Implementation and Performance Analysis of Incremental Equations for Nested Relations

Jixue Liu Millist W. Vincent Mukesh Mohania
School of Computer and Information Science
The University of South Australia
Email: {j.liu, vincent, mohania}@cs.unisa.edu.au

Abstract

Incremental view maintenance is widely preferred to view recomputation when an update to a base relation is small in size. The update size becomes an important concept for measuring the cheap performance of the incremental maintenance. In this paper, we investigate what is the limit of the update size for the incremental maintenance. When the size of an update exceeds the limit, the incremental maintenance is no longer cheaper than the view recomputation. The investigation is based on incremental equations for operators in the nested relational model. We implement these equations in the Informix Universal Database Server. We propose a cost model for the performance analysis of the equations. We analyze the factors affecting the incremental maintenance and finally we study the maintenance limit for each operator and for the combination of the operators.

1. Introduction

Materialized views are derived data collections, stored in a database, that are used to improve query processing time or to improve the accessibility of data. They are especially important in data warehousing technology where the contents of the warehouse can typically be considered as a set of materialized views. After creation, a materialized view needs updating whenever the source data from which the view is derived is changed. This process is called *view maintenance*.

There are two main approaches to view maintenance. The first is to recompute the materialized view from scratch after an update to the source data. The other approach is to maintain the materialized view incrementally. In the incremental approach, the updated materialized view is computed from the update, the current materialized view and possibly some source data. For example, let the view V be defined in the flat relational model as $V = r \bowtie s$. For an insertion of a set of tuples δr to r, the incremental technique calculates the change to V as $\delta V = \delta r \bowtie s$ and computes the new view, $V^{new} = (r \cup \delta r) \bowtie s$, by $V^{new} = V^{old} \cup \delta V$ (where V^{old} equals $r \bowtie s$). In this procedure, there exist an equation of $(r \cup \delta r) \bowtie s = r \bowtie s \cup (\delta r \bowtie s)$ supporting the incremental maintenance so that the view content after incremental maintenance is the same as that of view recomputation. An equation of this type is called an *incremental equation* (IE).

Among the two approaches, the incremental approach is generally considered to be less expensive [4] when the size of an update is small in relation to the size of the source data. However, a precise analysis of exactly what 'small' is and determining the limit at which incremental maintenance ceases to be cheaper than full recomputation is a topic which, to the best of our knowledge, is one which has not been investigated before. The main focus of this paper is to investigate what is the limit when incremental computation cease to be cheaper than recomputation. We call this limit the *maintenance limit*.

The framework for investigating the maintenance limit is the IE's developed in [4, 3] for the nested relational model . The reason for using the nested relational model is because it is both an important generalization of the relational model and also an important subset of the object-relational model, a model that is predicted to supersede the relational model as the industry standard [6]. Also, since flat relations are a subset of nested relations, our results also apply to flat relations.

The experimental approach we used to determine the maintenance limit for IE's in the nested relational model is as follows. A nested relational database containing multi-level nested relations is implemented in

Proceedings of the 2000 International Database Engineering and Applications Symposium (IDEAS'00)
0-7695-0789-1/00 $10.00 © 2000 IEEE

the Informix Universal Server. PNF operators for the database are implemented using ESQL/C functions. With the operators, views are created over the relations and IEs are implemented for the views in the server. A cost model is then developed for analyzing the performance of the IEs. The effects of update types and operators on the incremental maintenance are studied and finally, the maintenance limit for each IE is obtained.

2. IEs for Nested Relations

In this section, we introduce notations to be used in this paper. Then we review the operators defined and the IEs developed for nested relations in [4, 3]. Finally, we outline the database in our implementation.

2.1. Notation

Let U denote a universal set of atomic attribute names. Each atomic attribute $A_i \in U$ has a domain denoted by $Dom(A_i)$. Then, a *nested relational schema* R is defined recursively by a set of attribute names $R = A_1 ... A_m R_1^* ... R_n^*$ ($m \geq 1$, $n \geq 0$) where $A_i \in U$ ($i \in [1, ..., m]$) is called an *atomic attribute* and R_j^* ($j \in [1, .., n]$) is another nested relational schema called a *structured attribute* We denote all atomic attributes of R by $\alpha(R)$ and all structured attributes of R by $\beta(R)$.

We define the *level* of an arbitrary structured attribute R_{ar}^* in R to be the number of schemas nesting R_{ar}^*. With the definition, R_j^* of R is on the first level (top level).

Example 2.1 This example introduces a nested relation schema $STUD = \{id, name, subjs^* : \{sjname, year, marks^* : \{ttype, mark\}\}, addrs^* : \{addr\}\}$. The schema is designed for a university database to model the following information of students: (1) *id*, *name*, and addresses $addrs^*$ (a set); (2) their subjects identified by *sjname*, *year*; and (3) marks they obtain for all tests (*ttype*) of a subject.

The set of atomic attributes of $STUD$ is $\alpha(STUD) = \{id, name\}$ while the set of structured attributes of $STUD$ is $\beta(STUD) = \{subjs^*, addrs^*\}$. $marks^*$ is a structured attribute on the second level of R: $\alpha(marks^*) = \{ttype, mark\}$, $\beta(marks^*) = \phi$. □

The domain of schema R is given by $Dom(R) = Dom(A_1) \times ... \times Dom(A_m) \times P(Dom(R_1^*)) \times ... \times P(Dom(R_n^*))$, where $P(Dom(R_j^*))$ is the power set of $Dom(R_j^*)$. A *nested relation* r on R is a set of elements (*tuples*) from $Dom(R)$. Suppose x is a tuple in r. Then the value of x for A_i and the value of x for R_j^* are denoted by $x[A_i]$ and $x[R_j^*]$ respectively. $x[R_j^*]$ is called a *sub-relation* of r. We define $r[A_i]$ to be $r[A_i] = \{x[A_i] \| x \in r\}$.

Definition 2.1 (subschema) A schema $S = \alpha(S) S_1^* ... S_{n'}^*$ is a subschema of schema R, denoted by $S \Subset R$, if there exists a structured attribute R_{ar}^* on some level of R such that: (1) $\alpha(S) = \alpha(R_{ar}^*)$ and (2) for each $S_j^* \in \beta(S)$, $\exists R_{ar_i}^* \in \beta(R_{ar}) \wedge S_j^* \subseteq^p R_{ar_i}^*$ (recursive). If $S \Subset R \wedge \alpha(S) = \alpha(R)$, S is called the *prime subschema* of R. □

We distinguish between a *subset* (denoted by $\subseteq$) of R and a subschema of R. Let $Y \subseteq R$ and $S \Subset R$. If $Y_i^* \in \beta(Y)$, then $Y_i^* \in \beta(R)$. However, if $S_i^* \in \beta(S)$, then S_i^* may not be in $\beta(R)$ because subschema is defined recursively.

Definition 2.2 (Y-disjoint tuple) Let r and δr be two relations defined over a nested relational scheme R and let Y be a subset of R. For $x \in r$, if there exists $y \in \delta r$ such that $x[R - Y] = y[R - Y]$, then x and y are defined to be *Y-overlapping*. If there is no such y existing, x is a *Y-disjoint tuple* from δr. □

In the rest of this paper, we assume that a nested relation is in *partitioned normal form*. That is, $\alpha(R) \neq \phi$ is the key of the relation and the same property applies recursively to all subrelations. We also assume there is no null included in the key value of a tuple in a relation. We abbreviate $\alpha(R)$ to A in the later discussion.

2.2. Operators on Nested relations

We now list operators for PNF nested relations. Readers are referred to [1, 4, 3] to get the precise definitions of the operators.

Let R, S, T be schemas. Let r and δr be relations over r. Let s be a relation over S. Let c be a selection condition. The operators for PNF nested relations are listed in Table 1.

2.3. Incremental Equations

We now recall IEs derived in [4, 3] for the operators listed above.

Expansion

(i) $\eta_R(r \oplus \delta r) = \eta_R(r) \oplus \eta_R(\delta r)$

(ii) $\eta_R(r \ominus \delta r) = \eta_R(r) \ominus \eta_R(\delta r)$

Intersection

(i) $(r \oplus \delta r) \odot s = (r \odot s) \oplus (\delta r \odot s)$

(ii) $(r \ominus \delta r) \odot s = \sigma_{A \notin \delta r[A]}(r \odot s) \oplus ((\sigma_{A \in \delta r[A]}(r)) \ominus \delta r) \odot s$.

Proceedings of the 2000 International Database Engineering and Applications Symposium (IDEAS'00)
0-7695-0789-1/00 $10.00 © 2000 IEEE

Table 1. Operator List

Operator	Condition	Meaning
$\eta_R(r)$	$S \subseteq^p R$	Expansion of s to R
$\sigma_c(r)$		Selection of r
$\pi_S(r)$	$S \subseteq^p R$	Projection of r to S
$r \oplus \delta r$		Union of r_1 and δr
$r \ominus \delta r$		Difference of r_1 and δr
$r \odot \delta r$		Intersection of r_1 and δr
$r \bowtie s$	R, S are joinable	Join of r and s
$\nu_Y(r)$	$Y \subset R$	Nest of r on Y
$\mu_{Y^*}(r)$	$Y^* \in \beta(R)$	Unnest of r on Y^*

Selection

(i) $\sigma_c(r \oplus \delta r) = \sigma_{A \notin \delta r[A]}(\sigma_c(r)) \oplus \sigma_c(\sigma_{A \in \delta r[A]}(r) \oplus \delta r)$

(ii) $\sigma_c(r \ominus \delta r) = \sigma_{A \notin \delta r[A]}(\sigma_c(r)) \oplus \sigma_c(\sigma_{A \in \delta r[A]}(r) \ominus \delta r)$

Projection

(i) $\pi_S(r \oplus \delta r) = \pi_S(r) \oplus \pi_S(\delta r)$

(ii) $\pi_S(r \ominus \delta r) = \pi_S(r) \ominus \pi_S(\delta r) \oplus \pi_S(r \ominus \delta r)$.

Join

(i) $(r \oplus \delta r) \bowtie s = (r \bowtie s) \oplus (\delta r \bowtie s)$

(ii) $(r \ominus \delta r) \bowtie s = \sigma_{A \notin \delta r[A]}(r \bowtie s) \oplus (\sigma_{A \in \delta r[A]}(r) \ominus \delta r) \bowtie s$

Nest

(i) $\nu_Y(r \oplus \delta r) = \nu_Y(r) \oplus \nu_Y(\delta r)$

(ii) $\nu_Y(r \ominus \delta r) = \sigma_{A_A \notin \delta r[A_A]}(\nu_Y(r) \oplus \nu_Y(\sigma_{A_A \in \delta r[A_A]}(r) \ominus \delta r)$

Unnest

(i) $\mu_{Y^*}(r \oplus \delta r) = \mu_{Y^*}(r) \oplus \mu_{Y^*}(\sigma_{A_A \in \delta r[A_A]}(r) \oplus \delta r)$

(ii) $\mu_{Y^*}(r \ominus \delta r) = \sigma_{A_A \notin \delta r[A_A]}(\mu_{Y^*}(r)) \oplus \mu_{Y^*}(\sigma_{A_A \in \delta r[A_A]}(r) \ominus \delta r)$

All the IEs for deletions and some for insertions possess similar properties. The right hand side of each IE is composed of two parts. The first part characterized by $\sigma_{A \notin \delta r[A]}(...)$ is to delete from the old view the tuples derived from tuples in r A-overlapping with δr. In the remaining part, the A-overlapping tuples in r are selected from r by $\sigma_{A \in \delta r[A]}(r)$. These overlapping tuples are then unioned or differenced with δr to produce a temporary relation. Finally, the operator defining the view is applied to the temporary relation and a view update is produced.

2.4. The Database for the Implementation

We now describe the database for the implementation. The schema names and table names of the database is described in Table 2. The schema $STUD$ has been described in Example 2.1. The schema $TEACH$ contains the lecturer names ($lcname$) for each subject in a year. $LECT$ is a schema modeling the information of lecturers. $TEST$ is the schema to describe the test details for a subject in a year. The last schema, $HOBBY$, describes hobbies of a student.

Table 2. Schema of the university database

name	*subj**				*addrs**
	sjname	*year*	*marks**		*addr*
			ttype	*mark*	

schema $STUD$ of relation *stud*

sjname	*year*	*lcnames**
		lcname

schema $TEACH$ of relation *teach*

lcname	*salary*	*speciality*

schema $LECT$ of relation *lc*

sjname	*year*	*ttypes**	
		ttype	*description*

schema $TEST$ of relation *test*

name	*hobbies**
	hobby

schema $HOBBY$ of relation *hobby*

3. Implementation

In this section, we describe the implementation platforms, cost model, and some performance analysis considerations.

3.1. Implementation Platforms and Query Language

The implementation is performed using a Pentium 166 PC computer with two hard disks, 96 MB of memory, and Microsoft Windows NT 4.0 operation system. The database management system used is Informix Database Server 9.14.

The query language we used for the implementation is the OR-SQL proposed in [6]. The part of the SQL relevant to this implementation is the function to query sets, rows, and row-sets. Here, the sets, rows, and row-sets, called *complex objects* are Informix terms corresponding to sets containing atomic values, tuples, and sub-relations in nested relations. The detail of the language can be obtained in [6].

3.2. Implementation Details

This version of the Informix Server is insufficient to directly support for implementing the IEs described in

Section 2. Its has the limitation of supporting multi-level nested row-sets. A detailed discussion of the limitation can be found in [5].

Our way to overcome the limitation is to split the *stud* relation in Table 2 into two tables: *sd* and *sj*. The schema of *sd* is $\{name, subjs^* : \{id\}, addrs^* : \{addr\}\}$. The schema of *sj* is $\{id, sjname, year, marks^* : \{ttype, mark\}\}$. The two tables are linked by the common attribute *id*; the values of *id* in $subjs^*$ of relation *sd* reference *id* values of relation *sj* so that the relationship between subjects and students is kept.

The implementation programs are written in ESQL/C [2] so that derived tables can be used.

3.3. The Cost Model

The cost model for the performance analysis involves the costs of view creation, incremental maintenance, and recomputation. To clarify the description, we use an example of the incremental equation for the join operator with a deletion update to show the relationship between costs and items of the equation.

$$\underbrace{(r \ominus \delta r) \bowtie s}_{c_{rec}} =$$

$$\underbrace{\underbrace{\sigma_{A \notin \delta r[A]}(\overbrace{r \bowtie s}^{c_{cre}})}_{c_{del}} \underbrace{\oplus}_{c_{ins}} \underbrace{(\sigma_{A \in \delta r[A]}(r)}_{c_{ovl}} \underbrace{\ominus}_{c_{cmb}} \delta r) \underbrace{\bowtie}_{c_{inc}} s}_{c_{mtn}}$$

We now detail each cost.

- c_{cre} is the time for creating the materialized view $r \bowtie s$.
- c_{rec} in the left hand side is the time for recomputing the view: $(r \ominus \delta r) \bowtie s$.
- c_{mtn} in the right hand side is the time for incrementally maintaining the view. This time consists of the following components.
  - c_{del} is the time for deleting from the old view the tuples derived from A-overlapping tuples in r.
  - c_{ins} is the time for inserting the tuples of the view update into the view. Since the tuples in the view update are A-disjoint with the view because of the select operation against r, this insertion in fact is the set operation.
  - c_{ovl} is the time for selecting A-overlapping tuples from r, the being-updated relation.
  - c_{cmb} labels the time to conduct PNF union or difference between A-overlapping tuples of r and δr.
  - c_{inc} denotes the time for computing the view update using the operator that defines the view (e.g., $\bowtie$).

  Among these costs, c_{del}, c_{ins}, c_{ovl}, and c_{cmb} are not operator-specific while c_{inc} is specific to the operator for which the IE is derived.

With this cost model, a typical performance analysis diagram is the one shown in Figure 1. The horizontal axis indicates different update sizes while the vertical axis indicates the relative time to view creation. There are three lines in the diagram. One is labeled by 'rec'. It shows the relative view recomputation time: c_{rec}/c_{cre}. It goes downward from top-left corner when updates are deletions. When the update size reaches 50% of the original size, it should come down to half length of the vertical axis.

The second line is labeled by 'inc'. It shows the relative pure recomputation time c_{inc}/c_{cre}. This line is drawn by using updates that do not have A-overlapping tuples with r. Because there is no A-overlapping tuples in the update, no recomputation and no deletion from the old view are needed for the incremental maintenance. Therefore, It is an ideal line for incremental maintenance. When update size reaches 50%, this line meets 'rec' line at half of the vertical axis.

The third line labeled by 'mtn' is the relative time for general incremental maintenance: c_{mtn}/c_{cre}. It goes up from the lower-left corner. The intersection of line 'rec' and line 'mtn' should be above half of the vertical axis and less than 50% of the horizontal axis. The horizontal coordinate of the the intersection point is the maintenance limit.

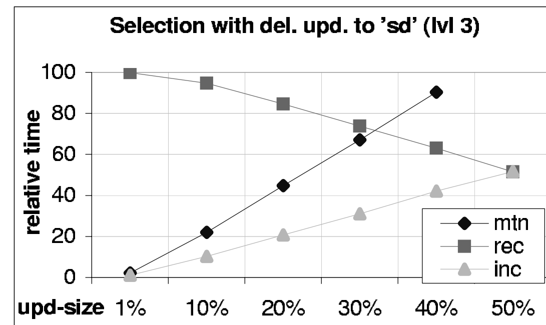

Figure 1. Performance of the selection operator

3.4. Update Type Consideration

In our performance analysis, we only consider the case of deletion updates. This is because the cost of recomputing a view with insertion updates is more expensive than the initial view creation time. It determines that the 'rec' line for insertions goes upward as shown in Figure 2. As a result, the maintenance limit

0-7695-0789-1/00 $10.00 © 2000 IEEE

for insertion updates is larger (better) than that for deletion updates. Consequently, the worst cases which we aim to study will occur when updates are deletions.

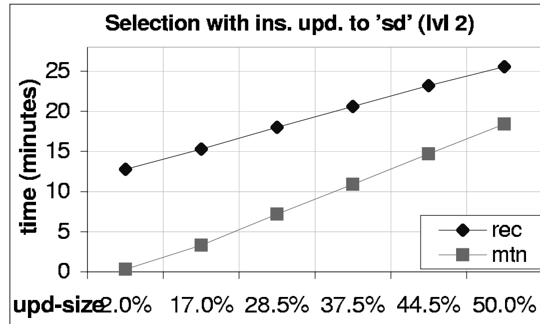

Figure 2. Performance for insertion updates

3.5. Operator Consideration

We choose to study the performances of IEs for the operators of selection, join, nest, unnest individually and for a complex query which consists of more than one operator. This leaves out the analysis on IEs for intersection, expansion, and projection operators. We choose not to study the equations for these three operators because their performances can be obtained by comparison with similar operators. By 'similar' we mean that the definition structures of the operators are the same. For example, when the join of two relations happens on the top level, the definition structure of join is similar to that of the intersection. Therefore, we can estimate the performance of the equations for the intersection operator by that of the equations for the join operator.

4. Performance of Common Operations for All IEs

In this section, we test the performance of common operations in all IEs. These operations corresponding to the costs of c_{del}, c_{ovl}, c_{cmb}, and c_{ins} respectively.

4.1. Test of c_{del}, c_{ovl}, and c_{ins}

The costs of c_{del}, c_{ovl}, and c_{ins} are tested using two relations, r and δr, having the same schema. At the beginning of the test, we select a certain number of tuples from a relation r and put them into δr. Then, c_{ovl} is tested by simulating the query of 'SELECT * FROM r WHERE name IN (SELECT name FROM δr) INTO TEMP temp'. This query selects A-overlapping tuples from r and put them into a temporary relation. After that, c_{del} is evaluated by a deleting query of 'DELETE FROM r WHERE name IN (SELECT name FROM δr)' which deletes A-overlapping tuples from r. In the end, c_{ins} is tested by 'INSERT INTO r (SELECT * FROM δr)'.

Table 3 shows the results of the tests. The data in the table is the average of six repeating experiments. Part (a) of the table gives the costs when r is the three-level nested relation *stud*. Part (b) is about the two-level nested relation *sj*. For both parts (a) and (b), the first column shows the size of δr, while the first row indicates the cost names. The values at the intersection of the first column and the first row are average processing time measured with minutes per thousand tuples(min/k-tuple).

Table 3. c_{del}, c_{ovl}, and c_{ins} (min/k-tuple)

δr size	c_{del}	c_{ins}	c_{ovl}
1000	.050	.090	.080
3000	.130	.107	.080
5000	.136	.116	.088

(a) *stud*

δr size	c_{del}	c_{ins}	c_{ovl}
1000	.067	.042	.050
5000	.053	.033	.047
9000	.056	.040	.052

(b) *sd*

From Table 3 we can make the following observations.

- The data in (a) is larger than the corresponding data in (b). This means that as the number of nesting levels increases, the costs get larger.
- Although $c_{del} > c_{ins}$ and $c_{del} > c_{ovl}$ hold in both tables, their magnitude is on the same grade and the difference is not obvious.
- The three costs in (a) increases slightly as the cardinality of δr gets larger while the costs in (b) are more random. We believe that (a)'s slight increase is related to the memory swap of the server. In the experiments, we see that as the processing size and the number of nested cursors [2] increase, the virtual memory that the server uses expands quickly over the system physical memory and the swap of the virtual memory becomes obvious.

4.2. Test of c_{cmb} for PNF Difference

To conduct this test, we copy some tuples of r into δr. We then delete the tuples in δr from r using PNF difference operation. We test the cost when the difference is conducted on the first level, second level, and the third level. The results of the test is given in Table 4. Each number in the table is the cost of the PNF difference under the according condition.

Table 4. c_{cmb} (min/k-tuple)

size δr	1st level	2nd level	3rd level
1000	.380	.780	5.08
5000	.490	.885	5.10
9000	.492	.888	5.16

From this table we can draw the following conclusions.

- Vertically in the table, the numbers increases slightly as the cardinality of δr increases. As explained before, we believe this increase is caused by swapping virtual memory.
- Horizontally, as the number of nest levels becomes larger, the cost increases significantly. The costs for the first level are the smallest. This is because on the first level, the PNF difference is the same as flat relation difference. The PNF difference is computed on the server side without intermediate data transformation between the server and the client program.

  The cost for the second level is almost the double of that of the first level. This is determined by the process of the PNF difference on the second level. In the process, the top level tuple has to be fetched to the client side. By navigating the top level tuple, the subrelation on the second level is fetched. After two operand subrelations are obtained, the PNF difference of the two subrelations are computed on the client side and is sent to the server side to be stored. Compared with the first level, this procedure is longer and involves more data transformation.

  From the second level to the third level, there is a steep increase of about 5 times. This is because a further level of navigation is needed to access the third level subrelations. The deeper navigation introduces increasing numbers of internal queries from the client to the server, increasing number of nested cursors, and increasing amount of data transformation. Therefore, the cost for the third level PNF difference is much higher than the second level and the first level.

By comparing the data in Tables 4 and 3, we see that the cost of c_{cmb} is much larger than the sum of $c_{del} + c_{ins} + c_{ovl}$, especially when the number of nest levels becomes larger. This indicates that c_{cmb} will be one of the main influences on the total maintenance time c_{mtn}.

5. Maintenance Limit for Each Operator

In this section, we test the cost of c_{inc} and the total maintenance cost of c_{mtn} for different update sizes. Based on these costs, we determine the maintenance limit for each operator.

5.1. Selection Operator

The query for simulating the maintenance limit of the selection operator is to list all tuples of table *stud* if a student has at least one good mark (≥ 90) for at least one subject. The query is

```
SELECT * FROM stud s WHERE EXISTS
 (SELECT * FROM table(s.subjs) j WHERE EXISTS
 (SELECT * FROM table(j.marks) WHERE mark≥90));
```

The selection condition $mark \geq 90$ in the query is set up on third level of the relation *stud* with the selectivity being 10%. The performance analysis diagram has been given Figure 1. From the diagram, we see that the maintenance limit is about 37%.

We also analyzed the maintenance limits for the cases where selection conditions are on the first level and the second level respectively. The maintenance limits for selection condition on all levels are listed in Table 5.

Table 5. Maintenance limits for selection

condition level	1st	2nd	3rd
c_{cmb}/c_{inc} (20%)	1.26	0.36	1.10
limits(%)	18	40.5	32

The data in Table 5 shows that the maintenance limit changes as the number of nested levels increase, but the level number does not determine the maintenance limit. To understand the main reason causing the maintenance limit change, we recall that the cost of c_{cmb} is the main part in the total cost as nest level increases. Therefore, we list a ratio of c_{inc}/c_{cmb} for the update size of 20% in the second row of the table to show that this ratio is opposite proportional to the maintenance limit.

5.2. Join Operator

The query for simulating the maintenance limit of the join operator is

```
SELECT * FROM stud s WHERE EXISTS
 (SELECT * FROM table(s.subjs) j, test t WHERE
  j.sjname=t.sjname AND j.year=t.year AND
  EXISTS (SELECT * FROM table(j.marks) a,
  table(t.ttypes) b WHERE a.ttype=b.ttype) );
```

Proceedings of the 2000 International Database Engineering and Applications Symposium (IDEAS'00)
0-7695-0789-1/00 $10.00 © 2000 IEEE

The two tables in the query join on the second level and the third level. Similar queries are used to simulate join operations on the first level and the second. The maintenance limits for the three case is given in Table 6.

Table 6. Maintenance limits for join

condition level	1st	2nd	2nd & 3rd
c_{cmb}/c_{inc} (20%)	1.48	0.20	0.75
limits(%)	21	44	36

The data in the table is quite similar to that of Table 5. So, we omit the explanation.

The above data is obtained by applying updates to table *stud*, which we call the left operand of the join. We now consider the cases where updates are applied to the right operand of the join.

Our experiments show that when the join is on the third level, difference sizes of updates to the right operand cost do not cause 'mtn' line vary. The line keeps on top of the diagram, which the cost of maintaining views in such case is almost the same as view creation. This is because the navigation of *stud* down to the third level consumes most of the time of the join operation. The experiments also indicates that as the level on which the join happens becomes shallower, the cost of the incremental maintenance changes toward the trend of updates to *stud*.

5.3. Unnest, Nest Operators, and a Complex Query

Table 7 gives the maintenance limits for unnest and nest operators of our experiments.

Table 7. Maintenance limits for unnest, nest, and a complex query

operators	unnest	nest	complex query
c_{cmb}/c_{inc} (20%)	4.83	2.43	0.65
limits(%)	26	29	33

The complex query we analyze include the selection, projection, join, and the unnest operator (used twice). The query is.

```
unnest(marks)(
unnest(subjs)(
 SELECT s.name, s.subjs FROM stud s WHERE EXISTS
  (SELECT * FROM table(s.subjs) j WHERE EXISTS
   (SELECT * FROM table(j.marks) WHERE
   mark≥ 90) )   AND EXISTS
  (SELECT * FROM table(s.subjs) j, test t WHERE
   j.sjname=t.sjname AND j.year=t.year AND EXISTS
   (SELECT * FROM table(j.marks) a,
   table(t.ttypes) b WHERE a.ttype=b.ttype) )
) );
```

where unnest(Y)(r) is a function defined to unnest r on attribute set Y.

The maintenance limit for the query is 33% as indicated in Table 7. The number indicates that the maintenance limits for complex queries may not be worse than the worst limit among all single operators.

6. Conclusion

In this paper, we implemented IEs for the nested relations in the Informix Universal Database Server and conducted performance analysis on them. The performance analysis show that the PNF union and difference operations are the main reasons causing performance decrease of the incremental computation. Generally, the maintenance limits of the incremental equations are among 17–44%. As the number of nested levels increase, the maintenance limit decreases.

References

[1] S. Abiteboul and N. Bidoit. Non first normal form relations: an algebra allowing data restructuring. *Journal of Computer and System Sciences*, 33(3):361–393, 1986.

[2] Informix Corporation. *INFORMIX-ESQL/C Programmer's Manual, Version 9.1.*

[3] J. Liu, M. Vincent, and M. Mohania. Incremental evaluation of nest and unnest operators in nested relations. *Proc. of 1999 CODAS Conf.*, pages 264–275, 1999.

[4] J. Liu, M. Vincent, and M. Mohania. Incremental maintenance of nested relational views. *Proc. of 1999 IDEAS Conf.*, pages 197-205, 1999.

[5] J. Liu, M. Vincent, and M. Mohania. Implementation and performance analysis of incremental equations of nested relations. *Tech Report, ACRC-2000-010* University of South Australia, 2000.

[6] M. Stonebraker and P. Braon. *Object-relational DBMSs tracking the next great wave.* Morgan Kaufmann Publishers, Inc. California, 1999.

0-7695-0789-1/00 $10.00 © 2000 IEEE