 \bowtie \cdots \bowtie D'_n$) can be self-maintained as follows. Recall that each relation/table R' in the new DW view can be expressed in terms of its old relation R, its insertion $\triangle$R, and its deletion ∇R, i.e., $R' = R - \nabla R \cup \triangle R$. Therefore, a new view^{viewnew} = ($F' \bowtie D'_1 \bowtie \cdots \bowtie D'_n$) can be expressed as follows:

$$v' = (F' \bowtie D'_1 \bowtie \cdots \bowtie D'_n) = (F - \nabla F \cup \triangle F) \bowtie (D_1 - \nabla D_1 \cup \triangle D_1) \bowtie \cdots \bowtie (D_n - \nabla D_n \cup \triangle D_n) \quad (7)$$

which can be factored into 3^{n+1} terms. As usual, this number can be reduced by grouping the "deletion" terms. As a result, the number of terms in Equation (7) is reduced to (n+1) "deletion" terms (i.e., one "deletion" term for each tables) plus 2^{n+1} "join" terms. Among these 2^{n+1} "join" terms, we observe the following:

- The term ($F \bowtie D_1 \cdots \bowtie D_n$) represents the old DW view.
- Any term containing ($F \bowtie \triangle D_j$), for $1 \leq j \leq n$, results in an empty relation. There are $2^n - 1$ such terms, which contain *F* with at least one $\triangle D_j$.
- The term ($\triangle F \bowtie \triangle D_1 \cdots \bowtie \triangle D_n$) involves on-

ly differential files $\triangle D_j$ (for $1 \leq j \leq n$).

- For the remaining 2^n - 1 terms, they contain $\triangle F, D_i$ and $\triangle D_j$ (for some $1 \leq i, j \leq n$, where i $\neq$ j). To achieve self-maintainability, we replace each occurrence of D_i by $RefFile_F(D_i)$. As a result, we no longer need to access the source data.

In summary, the new DW view can be computed as follows:

$$v' = (F' \bowtie D'_1 \bowtie \cdots \bowtie D'_n) = (F \bowtie D_1 \cdots \bowtie D_n) \cup (\cup \triangle F \bowtie RefFile_F(D_i) \bowtie \triangle D_i) \cup (\triangle F \bowtie \triangle D_i \bowtie \cdots \bowtie \triangle D_n) - \nabla F - \nabla D_1 - \cdots - \nabla D_n \quad (8)$$

Since the star schema is the most common modeling paradigm in the data warehousing environment, our proposed method described here can be very beneficial.

EXAMPLE 3. *Consider a DW view containing a join over a fact table and two of the dimension tables in a star schema. The new DW views can be expressed as follows:*

$$v' = (F' \bowtie D'_1 \bowtie D'_2) = v \cup (RefFile_F(D_1) \bowtie \triangle F \bowtie RefFile_F(D_2)) \cup (RefFile_F(D_1) \bowtie \triangle F \bowtie \triangle D_2) \cup (\triangle D_1 \bowtie \triangle F \bowtie RefFile_F(D_2)) \cup (\triangle D_1 \bowtie \triangle F \bowtie \triangle D_2) - \nabla F - \nabla D_1 - \nabla D_2 \quad (9)$$

By replacing F, D_1, D_2 with S, R, T, it is obvious that the above view is equivalent to that in Equation (6).

4.3 Data Warehouse Views Involving Three Relations (Case B: R → S → T)

In this section, we show how a new DW view containing a join over three relations (say, *R*, *S* and *T*) where (i) a foreign key of *R* references a candidate key of *S* and (ii) a foreign key of *S* references a candidate of *T* can be expressed in terms of the old view and the "delta" of the corresponding relations (i.e., without accessing base relations), and resulting in self-maintainability.

Similar to Case A, a new/updated view ($R' \bowtie S' \bowtie T'$) can be expanded as follows:

$$v' = (R - \nabla R \cup \triangle R) \bowtie (S - \nabla S \cup \triangle S) \bowtie (T - \nabla T \cup \triangle T) \quad (10)$$

This expression can be factored into $3^3 = 27$ terms, and then reduced to $2^3 + 3 = 11$ terms by grouping all the "deletion" terms. A careful analysis reveals that, among these 11~terms, some of the joins can be eliminated. For example, the joins $(R \bowtie S \bowtie \triangle T)$, $(R \bowtie \triangle S \bowtie T)$, $(R \bowtie \triangle S \bowtie \triangle T)$ and $(\triangle R \bowtie S \bowtie \triangle T)$ are not necessary because any joins involving $(R \bowtie \triangle S)$ or $(S \bowtie \triangle T)$ would result in an empty relation. This is due to the referential integrity constraints.

Because of referential integrity constraints, whenever we insert a tuple into the referencing relation *S*, we check to see if there exists a corresponding tuples in both referenced relations *R* and *T*. So, we can easily create $RefFile_S(R)$ and $RefFile_S(T)$. By using these files, Equation (10) can be rewritten as follows:

$$v' = v \cup (\triangle R \bowtie RefFile_R(S) \bowtie T) \cup (\triangle R \bowtie \triangle S \bowtie RefFile_S(T)) - \nabla R - \nabla S - \nabla T \quad (11)$$

However, there is a term in this expression, namely $(\triangle R \bowtie RefFile_R(S) \bowtie T)$, involves *T*. To ensure self-maintainability, we introduce a Con-Diffile.

Basically, when a tuple *r* is inserted into *R*, the database checks to see if there exists a relevant tuple $s \in S$ such that $r.fk = s.ck$. If *s* exists, we

store r in $\triangle R$ and s in $RefFile_R(S)$. Here, we take one additional step. Whenever, we put s in $RefFile_R(S)$, we search for its relevant tuple $t \in T$. (Due to referential integrity constraints, there must exist $t \in T$ such $t.ck = s.fk$.) Notice that this search can be done very efficiently because it is an equality search applied to the candidate key attribute (of T). Once such T is found, we put T in $LiDiF_R(T)$.

DEFINITION 2. *Consider the situation where (i) a SPJ view* $\pi A \sigma C (R \bowtie S \bowtie T)$ *is created in terms of three relations R, S and T, and (ii) a referential integrity constraint is imposed on R, S and T such that R.fk = S.ck and S.fk = T.ck where fk denotes the foreign key and ck denotes the candidate key. Then, when a tuple r is successfully inserted into R, we create* $RefFile_R(S)$. *We perform one extra step: For each s inserts into* $RefFile_R(S)$, *we create a* **linked differential file** $LiDiF_R(T)$ *to keep all and only those tuples (in T) that are truly relevant to* $s \in RefFile_R(S)$. *Here, s.ck = r.fk.* □

There are some nice property of this *LiDiF*, as described below:

① $LiDiF_R(T)$ contains all and only those tuples (in T) that are relevant to the join ($\triangle R \bowtie RefFile_R(S) \bowtie T$))

② $LiDiF_R(T)$ can be created with a minimum cost because the search for relevant T to be put in $LiDiF_R(T)$ can be done very efficiently.

③ The number of tuples in such a $LiDiF_R(T)$ is bounded above by the number of tuples in $RefFile_R(S)$, which in turn is bounded above by the number of tuples in $\triangle R$. This is due to the referential integrity constraints. More specifically, because $r.fk = s.ck$, many r can reference one s but each r can only reference one s. Similarly, because $s.fk = t.ck$, many s can reference one T but each s can only reference one T.

With $LiDiF_R(T)$ storing the relevant tuples in T, Equation (11) can be modified to become to the following, which achieves self-maintainability (i.e., no access of base relations).

$$v' = v \cup (\triangle R \bowtie RefFile_R(S) \bowtie LiDiF_R(T)) \cup (\triangle R \bowtie \triangle S \bowtie RefFile_S(T)) - \nabla R - \nabla S - \nabla T \quad (12)$$

View can be recomputed by using only differential files (insertion files, and/or deletion files) and *RefFiles*. To elaborate, the first term in the above expression is view = $R \bowtie S \bowtie T$ *is the old view. the next four terms use RefFiles with* $\triangle R$ or $\triangle S$. The remaining terms are deletion only, and they do not require accessing the source data either.

EXAMPLE 4. *Consider three relations R, S and T:* $R(\underline{A}, B) = \{<a1, b1> <a2, b2>\}$, $S(\underline{B}, C) = \{<b1, c1>, <b2, c2>, <b3, c3>, <c3, d3>\}$, $\nabla R = \{<a2, b2>\}$, $\nabla S = \{<b3, c3>\}$, $\triangle R = \{<a3, b1>\}$, $\triangle S = \{<b4, c4>\}$, $RefFile_R(S) = \{<b1, c1>\}$, $LiDiF_R(T) = \{<c1, d1>\}$, and $RefFile_S(T) = \{<c2, d2>\}$. *In this example, when the tuple* <a7, b1> *is inserted into R, its corresponding tuple in S (namely,* <b1, c1>) *is kept in* $RefFile_R(S)$. *The* <c1, d1> $\in T$ *is then recorded in* $LiDiF_R(T)$. *Note that the join* ($\triangle R \bowtie RefFile_R(S) \bowtie LiDiF_R(T)$) *gives the same result as the join* ($\triangle R \bowtie S \bowtie T$); *the join* ($\triangle S \bowtie RefFile_S(T)$) *gives the same result as the join* ($\triangle S \bowtie T$). *So, by keeping these differential files (namely,* $RefFile_R(S)$, $LiDiF_R(T)$ *and* $RefFile_S(T)$, *one can compute the joins more efficient. As a result, one can self-maintain the DW views containing* ($R' \bowtie S' \bowtie T'$) □

4.4 Self-Maintenance of Data Warehouse Views for a Snowflake Schema

Case B can be further generalized to handle the self-maintenance of data warehouse views containing joins over several relations modeled in the form of a **snowflake schema**. is a variant of the star schema model; it forms a graph similar to a snowflake-shaped. Specifically, it contains a **fact table** and several **dimension tables.** These tables are connected in such a way that for each dimension table D_i, there exists a foreign key of either the fact table F or another dimension table D_j referencing a candidate key of D_i.

The key difference between the star schema and the snowflake schema is the addition of dimension tables that are referenced by other dimension tables. In this section, we first generalize Case B to handle the path that F references D_i, which in turn references D_j. Once the generalization is established, the technique can be combined with that for the star schema (see Section 4.2)to provide the self-maintainability of DW views containing relations in a snowflake schema.

We start with $v' = (D'_1 \bowtie \cdots \bowtie D'_n)$, which can be factored into 3^n terms. By grouping the "deletion" terms, we end up have $2^n + n$ terms. Among these terms, we observe the following:

- n of these terms represent the deletion of n tables.
- The term $(F \bowtie D_1 \cdots \bowtie D_n)$ is the old DW view
- Any term containing $(D_i \bowtie \triangle D_j)$, for $1 \leq i < j \leq n$, results in an empty relation. There are $2^n - n - 1$ such terms.
- The term $(\triangle D_1 \bowtie \cdots \bowtie \triangle D_n)$ involves only $\triangle D_j$ (for $1 \leq j \leq n$).
- For the remaining $n - 1$ terms, they contain $\triangle D_i$ and D_j (for $1 \leq i < j \leq n$). To achieve self-maintainability, we replace occurrence of D_j by *RefFile* and *LiDiF*. As a result, we no longer need to access the source data.

Since the star schema and the snowflake schema are two common modeling paradigms in the data warehousing environment, our proposed method described here can be very beneficial.

4.5 Data Warehouse Views Involving Three Relations (Case C: R → S ← T)

In this section, we show a different case. Specifically, we show how a new DW view involving three relations (say, R, S and T) where foreign keys of both R and T reference a candidate key of S can be expressed as an aggregate view from old view with the delta of the corresponding relations (i.e., without accessing base relations), and resulting in self-maintainability. Similar to previous two cases (Cases A and B), we start with $3^3 = 27$ terms, which can be reduced to $2^3 + 3 = 11$ terms. A careful analysis on these terms reveals that some of the terms can be eliminated. For example, the joins $(R \bowtie \triangle S \bowtie T)$, $(R \bowtie \triangle S \bowtie \triangle T)$ and $(\triangle R \bowtie \triangle S \bowtie T)$ are not necessary because any joins involving $(R \bowtie \triangle S)$ or $(\triangle S \bowtie T)$ would result in an empty relation. This is due to the referential integrity constraints. Because of the referential integrity constraints, whenever we insert a tuple into the referencing relation R or T, we check to see if there exists a corresponding tuples in the referenced relation S. So, we keep $\triangle R$, $\triangle T$, $RefFile_R(S)$ and $RefFile_T(S)$. By using these files, the new DW view can be rewritten as follows:

$v' = v \cup (R \bowtie RefFile_T(S) \bowtie \triangle T) \cup (\triangle R \bowtie RefFile_R(S) \bowtie T) \cup (\triangle R \bowtie [RefFile_R(S) \cap RefFile_T(S)] \bowtie \triangle T) - \nabla R - \nabla S - \nabla T$ (13)

However, there is two terms in Equation (13), namely $(R \bowtie RefFile_T(S) \bowtie \triangle T)$ and $(\triangle R \bowtie RefFile_R(S) \bowtie T)$, involve the source data R and T. To ensure self-maintainability, we introduce a **ReDiFile**. Basically, when a tuple r is inserted into R, the database checks to see if there exists a relevant tuple $s \in S$ such that $r.fk = s.ck$. If s exists, we store r in $\triangle R$ and s in $\text{ReDiF}_R(S)$. Given that there are two relations referencing S, we only know which tuples in R (or $\triangle R$) references the tuples in $\text{ReDiF}_R(S)$, but we do not know which tuples in T references the tuples in $\text{ReDiF}_T(S)$. Here, we take one additional step to get this information. Whenever, we put s in $\text{ReDiF}_R(S)$, we search for its relevant tuple t *in* T (i.e., find $t \in T$ such that $t.fk = s.ck$). Such a search is similar to that of finding the tuples in T that reference s when one wants to delete s for the **on delete cascade**, **on delete set NULL** and **on delete set default** options). If such T is found, we put T in $BkDiF_R(T)$. There is a nice property of this *BkDiF*:

① $BkDiF_R(T)$ contains all and only those tuples that are relevant to the join $(\triangle R \bowtie \text{ReDiF}_R(S) \bowtie T)$. Similarly, we can create $\boldsymbol{BkDiF}_T(R)$ for dealing with the join $(R \bowtie \text{ReDiF}_R(S) \bowtie \triangle T)$. With the use of $\boldsymbol{BkDiF}_R(T)$ and $BkDiF_T(R)$, which store the relevant tuples in T and R, respectively. Equation (13) can be modified to become to the following to achieve self-maintainability (i.e., no access of base relations).

$v' = v \cup (\triangle R \bowtie \text{ReDiF}_R(S) \bowtie BkDiF_{R,S}(T)) \cup (LiDiF_{T,S}(R) \; R \bowtie \text{ReDiF}_R(S) \bowtie \triangle T) \cup (\triangle R \bowtie \triangle S \bowtie \text{ReDiF}_S(T)) - \nabla R - \nabla S - \nabla T$ (14)

EXAMPLE 5. *Consider three relations R, S and T: R* (A, B) ={<a1, c1> <a2, c2>}, *S* (C, D) = {<c1, d1>, <c2, d2>, <c3, d3>}, T(B, C) = {<b1, c1>, <b2, c2>}, ∇R = {<a2, c2>}, $\triangle R$ = {<a4, c1>}, ∇S = {<c3, d3>}, $\triangle S$ = {<c5, d5>}, ∇T = {<b2, c2>}, $\triangle T$ = {<b5, c2>}, $\text{ReDiF}_R(S)$ ={<c1, d1>}, $BkDiF_R(T)$ = {<b1, c1>}, $BkDiF_T(R)$ = {<a2, c2>}, and $\text{ReDiF}_T(S)$ = {<c2, d2>}. □

4.6 Self-Maintenance of Data Warehouse Views for a Galaxy Schema

The above case can be further generalized to handle the self-maintenance of data warehouse views that are modeled in the form of a fact constellation schema (or a galaxy schema). In this schema, multiple fact tables share dimension tables. This situation may occur in some sophisticated applications.

In this section, we first generalize the three-relation case to handle the path that F references D_i, which in turn references D_j. Once the generalization is established, the technique can be combined with that for the star schema to provide the self-maintainability of DW views modeled in a snowflake schema.

Here, we assume that the join involves one dimension table and n fact tables. Similar to previous cases, we analyze 2^{n+1} terms:

- The term $(D \bowtie F_1 \cdots \bowtie F_n)$ is the old DW view
- Any term containing $(\triangle D \bowtie F_j)$, for $1 \le j \le n$, results in an empty relation. There are $2^n - 1$ such terms, which contain $\triangle D$ with at least one F_j.
- The term $(\triangle D \bowtie \triangle F_1 \cdots \bowtie \triangle F_n)$ involves only $\triangle F_j$ (for $1 \le j \le n$).
- For the remaining $2^n - 1$ terms, they contain D, F_i and $\triangle F_j$ (for some $1 \le i, j \le n$, where $i \ne j$).

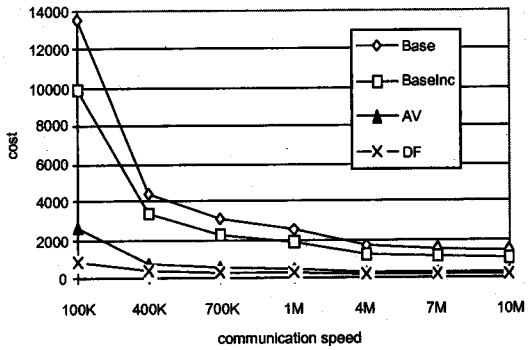
To achieve self-maintainability, we replace each occurrence of F_i by *BkDiF*. As a result, we no longer need to access the source data.

Since the star schema, the snowflake schema, and the galaxy schema are common modeling paradigms in the data warehousing environment, our proposed method described here can be very beneficial.

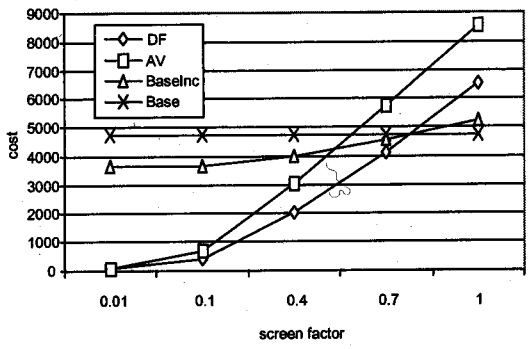
In the previous section, we showed how we self-maintain data warehouse views involving three relations. In real-world applications, it is not unusual that more than three relations as involved (and connected). Here, we show how a new data warehouse view involving k relations (say, $R_1, \cdots, R_k$), where a foreign key of R_I references a candidate key of R_{I+1} (for all i from 1 to k-1), can be expressed as an aggregate view from old view with the delta of the corresponding relations (i.e., without accessing base relations), and resulting in self-maintainability.

V. Performance Analysis

The following values are assigned to the parameters for the analysis. The block size is generally assumed to be 4,000 bytes, and the I/O cost 25 ms/block. The record sizes of the base tables are assumed to be the same as 200 bytes and 220 bytes, respectively. In the deleted case, only the identifier will be sent, i.e., 20 bytes. The cardinalities of the base tables are assumed to be examined from 1,000 and 10 Tera tuples respectively, and the size of the differential file is varied in the experiment. The communication speed varies from a very low case to a high - speed case 100Kbps~10Mbps. Tuples are filtered from a no screening case (1.0) and a highly screened case (0.001). According to the above parameters and cost functions presented in Appendix, the following four methods are analyzed: (1) the base table method (Base), (2) the base table incremental method (BaseInc), (3) the auxiliary view method (AV), and (4) the differential file method (DF).



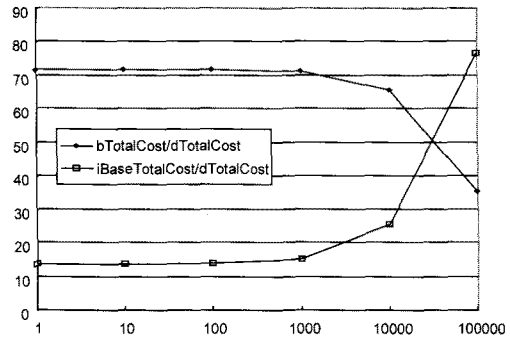
<Figure 2> Cost traverses of Base, BaseInc, AV, and DF with changing communication speed



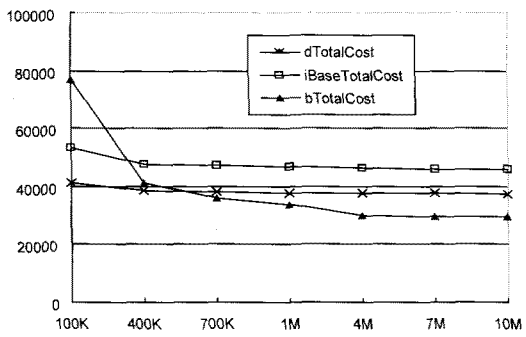
<Figure 2> Cost traverses of Base, BaseInc, AV, and DF method with changing screen factors

A total of eight figures are presented with respect to various criteria. <Figure 1> and <Figure 2> show that the total costs of the three methods are strongly dependent on both the selectivity (with the screen factor) and the communication speed. <Figure 1> represents that the costs of all

methods are decreased along with communication speed, and the cost of DF is consistently less than .10 of base method in any communication speed. This shows that the size of the data per se is the most critical factor. If the tuples are filtered highly (up to about 0.01), as in <Figure 2>, the Base method and the BaseInc method are less advantageous than the DF method or the AV method.



<Figure 3> Cost Ratios of Base method and BaseInc method over DF method with increasing view size



<Figure 4> Cost Ratios of Base and BaseInc, and DF method with increasing communication speed

<Figure 3> and <Figure 4> represent the cost ratios of Base method and BaseInc method over DF method with respect to increasing view sizes. Basically the Base method is unwavering according to the changes of DF, because it does not use the DF. But the other methods are changed and it is natural that the DF method is the most highly affected. We allow an apparent disadvantage to the DF method if we treat the DF as individual tuples twice at a time. Since the tuple in the DF has additional attributes such as operation and sysdate, an additional operation is required to merge a differential tuple with the relevant base relation [Khan and Mott, 2002; Liu et al., 1998]. It can be said that the DF method scheme is preferable to the other methods if the size of the DF is about .70 less than that of the base table. We can expect that the DF method is appropriate to a DW environment in its huge data volume. As for the quantity of the object, it can be concluded that the sheerer the volume, the more advantageous is the DF method.

Ⅶ. Conclusion

DW views provide an efficient access to information that is normally integrated from various sources. As changes are made to the source data, the corresponding DW views may be outdated. Hence, the maintenance of DW views is crucial for the currency of information. In this paper, we proposed a novel method to self-maintain the DW views that contain a SPJ view over multiple relations. Specifically, we exploit the referential integrity constraints imposed on the relations in the underlying databases (or data sources) because these constraints are the most fundamental ones that any relational databases should satisfy. With our proposed method, DW views can be updated by using only the old views and the differential files such as insert/delete differ-

ent files ($\triangle$R, ∇R), referential integrity differential files (*RefFile*), linked differential files (*LiDiF*), and backward-linked differential files (*BkDiF*), that keep the truly relevant tuples in the delta. This method avoids accessing the underlying databases in that the method achieves self-maintainability even in preparing auxiliary information. The proposed method can be applicable to the self-maintenance of DW views that contain joins over multiple sources.

References

[1] Babcock B., Babu S., Datar M., Motwani R., Widom J., "Models and Issues in Data Stream Systems," *Proceedings of the ACM PODS*, 2002, pp. 1-16.

[2] Berndt D., Hevnera A. and Studnicki J., "The Catch data warehouse: support for community health care decision-making," *Decision Support Systems*, Vol. 35, No. 3, 2003, pp. 367-384.

[3] Blakeley J., Larson P. and Tompa P., "Efficiently updating materialized views," *Proceedings of the ACM SIGMOD*, 1986, pp. 61-71.

[4] Ceri S., Cochrane R. J. and Widom J., "Practical Application of Triggers and Constraints: Successes and Lingering Issues," *Proceedings of the Very Large Data Bases*, 2000, pp. 254-262.

[5] Chari K., "Model composition in a distributed environment," *Decision Support Systems*, Vol. 35, No. 3, 2003, pp. 399-413.

[6] Espil M. and Vaisman A., "Revising aggregation hierarchies in OLAP: a rule-based approach," *Data and Knowledge Engineering*, Vol. 45, No. 2, 2003, pp. 225-256.

[7] Griffin T. and Libkin L., "Incremental maintenance of views with duplicates," *Proceedings of the ACM SIGMOD*, 1995, pp. 328-339.

[8] Griffin T., Libkin L. and Trickey H., "An improved algorithm for the incremental recomputation of active relational expressions," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 9, No. 3, 1997, pp. 508-511.

[9] Gupta H. and Mumick I. S., "Selection of views to materialize under a maintenance cost constraint," *Proceedings of the ICDT*, 1999, pp. 453-470.

[10] Hyun N., "Multiple-view self-maintenance in data warehousing environments," *Proceedings of the Very Large Data Bases*, 1997, pp. 26-35.

[11] Khan S. and Mott P. L., "LeedsCQ: a scalable continual queries system," *Proceedings of the DEXA*, 2002, pp. 607-617.

[12] Lakshmanan L., Pei J. and Zhao, Y., "QC-Trees: An Efficient Summary Structure for Semantic OLAP," *Proceedings of the ACM SIGMOD*, 2003, pp. 64-75.

[13] Laurent D., Lechtenbrger J., Spyratos N. and Vossen G., "Monotonic complements for independent data warehouses," *The VLDB Journal*, Vol. 10, No. 4, 2001, pp. 295-315.

[14] Lechtenborger J. and Vossen G., "On the computation of relational view complements," *ACM TODS*, Vol. 28, No. 2, 2003, pp. 175-208.

[15] Lechtenborger J., "The impact of the constant complement approach towards view updating," *Proceedings of the ACM PODS*, 2003, pp. 49-55.

[16] Lee Wookey and Geller J., "Semantic Hierarchical Abstraction of Web Site Structures for Web Searchers," *Journal of Research and Practice of Information Technology*, Vol. 36, No. 1, 2004, pp. 71-82.

[17] Lee Wookey, Hwang Y., Kang S., Kim S., Kim C. and Lee Y., "Self-maintainable data warehouse views using differential files," *Proceedings of the DEXA*, 2002, pp. 216-225.

[18] Liu L., Pu C., Tang W., Buttler D., Biggs J., Zhou T., Benninghoff P., Han W. and Yu F., "CQ: a personalized update monitoring toolkit," *Proceedings of the ACM SIGMOD*, 1998, pp. 547-549.

[19] Markowitz V., "Safe referential integrity and null constraint structures in relational databases," *Information Systems*, Vol. 19, No. 4, 1991, pp. 359-378.

[20] Mohania M. and Kambayashi Y., "Making aggregate views self-maintainable," *Data and Knowledge Engineering*, Vol. 32, No. 1, 2000, pp. 87-109.

[21] Nematia H., Steiger D., Iyer L. and Herschel R., "Knowledge warehouse: an architectural integration of knowledge management, decision support, artificial intelligence and data warehousing," *Decision Support Systems*, Vol. 33, No. 2, 2002, pp. 143-161.

[22] Park C., Kim M. and Lee Y., "Finding an efficient rewriting of OLAP queries using materialized views in data warehouses," *Decision Support Systems*, Vol. 32, No. 4, 2002, pp. 379-399.

[23] Piccinelli G., Finkelstein A. and Costa T., "Flexible B2B processes: the answer is in the nodes," *Information and Software Technology*, Vol. 45, No. 15, 2003, pp. 1061-1063.

[24] Pourabbas E. and Shoshani A., "Answering Joint Queries from Multiple Aggregate OLAP Databases," *Data Warehouse and Knowledge Discovery*, 2003, pp. 24-34.

[25] QuassD., Gupta A., Mumick I. and Widom J., "Making views self-maintainable for data warehousing," *Proceedings of the PDIS*, 1996, pp. 158-169.

[26] Quian X. and Wiederhold G., "Incremental recomputation of active relational expressions," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 3, No. 3, 1991, pp. 337-341.

[27] Sifer M., "A visual interface technique for exploring OLAP data with coordinated dimension hierarchies," *Proceedings of the CIKM*, 2003, pp. 532-535.

[28] Sismanis Y., Deligiannakis A., Kotidis Y. and Roussopoulos N., "Hierarchical dwarfs for the rollup cube," *Proceedings of the DOLAP*, 2003, pp. 17-24.

[29] Theodoratos D. and Bouzeghoub M., "A general framework for the view selection problem for data warehouse design and evolution," *Proceedings of the DOLAP*, 2000, pp. 1-8.

[30] Theodoratos D., Ligoudistianos S. and Sellis T. K., "View selection for designing the global data warehouse," *Data and Knowledge Engineering*, Vol. 39, No. 3, 2001, pp. 219-240.

[31] Trujillo J., Lujan-Mora S. and Song I., "Applying UML For Designing Multidimensional Databases And OLAP Applications," *Advanced Topics in Database Research*, Vol. 2, 2003, 13-36.

[32] Wixom B. and Watson H., "Data Warehousing and Business Intelligence," Minitrack Introduction, *Proceedings of the HICSS*, 2004.

[33] Zhuang S., Fong S. and Chan S., "Data Mining on Users' Access Trials for Web Business Intelligence," *Neural Networks and Computational Intelligence*, 2003, pp.13-18.

[34] Zhuge Y., Garcia-Molina, Hammer H. J. and Widom J., "View maintenance in a warehousing environment," *Proceedings of the ACM SIGMOD*, 1995, pp. 316-327.

◆ 저자소개 ◆

이우기 (Lee, Wookey)
서울대학교 산업공학과에서 학사, 석사 및 박사학위를 취득한 후 성결대학교 컴퓨터학부 부교수로 재직중이며 한국과학재단의 지원으로 캐나다UBC(2002~2003)에 교환교수를 하였다. 그의 논문은 *Journal of Database Systems, Int'l Journal of Computer Information Systems, Journal of Research and Practice of Information Technology* 등의 국제저널과 경영정보학연구, 경영과학회논문지, 한국정보과학회논문지 등 국내저널에 게재되었으며, 주요 연구 관심분야로는 데이터웨어하우스 및 데이터모델링, Web Structure Mining, 그리고 Business Model 등이다.

◈ 이 논문은 2004년 1월 26일 접수하여 1차 수정을 거쳐 2004년 7월 21일 게재확정되었습니다.