Deriving Differential Auxiliary Information for Self-maintainable Views

*Wookey Lee, Myung Keun Shin**

Sungkyul University, Anyang 8 dong, Kyongkido, Korea
*Graduate School of Management, KAIST, Dongdaemun-gu, Seoul 130-012, Korea

Summary
We present an innovative method of deriving an auxiliary view from the running databases. During view maintenance, our method minimized to accesses base relations as well as not to re-execute the view definition again. Using this approach, long transactions from the materialized view maintenance intermingled with database transactions can be committed without fear of abort. We use a formal algebraic approach and develop relevant theorems and proofs. For comparison purposes, four corresponding methods are compared: the bottom line base relation method (Base), incremental base relation method (BaseInc), auxiliary view method (AV), and differential file method (DF). We also consider the handicapped cost of storing information as we examine the worst as well as the best case scenarios of our method. Experimental analyses show that the DF method is superior to the other methods in a large data environment of up to 10 Terabyte tuples of relations. The results show that compared with the other methods, the DF method can considerably reduce the number of IO's. Our most important finding is that the DF method can successfully update the aggregate SPJ views (practically) self-maintainable in the tuple level as well as (theoretically) is independent of the DBMS.

Key words:
Referential Integrity, View Maintenance, Differential Files, Self-maintainability.

1. Introduction

Views are materialized to provide fast access to information that is usually integrated from several distributed data sources. One of the critical weaknesses of materialized views (MV) is that the views are liable to become outdated or desynchronized with the source data as changes are made to the source data upon which the views are defined. In order to guarantee the correctness (or currency) of the MV, all changes to source data have to be applied to the views. Many studies have been extensively undertaken on what is called the view maintenance problem or the materialized view update [13, 14, 20].

In response to source changes, a view can be either recomputed from the source data or maintained incrementally without accessing the source data (called self-maintenance).

We propose an innovative view update algorithm that does not access base relations. We use a differential file and an Auxiliary Integrated File to fully maintain materialized views without accessing base relations. The DF is the changed portion of a base relation, which is sometimes called the delta of a base relation. The AIF is defined as auxiliary information derived from a base relation through the referential integrity constraints between relevant base relations.

The rest of the paper is organized as follows. Section 2 discusses the related works. Section 3 introduces a formal approach to materialized view maintenance. The formal algebra is in section 4. We present cost functions with the parameters and performance analyses in section 5, and section 6 concludes our paper.

2. Related Works

Self-maintenance is a notion that can be defined as maintaining views by materializing supplementary data so that the view can be maintained without (or at least mostly without) accessing base relations. The notion was originally introduced by Blakeley [2]. 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. Blakeley's algorithm is a special case of the counting algorithm applied to select-project-join expressions (no negation, aggregation, or recursion). Theodoratos et al. [20] summarize issues extensively related to self-maintainability, and suggest a view selection approach based on a DAG (Please write out the whole word DAG then put abbreviation in parentheses) 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 [11, 18].

Algebraic approaches for maintaining materialized views are discussed in [2, 19, 7, 11, 9, 20]. Excluding

conventional database approaches, [2], Quian & Widerhold [19] present an algorithm for incremental view maintenance based on finite differencing techniques (later corrected in [8]). The algorithm 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. They considered two types of operations: insertions and deletions. However, the algorithm uses source relations and thus it lacks the self-maintenance notion. Griffin & Libkin in [7] extend the techniques in [19]. [11] proposes to include functional dependencies. [9] integrates outer joins. These references do not consider the concepts of referential integrity for the maintenance of materialized views. In this paper, some of the common notations (mainly from [19], [7], and [8]) are extended to present some propagation rules for materialized views based on referential integrity constraints.

There has been some research that considers the database system as a rule system [6, 5]. Widom et al. [5] (enhanced from [6]) present a comprehensive survey on the roles played in materialized 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.') Though not directly related, there are several works [16, 10, 11] 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, [16] presents the run time execution problem and the safeness condition respectively. [15] investigates the view maintenance problem with inclusion dependency but no referential integrity rules.

Entity and referential integrity rules are the most fundamental constraints that any relational database should satisfy [15]. The entity integrity rule starts from the selection of a candidate key and referential integrity starts from the selection of a foreign key. Codd's definition of referential integrity is 'No component of a foreign key is allowed to have an I-maked value,' where an I-marked value means a null value of the type 'value does not exist,' or ' value at present exists but is unknown,' or 'value is inapplicable'.

Database rules, including referential integrity constraints, are utilized in maintaining materialized views in several articles such as [17, 18]. Quass et al. [18] and Mohania et al. [17] use the referential integrity constraint to determine whether a base relation is participating in the views, and [18] extends the works of [19] and [7] to transform change propagation equations into more efficient ones. They use an auxiliary view (in [17] 'auxiliary relation', in [12] 'auxiliary data', and 'complements' in [14, 15]) in order to maintain a select-project-join (SPJ) view without accessing base relations at the sources. 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.

In applying referential integrity to view maintenance, the work of Quass et al. [18] (called, 'AV method' in this paper) is slightly related to our approach, but there are several differences. While Quass et al. assume that the structure of the referential integrity conditions should be a tree; our approach does not assume it. Thus in the AV method, any kind of cycles in a database schema, including self-join, cannot be supported. For example, a transitive closure algorithm [18] cannot support the schema in Section 2. Another difference is that it can be said that the AV method uses the rule in a macroscopic way (i.e., using the rule to find a corresponding 'relation'), but our method uses it in a microscopic way (i.e., using it to find a corresponding 'tuple' in the relation). This is one of the innovative features of our approach that differentiates it from others.

3. Algebraic Representation

It is assumed that the materialized view in this paper is not from independent relations, but from referentially integrated relations. Thus, a relation has an attribute called a foreign key such that it has a referential integrity condition with the key of some relation. (We will call it a RI condition or a RI constraint.) In that case, one relation is called a referenced relation (e.g., DEPT) and the other a referencing relation (e.g., EMP). The referencing relation can be called a fact table and the referenced relation a dimension table. From this point of view, they are called a star schema or a snowflake schema in the materialized environment. Let s represent a referencing relation and r be a referenced relation respectively throughout this Section. The condition of the RI is typically assumed to have one of the representative RI conditions such as restrict, cascade, and nullify [16]. We have a focus on the RI condition that can also be extended to a modified form or a nested form.

We explain the view updates in terms of the changes to base relations and how these changes affect other relations. They are classified as follows: the insertion in the referenced relation case, the deletion in the referenced relation case, the deletion in the referencing relation case, and the insertion in the referencing relation case, including these changes with base relations. The orders are not significant. The last case is the most complicated one to solve. Thus a new schema called an AIF in terms of referential integrity conditions is presented.

The following notations and algebraic expressions on the data base schema R_i are introduced:

R_i := Ø empty set
$|\sigma_p(r)$ selection with condition P
$|\prod_A(r)$ projection over an attribute A
$|r \oplus r$ disjoint union
$|r \ominus r$ contained difference
$|r \bowtie r$ natural join

Definition 1 Let a tuple t defined on a relation r_i for $i \in \psi$ be represented as $t(r_i)$. An inserted tuple $\triangle t(r_i)$ and a deleted tuple $\triangledown t(r_i)$ belong to the insertion $\triangle r_i$ and the deletion $\triangledown r_i$ of the relation r_i respectively. It can be specified by a *key* and by a *foreign key* (*fk*) as $t(r_i.key)$and $t(r_i.fk)$ respectively. □

Definition 2 A differential file $DF(r_i)$ of a base relation r_i $=\{key, a_{i1}, a_{i2}, \ldots, a_{im}\}$ can be defined as $\{key, a_{i1}, a_{i2}, \ldots, a_{im}, operation, sysdate\}$ for $i, m \in \psi$, where *operation* is an operation type having two possible values i.e. 'insert', 'delete'. A modification is a deletion and an insertion in series having the same *sysdate*. The *sysdate* is the timestamp recorded by a committed transaction to the base relation r_i. Then the differential file consists of insertions and deletions represented algebraically as $DF(r_i) = \triangle r_i \ominus \triangledown r_i$. □

Example 4.1 Let's consider the instance of relations *Pt, Su*, and *Sp* defined as those in Section 2. The corresponding differential files of *Pt* and of *Sp* (i.e., *DF*(*Pt*) and *DF*(*Sp*)) are respectively as shown below. There are no changes in the other tables. The first tuple of *DF*(*Pt*), say {P4, printer, red, 300, delete, 10/08/06}, represents that a 'printer' with price '300' supplied by the manufacturer 'red' was deleted at time '10/08/06'. Similarly, in the *differential file* of *Sp DF*(*Sp*), we can tell that the first and the second tuples indicate that the product 'P1' supplied by 'S1' was modified at time '05/08/06'. Then a product 'P6' supplied by 'S4' was inserted.

Table 4.1: Differential file *DF*(*Sp*)

pk	sk	ok	qty	sprice	operation	sysdate
P1	S1	O1	10	2500	delete	05/08/06
P1	S1	O1	20	2800	insert	05/08/06
P6	S4	O2	40	3000	insert	08/08/06

Table 4.2: Differential file *DF*(*Pt*)

pk	pn	mfr	price	operation	sysdate
P4	printer	red	300	delete	10/08/06

Corollary 3 A new base relation (after image) r_i^{new} can be expressed as the base relation (old image) r_i and its $DF(r_i)$ as follows: $r_i^{new} = r_i \oplus DF(r_i) = r_i \oplus(\triangle r_i \ominus \triangledown r_i) = (r_i \oplus \triangle r_i \ominus \triangledown r_i)$ for $i \in \psi$ □

Notice that the schema adjustment between a base relation and the corresponding DF is assumed to be exchangeable in this paper. Since the *DF* has two more columns (i.e., *operation* and *sysdate*) than the corresponding base relations. For simplicity, we do not create an additional operator to adjust between them. In this paper, we follow the notation used in previous research [7, 8, 18, 19].

Example 4.2 The new relation Pt^{new} is derived from *Pt* (in example 2.1) and with *DF*(*Pt*) in Table 4.1 as:
$Pt^{new} = Pt \oplus DF(Pt) =$ {(P1, computer, red, 2000), (P6, DVD, yellow, 200), (P7, computer, green, 3000)}

Definition 4 A relation r_i satisfies a *RI condition*, which can be represented as: $r_i^{\mathrm{RI}\bullet[condition]}$, where the superscripted [*condition*] can optionally represent a RI condition detail. □

For example, an insertion in *s* due to an RI condition of a deletion in a relation *r* can be represented: $\triangle s^{RI\bullet[\triangledown r]}$.

Example 4.3 Note that there is an RI condition between the relations *Cu* and *Or* in section 2. If the RI condition is 'On delete Cascade' and a deletion in *Cu* (e.g., 'C2') may affect the relation *Or*, then the corresponding tuple in *Or*, (e.g, O2, C2, 5, 2500) should also be deleted. It can be represented as follows:
$\triangledown Or^{RI\bullet[\triangledown Cu]} =$ {(O2, C2, 5, 2500, delete, 05/08/06)}

A change in a relation can cause other changes in turn due to the *referential integrity* constraints. The nested changes of base relation can be represented in general as follows: In that case, the changes can be represented as a nested form in terms of *RI* conditions.

Corollary 5 A change in the *referenced relation* (r_1) can generate a change due to the *RI condition* to the corresponding *referencing relation* (r_2) that results, in turn, in a change recursively to r_n and finally it may effect a change in the relation *s*. The generalized deletion and insertion due to *RI condition* in *s* can be represented respectively:
$\triangledown s^{RI\bullet[\triangledown rn \bullet\bullet\bullet[\triangledown r2\bullet[\triangledown r1]]]}$
and $\triangle s^{RI\bullet[\triangledown rn \bullet\bullet\bullet[\triangledown r2\bullet[\triangledown r1]]]}$ □

Example 4.4 Note that there also is an *RI* condition between the relations *Or* and *Sp* in section 2. In addition to the above example, a deletion in *Or* will also affect relation *Sp*, thus the corresponding tuple, i.e., (P1, S4, O2,

5, 2500) should also be deleted. Therefore in order to (successfully) delete a tuple in *Cu*, two more deletes in *Or* and in *Sp* are needed: $\triangledown Sp^{RI\bullet[\triangledown Or \bullet[\triangledown Cu]]} = \{$(P1, S4, O2, 5, 2500, delete, 05/08/06)$\}$

Note that by the RI conditions, the cascading RI representations of a deletion and an insertion are one of the DF tuples of the corresponding relation:

$\triangledown s^{RI\bullet[\triangledown rn \bullet\bullet\bullet[\triangledown r2\bullet[\triangledown r1]]]} \subseteq \triangledown s$
and $\triangle s^{RI\bullet[\triangledown rn \bullet\bullet\bullet[\triangledown r2\bullet[\triangledown r1]]]} \subseteq \triangle s$

For example, there are the two more deletes in the above examples (4.3 and 4.4) that did not originate from their own transactions, the deleted tuples are included among the DFs of their own, i.e., *DF*(*Or*) and *DF*(*Sp*) respectively.

Theorem 6
$\triangle r^{RI}{}_{\bullet}{}^{[\triangledown s]} = \triangledown r^{RI}{}_{\bullet}{}^{[\triangledown s]} = \triangle r^{RI}{}_{\bullet}{}^{[\triangle s]} = \triangledown r^{RI}{}_{\bullet}{}^{[\triangle s]} = \varnothing$

Proof. If there is no change in a *referencing relation s*, i.e., $\triangle s = \triangledown s = \varnothing$, then the above equations trivially hold. If there exists an insert transaction in *s*, then it will be committed if the relevant *key* exists in the *referenced relation r*, unless the transaction will be aborted. In the two cases, there is no change due to RI in the relation *r*. If there is a delete transaction in *s*, then the transaction commits without an inquiry to *r*. Therefore no changes happen in *r* due to the *RI* by the changes of *s*. □

From the above theorem, we can get the following trivial result.
Corollary 7 $\triangledown s \bowtie r \rightarrow \triangledown s$ □

Theorem 8 A deletion in *r* triggers a change in *s* by the *RI condition*. Then the following rule holds:
$(s - (\triangle s^{RI\bullet[\triangledown r]} \ominus \triangledown s^{RI\bullet[\triangledown r]})) \bowtie \triangledown r \rightarrow \varnothing$

Proof: If there is no deletion in the referenced relation *r*, then the above holds. Suppose that there exists a deletion but there exists no *insertion by RI* or no *deletion by RI* in a *referencing relation s*. Further suppose that there exist some tuples in the *referencing relation* corresponding to the deleted tuple of the *referenced relation*. This is a *referential integrity* violation. It contradicts the above assumption. Thus, as long as there remains a foreign key in a *referencing relation*, there will be two cases with respect to the *RI* condition: (1) If the *RI* condition is '*restrict*', then the deletion in the *referenced relation* cannot be committed, (which means $\triangledown r = \varnothing$). (2) If the *RI* condition is '*cascade*' or '*nullify*', then the changes caused by the *RI* will occur in the *referencing relation s* (say, $\triangle s^{RI} \oplus \triangledown s^{RI}$). This represents that the above rule holds. □

Example 4.4 $(Or \ominus \triangledown Or^{RI\bullet[\triangledown Cu]}) \bowtie \triangledown Cu = \varnothing$

Theorem 9 $s \bowtie \triangle r = \varnothing$

Proof: For some tuples t_1 such that $t_1(\triangle r.key) = s.fk$, if $\exists t$ such as $\exists t_2$, $t_2(\triangle r.key) = t_1(r.key)$, this violates uniqueness of key. □

Theorem 10 $s \bowtie \triangledown r \rightarrow \triangledown s \bowtie \triangledown r$

Proof: $s \bowtie \triangledown r \rightarrow (s^{new} \oplus \triangle s \ominus \triangledown s) \bowtie \triangledown r \rightarrow (s^{new} \bowtie \triangledown r) \oplus (\triangle s \bowtie \triangledown r) \ominus (\triangledown s \bowtie \triangledown r)$
$\rightarrow \triangledown s \bowtie \triangledown r$ The proof is based on *Corollary 3*, and *Theorem 4.2* respectively. □

Therefore, for all the *RI* conditions, the join of the deleted tuple(s) (i.e., $\triangledown r$) in the *referenced relation* with the tuples in *referencing relation s* should be Null or equivalent to the deltas.

Example 4.5 Suppose that a product 'P6' is not to be deployed in the previous example, i.e., a delete transaction for the tuple {P6, DVD, yellow, 200} is issued at time 01/09/06. The transaction will then be executed in terms of the *RI* conditions:

1) If the *RI* condition is 'On delete restrict' on the *key* of *Pt*, i.e., $t(\triangledown Pt)^{RI\bullet['restrict']}$, the transaction will be aborted due to RI in *Sp*.
2) If the *RI* condition is 'On delete cascade' on the *key* of *Pt*, i.e., $t(\triangledown Pt)^{RI\bullet['cascade']}$, it will delete a tuple in *Sp* (i.e. {P6, S1, O3, 20, 250} will be deleted).
3) If the *RI* condition is 'On delete nullify' on the *key* of *Pt*, i.e., $t(\triangledown Pt)^{RI\bullet['Nullify']}$, it will modify another tuple in *Sp* as {P6, S1, O3, 20, 250, delete, 01/09/06} and {NULL, S1, O3, 20, 250, insert, 01/09/06}are appended to *DF*(*Sp*) in series.

Note that the last case ('On delete nullify') leaves something (the nullified tuple) in the base relation (*Sp*). However, the tuple left behind is null-valued, and thus, the join yields nothing.

If there is an insertion in the *referencing relation*, there exists a sort of integrity function of *RI check* among *referential integrity* constraints to acknowledge the insertion to the *referenced relation*. The ***RI check*** can be found between the *foreign key* of the *referencing relation* (say, *s.fk*) and the *key* of the *referenced relation* (say, *r.key*).

Definition 11 An ***RI check*** condition is a Boolean function that there is an insertion transaction in the *referencing relation s*, and there follows an *RI check* to see if the *foreign key* in *s* corresponds to the *key* of the *referenced relation r*. The ***RI check*** condition is:

$t_1(\triangle s.fk^{RI\bullet[check]})$ *is true, if* $\exists t_2$, $t_1(\triangle s.fk) = t_2(r.key)$, *o/w, false* □

Lemma 12 An insert transaction in the *referencing relation* is committed, the ***RI check*** condition corresponding to the insertion is true.

Proof: Suppose the ***RI check*** condition is false but an insertion transaction in the *referencing relation* is committed. The ***RI check*** condition is false, $\triangle s.fk \neq r.key$. It violates fk integrity. □

Example 4.6 There are two RI constraints in the relation *Cu*. Thus the ***RI check*** is fired to check the relevance of the insertion in *Cu*. When a tuple {C3, BC, R&D, C2} is inserted in *Cu*, and then the relevant two RI constraints are fired. One is to acknowledge *Cu*.{BC} = *Ar*.{BC}, and *Cu*.{C2} = *Cu*.{C2}. If the results of both ***RI check***s are true, the insertion transaction will be committed. If any of the two RI checks are not true, the transaction will be aborted. □

Example 4.8 If there is a committed insertion {P6, S4, O3, 10, 350} in *Sp*, then three ***RI check*** tuples are appended (one each) to the *RI differential file* of *Pt*, that of *Su*, and that of *Or*. A tuple {P6, DVD, yellow, 200} is appended to *AIF* called *AIF*(*Pt*), a tuple {S4, A, CA} to *AIF*(*Su*) and a tuple {O3, C1, 30, 300} to *AIF*(*Or*). □

The descriptions of the equations are the propagation rules slightly modified from the rules based on [19, 7, 8].

5. View Maintenance

The view self-maintainability (SM) is one of the important considerations in this paper, defined as maintaining a view in response to changes of database relations using only the view and the differential files to the base relations, without accessing the base relations. The SP view satisfies SM [9] (we will show this algebraically in the next section), but join views might need some auxiliary information from other relations [18]. In order to achieve SM of views, auxiliary relations might be prepared before executing view expressions. In other words, with this, auxiliary views can satisfy SM in view execution time. The auxiliary view [18], auxiliary relation [17], auxiliary data [12], and complements [15] all need access to base relations in making their auxiliary information.

A view is a mapping between a query and data. If a join query is issued, then the join condition is mapped to RI constraints and is substituted with the data. Notice that the data is assumed to synchronize via corresponding DFs of database relations. Duplicated tuples in DFs should be eliminated by some algorithm that all the tuples are removed except the first and the last. The *AIF* is derived from the corresponding base relation, so the tuples of *AIF* may be duplicated with tuples within the *AIF* as well as those of *DF* of the corresponding relation. Thus the duplicated tuples should be eliminated. Notice that if the delta is refreshed immediately, there will be no duplicated tuples (within the *AIF*). In general, the duplicated tuples should be minimized in any view update policy (e.g., immediate update, periodic update, and deferred update, etc).

The schema of $AIF(r_i)$ of a base relation $r_i = \{key, a_{i1}, a_{i2}, \ldots, a_{im}\}$ can be defined as $\{key, a_{i1}, a_{i2}, \ldots, a_{im}, sysdate\}$ for i, $m \in \psi$, where *sysdate* is the timestamp recorded by the ***RI check*** condition if answered true. It is assumed that the timestamp recorded by the ***RI check*** condition answered is the same as the relevant committed transaction to the base relation r_i. So the *AIF* is appended by the order of *sysdate*.

Given an RI (e.g., $s.fk \subseteq r_i.key$) and view refresh time, the algorithm for duplicate elimination on the $AIF(r_i)$ is: at first within the $AIF(r_i)$ and then $DF(r_i)$.

Algorithm 1: Duplicate Elimination of *AIF*
Input:
- $DF(r_i)$ for $i \in \psi$

Output: consistent $DF(r)$
Procedural Steps:
1. $t_1 \leftarrow$ last refresh time; $t_2 \leftarrow$ current refresh time; $s \leftarrow m$
2. (Duplicate elimination within *AIF*) Get next tuple of $AIF(r_i)$ with $t_1 < AIF(r_i).sysdate \leq t_2$.
   If no tuple found, go to Step 6.
3. Delete $AIF(r_i).key$ from the hash index if there exists.
   Go to Step 2.
   Else continue.
4. (Duplicate elimination with *DF*) Get next tuple of $DF(r_i)$ with $t_1 < DF(r_i).sysdate \leq t_2$.
   If no tuple found, go to Step 6.
5. Delete $AIF(r_i).key$ from the hash index if there exists.
   Go to Step 4.
   Else continue.
6. If $s = 1$, then stop.
   Else do: $s \leftarrow s - 1$; Go to Step 2.

In this paper, it is assumed that the *AIF* has index on the candidate key. It is not a hard assumption, for the duplicated tuples in AIF can easily be deleted, even if there is no index. The index may be a hash index or a filter bit vector where it can contain only a *key* value. In terms of storage requirements, the hash index approach requires approximately 20*8/0.7 = 228.57 bits per *key* values, where the size of *key* is assumed 20bytes and 70% storage utilization of the hashing scheme, while the bit vector requires one bit per *key*.

5. Performance Analyses

The following values are assigned to the parameters for the analysis. The block size is generally assumed to be B = 4000 bytes, and the I/O cost CI/O = 25 ms/block. The cardinalities of the base tables are assumed to be examined from 1,000 and 10 Terabyte 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, that is Ccomm = 100Kbps ~ 10Mbps. Tuples are filtered from a no screening case (αs = 1.0) and a highly screened case (αs = 0.001). According to the above parameters and cost functions presented in Section 6, 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*).

Fig. 6.1 and Fig. 6.2 show that the total costs of the three methods are strongly dependent on both the selectivity with the screen factor and the communication speed. Fig. 6.1 represents deleted 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 Fig 6.2, the Base method and the *BaseInc* method are less advantageous than the DF method or the AV method. However, if tuples are less screened (so, the screen factor is up to about 1.0), the DF method is less advantageous than the AV method and, in some cases, even less advantageous than the Base method.

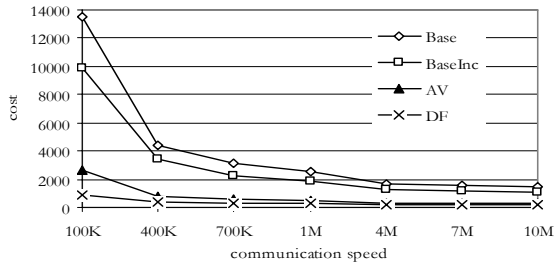

Fig. 6. 1 Cost traverses of *Base, BaseInc, AV*, and *DF* with changing communication speed

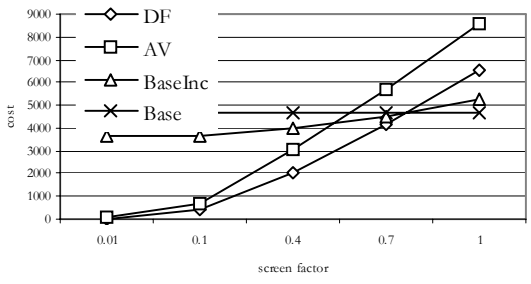

Fig. 6. 2 Cost traverses of Base, BaseInc, AV, and DF method with changing screen factor

Fig. 6.3 represents the IO gains of each method in which the size of the base relation can be assumed to increase with time. The IO gains of the DF method are negative in the initial stage when the size of the base table is increased rapidly, but then becomes positive and remains stable when the size of base table is evenly increased. This means that the number of IO is strongly dependent on the differential file size. So we should figure out the effect of changing the size of the differential file (say, delta) along with the base table remaining stable.

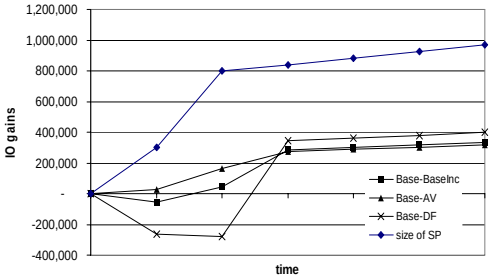

Fig. 6.3 Size of base relation and IO gains of *BaseInc*, *AV*, and *DF* method compared to *Base* method while the base tables are enlarged piecewise-linearly

6. Conclusion and Future Research

In this paper, we have presented an effective method that can derive auxiliary information on which we utilized the potentials of the referential integrity constraint in DBS. Our method, called *DF method*, uses a *DF* (*Differential File*). The *DF* is defined as the changed portion of a base relation, which is extracted from the (*referenced*) base relation by checking the *referential integrity* constraint. In developing algebraic expressions, a series of robust definitions and theorems having to do with several rules for auxiliary information are generated. When updating *SPJ* aggregate views, the relevant base tables are not re-accessed in order to generate auxiliary relations.

The performance analysis has proven to be useful in answering the following questions: (1) Do the *DF* and the *AIF* have a major role in the view maintenance? (2) What will be the mode of cost trajectories along with the data increasing (i.e., diverging or converging)? The performance analysis indicates that the total cost of the *DF method* is closely dependent on the size of the *DF* and the *AIF*, the selectivity (screen) factor, and the transmission rate factor. As these three factors increase with low transmission rate, so the total cost does. When the data size became increasingly large, the cost ratio of the *Base*

and *BaseInc method* diverged too much, but that of the *DF method* increased slightly and that of the *AV method* was intermediate. However, cost benefits are negligible in a worst-case scenario with such factors as a huge *DF* and *AIF* (up to the size of the base tables), and no selection cases. The *DF method* is still superior to the *Base* and *BaseInc* method as well as to the *AV method* in that it can make the *SPJ* aggregate views independent of the current database relations. By using this approach, long transactions from the materialized views intermingled with conventional database transactions can be committed without fear of transaction abort. In the worst-case environment the experimental results represent that managing the *referencing relation* and the view is important.

Our future research will be extended to various issues as follows. A data warehouse setting with various views will be developed in terms of *DF*s. View selection with multiple views and concurrency control issues using these can be worth tackling to relieve various distributed replica settings. In view of this, a view maintenance engine can be implemented. Our method has the potential to deal with the approximation problem of data streaming [1], accuracy of continuous queries and group queries [12], web applications [5], and mobile settings, etc., and therefore it can be easily extended to a myriad of environments.

References

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

[2] J. Blakeley, "Updating Materialized Database Views," In Proc. SIGMOD, pp. 61-71, 1986.

[3] R. Bruckner and A M. Tjoa, "Managing Time Consistency for Active Data Warehouse Environments," In Proc. DaWak, pp. 254-263, 2001.

[4] J. Chen, S. Chen and E.A. Rundensteiner, "A Transactional Model for Data Warehouse Maintenance," In Proc. ER, pp. 247-262, 2002.

[5] S. Ceri, R.J. Cochrane, and J. Widom, "Practical Application of Triggers and Constraints: Successes and Lingering Issues," In Proc. VLDB, pp. 254-262, 2000.

[6] S. Ceri and J. Widom, "Deriving production rules for incremental view maintenance," In Proc. VLDB, pp. 577-589, 1991.

[7] T. Griffin and L. Libkin, "Incremental maintenance of views with duplicates," In Proc. SIGMOD, pp. 328-339, 1995.

[8] T. Griffin, L. Libkin and H. Trickey, "An Improved Algorithm for the Incremental Recomputation of Active Relational Expressions," IEEE TKDE, Vol. 9, No. 3, pp. 508-511, 1997.

[9] Gupta, H., Mumick, I.S., "Selection of views to materialize in a data warehouse," IEEE TKDE 17, pp. 24–43, 2005.

[10] N. Hyun, "Multiple-View Self-Maintenance in Data Warehousing Environments," In Proc. VLDB, pp. 26-35, 1997.

[11] S. Khan and P.L. Mott, "LeedsCQ: A Scalable Continual Queries System," In Proc. DEXA, Vol. 2453, pp. 607-617, 2002.

[12] Y. Kotidis and N. Roussopoulos, "A Case for Dynamic View Management," In Proc. TODS, Vol. 26, No. 4, pp. 388-423, 2001.

[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] J. Lechtenbörger, and G. Vossen, "On the Computation of Relational View Complements," ACM TODS, Vol. 28, No. 2, pp. 175-208, 2003.

[15] V. Markowitz, "Safe Referential Integrity Structures in Relational Databases," In Proc. VLDB, pp.123-132, 1991.

[16] M. Mohania and Y. Kambayashi, "Making Aggregate Views Self-Maintainable", Data and Knowledge Engineering, Vol. 32, No. 1, pp. 87-109, 2000.

[17] D. Quass, A. Gupta, I. Mumick and J. Widom, "Making Views Self-Maintainable for Data Warehousing," In Proc. PDIS, pp. 158-169, 1996.

[18] D. Quass, "Materialized Views in Data Warehouses," Ph.D. Thesis, Computer Science, Stanford Univ., 1997.

[19] X. Quian and G. Wiederhold, "Incremental Recomputation of Active Relational Expressions," IEEE TKDE, Vol. 3, No. 3, pp.337-341, 1991.

[20] D. Theodoratos, "Detecting redundant materialized views in data warehouse evolution," Information Systems, Vol. 26, No. 5, pp. 363-381, 2001.

Wookey Lee received the B.S., M.S., and the Ph.D. in industrial engineering from Seoul National University, Korea. He got finished MSE in the Dept. Computer Science, Carnegie-Mellon University in 2000. He received diploma on TEFL (Teacher for English as a Foreign Language), ISS Canada. He was a visiting professor in the Dept. Computer Science, UBC, Canada from March 2002 to Aug. 2003. He is an Associate Professor in the department of Computer Engineering, Sungkyul University, Korea. He got the best paper award in Korea Management Science Operations Research Society in 2004. He has published many journal and conference papers in distributed database systems, Data Warehouses, and Web IR. He is ACM and IEEE member.

Myung Keun Shin received the B.S. and M.S. in computer science and the Ph.D. in management engineering from Korea Advanced Institute of Science and Technology, Korea. He is a researcher in the 2eit consulting and communication. He has published many journal and conference papers in knowledge management system, database, and Web IR. He is ACM and IEEE member, and Korea Management Science and Operations Research Society.