A Differential Data Replicator in Distributed Environments

Wookey Lee*, Jooseok Park**, Sukho Kang***

〈Abstract〉

In this paper a data replicator scheme with a distributed join architecture is suggested with its cost functions and the performance results. The contribution of this scheme is not only minimizing the number of base relation locks in distributed database tables but also reducing the remote transmission amount remarkably, which will be able to embellish the distributed databse system practical. The differential files that are derived from the active log of the DBMS are mainly forcing the scheme to reduce the number of base relation locks. The amount of transportation between relevant sites could be curtailed by the tuple reduction procedures. Then we prescribe an algorithm of data replicator with its cost function and show the performance results compared with the semi-join scheme in their distributed environments.

Keywords: logs, DBMS, replication servers, reduction procedures, semi-join, locks, materialized views

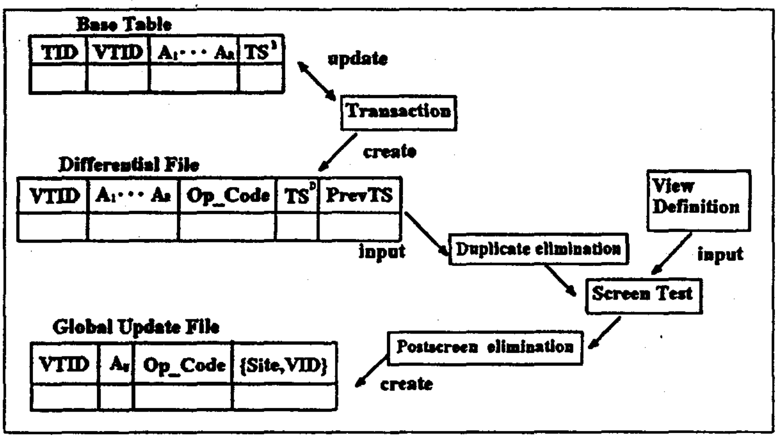
* Dept. of Computer Science Sungkyul Univ. AnYang, Korea
** Dept. of Bus. and Admin. Kyunghee Univ. Seoul, Korea
*** Dept. Industrial Engineering Seoul National University

I. Introduction

A replication server is now one of the important tools to control the complex problems of transaction consistency in distributed database systems. Data replications are basically nessessary and convenient especially in the distributed environments. But it burdens the entire system since mutual consistency of those replicated data must be hard. When communication fails between sites containing copies of the same logical data item, mutual consistency among copies becomes perplexing to ensure, resulting in a partition failure. It fragments the network into isolated sub networks called partitions. There are several replica control algorithms for managing replicated data in the face of network partitioning; these are primary site, voting, grid protocol and so forth [3], [4], [5], [10] and [12]. But these algorithms severely limit data availability during network partitioning and have the same disadvantages of 2PL [7].

Other approaches such as bulk reloads, data flagging, table snapshot, triggering or the rule-based approach have been suggested to solve these problems, but materialized view is superior [15], [17], [22]. Virtual view which is derived from several tables does not physically exist but logically. Materialized view is stored as a separate table; it is useful when users' application may approve noncurrent data with which the replication server manages materialized view on many sites and helps users to access the data they need. Furthermore, it reflects net changes only in the base table, reducing communication costs and relieving the difficulty of concurrency control, which helps the distributed database systems more pragmatic.

Most studies of materialized view have not considered their distributed environments. If ever, they are confined to selection view(S-View) or selection-projection view(SP-View). They do not support the materialized views made by join operation. This study therefore, deals with the structure of a replication server to update join materialized views in its distributed environment.



[Figure 1-1] The materialized view update procedure

We employ the differential update method which utilize logs that record the change of base tables during a certain period (the same refresh time, t_r) as dfferential files. In [figure 1-1] TID means thephysical identifier of a tuple, VTID means the unique identifier of the tuple, Au indicates the attribute related. TS^B and TS^D means the timestamp that the tuple that was updated in the base table and that was appended to the defferential file respectively. And the Op-Code means the operation codes (i.e., insertion, deletion, and modification) and the VID is the relevant view identifier. This method avoids base table locking, making the system's performance efficient. In this study, we update materialized view periodically to save the updating costs [17]. Then a screen test is applied to differential tuples in order to eliminate tuples that are irrelevant to any of the views being updated. Using these methods, we prescribe an architecture of replication server and an algorithm to update materialized view efficiently.

II. An architecture of the Scheme

1. The Concept of Differential Update

The differential update scheme utilizes the log as a differential file that records net changes of the base table just after updating materialized views. This minimizes communication costs and reduces the number of base table access, which makes performance of the system more efficient [17]. When a base tuple is updated by multiple transactions between refresh times, we can get all the views that have the same last refresh time denoted $LR_i = RT_j$ [16, 17, 21]. The primary key of the differential file is VTID that was basically entagged by DBMS. The Op_Code is an attribute that represents tuple changes; 'ins' when newly inserted, 'del' when deleted and 'del_m' and 'ins_m' in series when a modification occurs. When views are to be updated, the following four steps are basically performed. If several changes were happened in a tuple for some time, then we do not need to consider all the records, only the first and the last one which is called the duplicate elimination process. Next, the tuples are checked by the screen test that decides whether the changes are to be transmitted to views or not in terms of view definition. After the tested tuples are sorted by the Op_Code of VTID, condensed as one of those postscreening elimination procedure [17]. Then the results from the post screening elimination, all the values of Op_Codes can be reduced into one of three forms: 'del' , 'ins' , and 'mod'. The Global Update File(GUF) is finally generated to convey net changes to view sites, views are updated at that time.

[Table 2-1] Postscreening Eliminations

Input of the Post-screening	Output of the Post-screening
(ins or insm) + (del or delm)	ignored
(del or delm) + (ins or insm)	mod
ins + insm	ins
del + delm	del

2. General Notations

Gerneral notations:

Ω : the set of site index, for $i \in \Omega = \{1, 2,...,R\}$

B : the page size (bytes)

SF : the Semi join factor

$\otimes$: the join operator

Si, SMi : the sites where Ri is located and the materialized view MVi is located respectively.

$C_{I/O}$, Ccomm : the input and output cost (ms/block) and the transmission rate (bits/sec) respectively

H_{B_Sj} : Height of B^+ tree at Sj site

n_{Ri} : the number of Ri tuples per page (= B/W_{Ri})

Pr_{DF_Rj} : the probability that is needed to join in DF_{Rj}

Distributed reduction notations:

Ui : the number of tuples in the differential file of Ri

Ui^e, Ui^s, Ui^t : the number of tuples after the duplicate elimination procedure, the screen test, postscreening elimination, and transmitted to the view site respectively.

U^{Ri},U^{DJFj} : the number of tuples in Ri and DJFj.

U^{j_r} : the number of tuples in DJFj that are not participated in joining.

αe, αs, αp :the duplicate elimination factor, the screen factor for view predicate, and the postscreening elimination factor

W_{ins}, W_{del}, W_{mod} : the width of GUF tuples with operation code is insertion, deletion, and modification respectively

W_{Ri}, W_{DJFj}, W_{mvi} ,: the width of Ri, DJFj, and materialized view MVi record respectively.

W_B : the width of the B^+ tree.

3. Updating Join Views

One of the urgent problems of the replication server is how to reflect the changes of base table after materializing views. When the tuples of base table are inserted(ins), modified(mod), or deleted(del), we should determine whether these tuples to be newly joined or not.

Considering join operation we also make use of the Global Update File(GUF) [17] that has the following schema: GUF(VTID,Au,Op-Code^V, {Site, VID}), where Op-Code^V indicates types of update to be done for each view in the list {Site, VID}; it will be one of three codes: 'ins', 'del', or 'mod'. The values of $Op\text{-}Code^V$ in GUF are determined based on the Op-Code in the differential file and the screen test. For tuples that pass the screen test, 'ins' and 'del' in DF imply 'ins' and 'del' in GUF: a del_m, ins_m pair in DF implies either 'ins' ,'del', or 'mod' in GUF because a modification of a base tuple can cause its deletion from a view, its 'ins'ertion to a view, or a modification to a view that contain the tuple. Au is Ø when $Op\text{-}Code^V$ is 'del' (since the remote view needs only a VTID for a deletion), A_u for $u \in \Omega$ when $Op\text{-}Code^V$ is 'ins', and the data attributes needed for modification when $Op\text{-}Code^V$ is 'mod'; in case of modification we will assume that $Au = \{ Ai = Value_i \}$ where Ai is the name of the modified attribute and $Value_i$ is its new value.

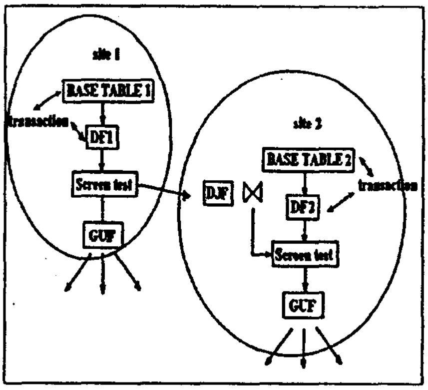
Regarding join operation we consider only two kinds of tuple changes: 'ins' and 'mod'. Because all the other GUF tuples need not to be joined, these tuples are sent directly to view sites where the pertinent materialized views are stored. Then the transmitted tuples are applied to update materialized view. If the Op-Code of GUF tuples is 'ins'; these tuples should be newly joined. In that place it will then be transmitted to the related sites where the tables participating in join operations are located. After being joined with the table, these tuples are appended to the relevant materialized views. When the attribute used in join predicate is changed(in this case Op-Code is 'mod'), GUF tuples that contain these must be manipulated in the same way.

The relevant GUF tuples will be collected at the site where the join operation is made. If the join operation is made by various GUF tuples sent from different sites, it is difficult to manage these tuples as one table, since the sizes of these tuples may be different. Thus we prescribe a new architecture called DJF (Differential Join File). The schema of DJF is as follows: DJF (Site_ID, VTID, Au, TS, Op_Code, Pointer) where Site_ID : a unique identifier of the site where differential file is made, Au: the attributes that are used in join predicate, TS means the timestamp, Op_Code is equivalent to the predicate of differential file's, and the Pointer is that connects the attributes of differential tuples used in materialized views.

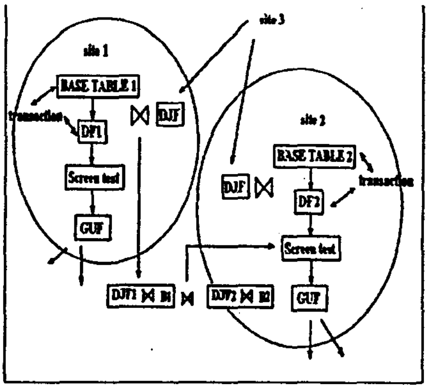
When we make a join operation using DJF, the ways of join operation can be divided into the following two cases: (1) join operation made by the foreign key (2) join operation made by the nonkey attributes. In both cases DJF is created by sending the relevant differential tuples to sites where the pertinent relation is located. In case of (1), DJFj cannot be created in a site other than S_j. When new tuples are inserted in R_i, since the tuples joined with them exist in Sj by the referential integrity rule, DJFj can be created. In case the foreign key of Ri (of course, it is the primary key of Rj) is updated, DJFj can be created also. But when new tuples are inserted in Rj, DJF need not be created; since it will not trigger any new relationship with the tuples of Ri. Even though there is a new insertion in both two tables simultaneously, join can be done by using DJFj.

Next, comparisons should be made between the Join_Attribute (foreign key) of GUFi and the primary key of DF_{Rj}. There are two strategies: if all the tuples sent from site i is matched with that of DF_{Rj}, then there is no need to search all base table of Rj, reducing the processing time needed to join. If there exist at least one tuple of DJF that does not match with DFRj, then all the table of Rj cannot but be searched.

In case of (2) (Join By Non-key), we cannot use referential integrity rule, then DJF should be created at the Si and Sj site. And we can't do join operation using DJF and differential file only, so we should search all the tuples of Ri or Rj.



[figure 2-2] Join by the foreign key



[figure 2-3] Join by non-key

4. Algorithms

We assume that table Ri is in site i and Rj in site j to be joined at site j for $i \neq j$ and these tables are indexed by B+ trees. Here, for convenience' sake, we set A_R be a foreign key of table Ri at site i; it is a primary key of table Rj. If T is a tuple, $T \cdot A$ denotes attribute A of T and the superscripted tuples T^D and T^B mean the differential tuple and the base tuple respectively.

(1) The differential join algorithm

Step1: Get $T_i^D \cdot Au$ where $RT_j < T_i^D \cdot TS \le t_r$

/* Get the tuples that have the same refresh times*/

Step2: Do Duplicate elimination and Screen test and postscreening elimination [17, 21].

Step3: Do algorithm DJF

Algorithm DJF

/* Generating Distributed Differential Join files */

Append file DDi

Do for $\forall Au$ where $u \in \Omega$:

$DD_i \cdot VTID$ $T_i^D \cdot VTID$

$DD_i \cdot Au$ $T_i^D \cdot Au$

$DD_i \cdot TS$ $T_i^D \cdot TS$

$DD_i \cdot Op\_Code$ $T_i^D \cdot Op\_Code$

/* Remote reduction before sending */

If $DD_i \cdot Op\_Code$ =: del

Send DD_j to site S_{GUF}

If $DD_i \cdot Op\_Code$ =: ins and $\exists\ T_i^D \cdot A_R = T_j \cdot A1$

Else stop;

Send DD to site j

If $T_i^{DD} \cdot A_R$=: $T_j^D \cdot A1$

then

join DD tuples to the differential file of Rj

Save it as Temp1

Else

join DD tuples to the base relation Rj

Save it as Temp1

Union Temp1 and Temp2

Save it as Jtemp_i_j

Send Jtemp_i_j to site S_{GUF}

If T_i^{DD}·Op_Code =: mod AND $T_i^{DD} \cdot A_R \neq: T_j^{D} \cdot A_I$

then

send DD_j to site S_{GUF}

Else

Join DDj with differential file of Rj

Save it as Temp3

Send Temp3 to the view site

Endif

End.

III. Examples

Some examples as follows will be applied the algorithm DJF_JOIN; EMP(VTID, ENO, ENAME, JNO) and PROJECT(VTID, JNO, JNAME, BUDGET) where ENO means the Employee number, ENAME is the Employee name, JNO: Project number, JNAME: Project name, BUDGET: Project budget. VTID is a primary key and JNO is a foreign key. Here we assume that relation EMP is located at site1 and relation PROJECT is located at site 2. A materialized view MATL_VIEW1 is assumed to be located at site 3 and is defined as follows:

```
CREATE VIEW MATL_VIEW1 AS
SELECT EMP.ENAME,PROJECT.JNAME, PROJECT.BUDGET
FROM EMP, PROJECT WHERE EMP.JNO = PROJECT.JNO and EMP.AGE > 30
REFRESHCED BY 7 DAYS ;
```

[Table 3-1] EMP, PROJECT tables and materialized view MATL_VIEW1

EMP

VTID1	ENO	ENAME	AGE	JNO
1	E1	LEE	31	J1
2	E2	KIM	25	J2
3	E3	PARK	36	J2
4	E4	YUN	43	J3
5	E5	BAE	29	J5
6	E6	SIN	37	J3
7	E7	HAN	28	J2

PROJECT

VTID2	JNO	JNAME	BUDGET
101	J1	CAD	100
102	J2	CAM	300
103	J3	OR	150
104	J4	LP	240
105	J5	MRP	450

MATL_VIEW1

ENAME	JNAME	BUDGET
LEE	CAD	100
PARK	CAM	300
YUN	OR	150
SIN	CAM	300

EMP, PROJECT, MATL_VIEW1 are shown in [table 1]. Tables 2 to 4 show that the results of duplicate elimination and screen test applied to the differential file of EMP.

[Table 3-2] Differential file of EMP

VTID1	ENO	ENAME	AGE	JNO	Op_Code
8	E5	CHOI	35	J4	'ins'
3	E3	PARK	36	J2	'del_m'
3	E3	PARK	36	J4	'ins_m'
5	E5	BAE	29	J5	'del_m'
5	E5	BAE	29	J5	'ins_m'
5	E5	BAE	29	J4	'del'
1	E1	LEE	31	J1	'del'

[Table 3-3] Differential file of EMP after duplicate elimination

VTID1	ENO	ENAME	AGE	JNO	Op_Code
8	E5	CHOI	35	J4	'ins'
3	E3	PARK	36	J2	'del_m'
3	E3	PARK	36	J4	'ins_m'
5	E5	BAE	29	J5	'del_m'
5	E5	BAE	29	J4	'del'
1	E1	LEE	31	J1	'del'

[Table 3-4] Differential file of EMP after the screen test

VTID1	ENO	ENAME	AGE	JNO	Op_Code
8	E5	CHOI	35	J4	'ins'
3	E3	PARK	36	J2	'del_m'
3	E3	PARK	36	J4	'ins_m'
1	E1	LEE	31	J1	'del'

[Table 3-5] Differential file of EMP after postscreening elimination

VTID1	ENO	ENAME	JNO	Op_Code
8	E5	CHOI	J4	'ins'
3	E3	PARK	J4	'mod'
1	E1	LEE	J1	'del'

After applying the above procedures, GUF tuples created are as follows: (8; E5; CHOI; J4; 'ins'; {S3,MATL_VIEW1}), (3; J4; 'mod'; {S3,MATL_VIEW1}), (1; 'del'; {S3, MATL_VIEW1}). For the tuple of which the Op_Code is 'del', it will be sent to site 3 directly. DJF created is shown in [table 6].

[Table 3-6] DJF created at site 2

SITE ID	VTID1	Au	ENAME
S1	8	J4	CHOI
S1	3	J4	PARK

The procedure which reflects the above changes to MATL_VIEW1 is as follows:

1. Among GUF tuples of EMP, send the GUF tuples of which the Op_Code is 'del' to site3 and delete all the tuples of MATL_VIEW1 that have the same VTID values.

[Table 3-7] updated materialized view by GUF

VTID1	VTID2	ENAME	JNAME	BUDGET
3	102	PARK	CAM	300
4	103	YUN	OR	150
6	103	SIN	CAM	300

2. Send GUF tuples of EMP of which the Op_Code is 'ins' and the tuples of which the foreign key is modified to site 2, then make DJF.

3. Compare Join_Attribute of DJF with primary key of differential file of PROJECT. If two values are the same then make a join operation and apply the screen test to these tuples. Send them to the site where MATL_VIEW1 is located and append them to MATL_VIEW1. If not, search table PROJECT and do the same operation to the whole table.

[Table 3-8] join operation using DJF

Site_id	VTID1	JNO	ENAME
S1	8	J4	CHOI
S1	3	J4	PARK

⊗

VTID2	JNO	JNAME	BUDGET
104	J4	LP	240

[Table 3-9] final MATL_VIEW1

VTID1	VTID2	ENAME	JNAME	BUDGET
4	103	YUN	OR	150
6	103	SIN	CAM	300
8	104	CHOI	LP	240
3	104	PARK	LP	240

IV. Performance Analysis

1. Cost functions

In this section, we are to compare algorithm DJF_JOIN with semijoin algorithm. In comparison, we considered communication costs and I/O costs and assumed that Ri and Rj have a clustered index on join_attribute (Au).

1.1 Applying the Yao's cost function

Yao suggested a cost function that for accessing N records randomly distributed in a file of P records stored in K pages, a formula for the expected optimal number of page accesses is given in [23]:

$$f(N, P, K) = m*[\, 1 - \Pi_{i=1}^{k} (n - n/m - I + 1)/(n - I + 1)\,].$$

This formula assumes that the scheduling of page accesses is optimal, that is, the same page is not accessed more than once.

1.2 Cost of the DJF scheme

In order to establish the cost functions, we first determine the number of tuples that pass through each stage of the procedure.

$Ui = Ui.ins + Ui.del + Ui.delm + Ui.insm$ (where $Ui.delm = Ui.insm$)

$Ui^e = U^ei.ins + U^ei.del + U^ei.delm + U^ei.insm = Ui.ins + Ui.del + \alpha e(Ui.delm + Ui.insm)$

$Ui^s = U^si.ins + U^si.del + U^si.delm + U^si.insm = \alpha sU^ei.ins + \alpha sU^ei.del + \alpha sU^ei.delm + \alpha sU^ei.insm$

$Ui^t = U^ti.ins + U^ti.del + U^ti.mod = \alpha pU^si.ins + \alpha pU^si.del + \alpha p(U^si.insm + U^si.delm)$

The total cost in algorithm DJF_JOIN can be divided by the site Si, Sj and Sm

[Table 4-1] Costs by algorithm RF: (a) Cost in site Si (b) Cost in site Sj (c) Cost in site Sm.

(a) Cost in site Si

title	expressions	cost function
CIO1	reading Ui tuples from DF_{Ri}	$C_{I/O}(U_{i\,ins} + U_{i\,del} + U_{i\,delm} + U_{i\,insm})W_{R/B}$
CIO2	Cost of sorting Ui^s tuples	$C_{I/O}*2*Ui^sW_{R/B}$
CCOM1	transmitting GUF to Sj and SMi	$8(U^t_{i\,ins}W_{ins} + U^t_{i\,del}W_{del} + U^t_{i\,mod}W_{mod})/C_{comm}$
Cost in Si = CIO1 + CIO2 + CCOM1		

(b) Cost in site Sj

title	expressions	cost function
CIO3	accessing the B^+ tree at the view site	$CI/O[(H_{B\ SMi} - 1) + f(\alpha sNr,\ \alpha_s N_r W_{R/B}, U^t)]$
CIO4	Cost of updating the data in the view table	$CI/O2 * f(\alpha sNr,\ \alpha sNrW_{R/B}, U^t)$
Cost in Si = CIO3 + CIO4		

(c) Cost in site Sm

title	expressions	cost function
CIO5	reading Uj tuples in DF_{Ri}	$C_{I/O}(U_{i\ ins} + U_{i\ del} + U_{i\ delm} + U_{i\ insm})W_{R/B}$
CIO6	Cost of sorting U_i^S tuples	$C_{I/O}*2*U_i^S W_{R/B}$
CIO7	Cost of reading JDFj	$C_{I/O} * U^{DJFj} WJDFj/B$
CIO8	sorting DJFj by join attribute	$C_{I/O}*2*U^{JDFj} W_{JDFj/B}$
CIO9	Cost of reading Rj tuples for join operation with U^{j_r}	$Pr_{DF\ Ri}*\{C_{I/O}[(H_{B\ Si} - 1) + f(U^{Rj}, U^{Rj} W_{Ri/B}, U^{j\_r})\}$
CCOM2	Cost of sending joined tuple to SMi + Cost of sending the change of Relation Rj to SMi	$8*U^{JDFj}*Wmvi /Ccomm + 8(Uj^t delWdel + Uj^t mod Wmod)/Ccomm$
Cost in Sj = CIO5 + CIO6 + CIO7 + CIO8 + CIO9 + CCOM2		

2.2 The cost of the semijoin

If algorthm DJF is not used, we can use semi-join to maintain materialized view after join operation. When we use semijoin, the algorithm and cost function is as follows.

(1) semi-join algorithm

Step1: Send the attribute of Ri which is used in join predicate to site Sj where Rj is located. (where size of Ri is greater than that of Rj)

Step2: In Sj, send the tuples of Rj, that are matched with the attributes of Ri sent from Si, to Si.

Step3: In Si, join Ri with the tuple sent from sj and send them to the sites where materialized views are located.

(2)The Cost function

Total cost by the algorithm semi-join can be divided by the site Si, Sj. It's as follows. The total cost is SIO1 + SIO2 + SIO3 + SIO4 + SIO5 + SCOM1 + SCOM2 + SCOM3.

[Table 4-2] Costs by algorithm semi-join: (a) Cost in site Si (b) Cost in site Sj

(a) Cost in site Si

Title	expressions	cost function
SIO1	Cost of reading join attribute index of Ri	$C_{I/O}[(HB\_Sj - 1) + U^{Ri}W_{R/B}]$
SIO2	Cost of reading the tuples of Rj sent from sj	$8*U^{Ri}WR/BCcomm$
SIO3	Cost of reading Ri to join with the tuples of Rj	$C_{I/O}*[(HB\_Sj - 1) + U^{Ri}W_{Ri/B}]$
SCOM2	Cost of sending joined tuple to the view sites	$8*\alpha_s*U^{Rj}*W_{mvi}/C_{comm}$
COST in Si = SIO1 + SIO2 + SCOM1 + SCOM2		

(b) Cost in site Sj

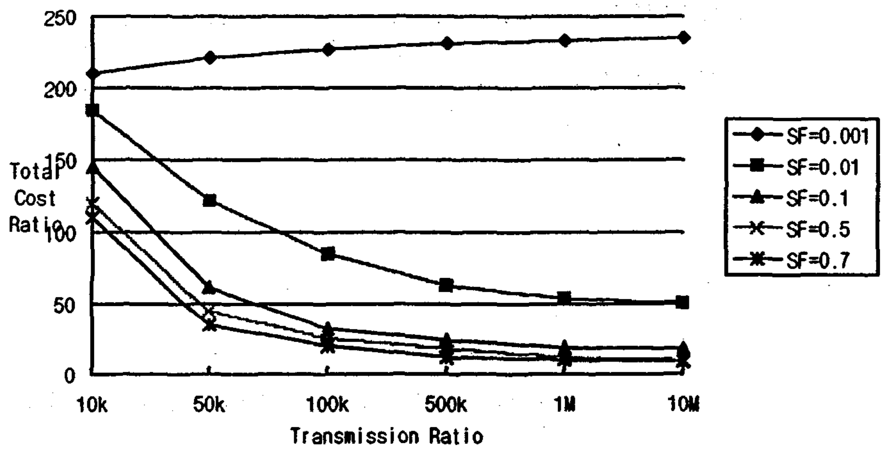
title	expressions	cost function
SIO4	Cost of reading index of Ri from Si	$CI/O*U^{Ri}W_{R/B}$
SIO5	Cost of reading Rj	$CI/O[(H_{B\_Si} - 1) + f(U^{Rj}, U^{Rj}W_{Ri/B}, SF*U^{Rj})]$
SCOM3	Cost of sending the tuples that match the attribute of Ri	$8*SF*U^{Rj}*W_{Ri}/C_{comm}$
COST in Sj = SIO3 + SIO4 + SCOM3		

3. PERFORMANCE Analysis

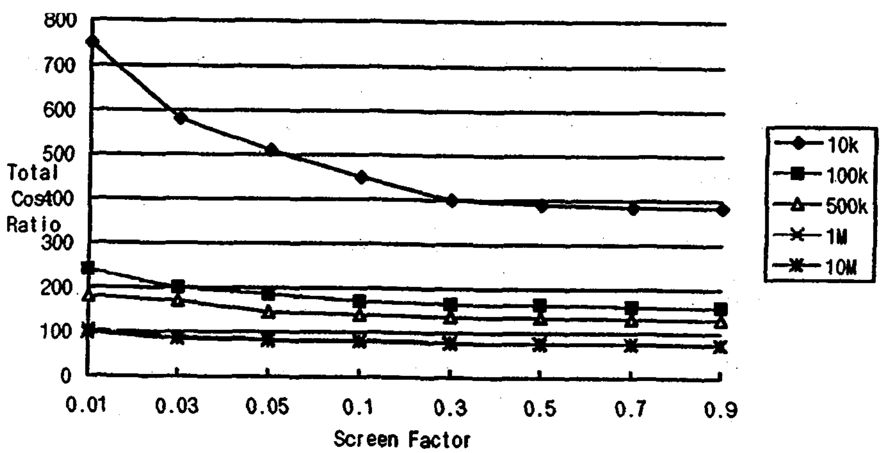
The following values are assigned to the parameters for the analysis. The communication speed varied between 10,000bps and 10,000,000bps. The screening factors(α_s) are varied between 0.01 and 1.0; Here α_s = 1.0 means that there is no screening and α_s = 0.0 is 100% screening that means no tuple will be sent to other sites. We assumed that the size of a page(B) is 4000 bytes, the width of experimental tables(WRi and WRj) will be 200 bytes, the size of B+ tree(WB) is 8 bytes, input and output cost (CI/O) are 25 ms/block, and the sizes of DJF that inserted tuples (Wins) will be 200 bytes, deleted tuples (Wdel) 8 bytes, and the updated tuples (Wmod) 100 bytes respectively.

Assuming the above values and varying numbers of differential tuples, we can calculate the total cost of each algorithm and compared the cost ratios. The results are summarized in figure 5, figure 6, and table 5. In figure 5 and 6, the tuple sizes of each relations are assumed to be 1,000,000 and 100,000 and the communication rate to be 1Mbps.

Varying the tuple numbers of differential files, we calculated the cost ratio (semijoin to algorithm DJF). The ratio is increased as the transmission rate and screen factors are decreased. In Figure 5 we can get a different result with a screening ratio = 0.001, it means that the DJF scheme will not show effective results for the tuples to be transferred is extremely small. Figure 6 shows that the cost ratio is increasing as the number of differential tuples decreases when the transmission rate and screen factors are fixed. Here U^{Ri} means the number of tuples in Ri and Ui Number of tuples in differential tuples of Ri.



[Figure 4-1] Total cost ratio I (Semijoin/DJF_JOIN)



[Figure 4-2] Total cost ratio II (Semijoin/DJF_JOIN)

[Table 4-3] The portion of communication cost in total cost(%)

Algorithm \ S.F		0.1	0.4
Semijoin		24.2	53.6
DJF_JOIN	Ui : 10k Uj : 1k	0.72	0.88
	Ui : 100k Uj : 10k	0.94	1.38
	Ui : 500k Uj : 50k	1.75	3.54

Varying the number of differential tuples and transmission rate, we summarized the total cost ratio between semijoin algorithm and algorithm DJF. When transmission rate and screen factor decreases, algorithm DJF is much better than semijoin. The share of the transmission cost in total cost is shown in table 6. It shows that algorithm DJF takes a much smaller share than that of semijoin, even if we maintain large differential tuples (up to half of a base table). It also indicates that the transmission cost is decreased when when we use algorithm DJF_JOIN. Table 7 shows that the I/O cost ratio also affected by the screen factors and the number of tuples transferred.

[Table 4-4] Total transmission cost ratio (Semijoin/DJF_JOIN)

s.f. \ the number of d.f tuples	Ui : 10k Uj : k	Ui : 100k Uj : 10k
0.1	925.1	89.4
0.4	821.7	81.3

[Table 4-5] Total I/O cost ratio (Semijoin/DJF JOIN)

s.f. \ the number of d.f. tuples	Ui :10k Uj : 1k	Ui : 100k Uj : 10k
0.1	22.1	3.38
0.4	7.38	1.02

V. Summaries and further researches

In this paper a replication server scheme with the architectures is suggested with an efficient performance results. The contribution of this scheme is not only to minimize the number of base relation locks in distributed database tables but also to reduce the remote transmission amount remarkably. The differential files derived from the active log made the scheme reduce the number of base relation locks. The amount of transportation between relevant sites could be curtailed by the tuple reduction procedure such as duplicate elimination, screen test, and postscreening elimination.

The performance tests show that the total cost of this scheme is much smaller than the base table one i.e., the semi-join in a general distributed environment. We want to examine the performance of this scheme comparing with other ones such as ORACLE's Symmetric Replicator, Sybase's Replication Server, IBM's Data Propagator Relational and Nonrelational, Praxis' Omnireplicator, Platinum's Infopump, and CA Ingres' Replicator, etc.

References

[1] Anat Gafni, K. V. Bapa Rao, "A Time based Distributed Optimistic Recovery and Concurrency Control Mechanism," Proceedings of the international Conference on Data Engineering, 1992, pp. 498-505.

[2] Blakeley, J. A., Larson, P. and Tompa, F. W. ,"Efficiently updating materialized views," Proceedings of ACM-SIGMOD Conference Management of Data, Washington, DC, May 1986.

[3] Cheung, S. Y., Ammar, M. H., and Ahamad, M., "The Grid Protocol : A High Performance Scheme for Maintaining Replicated Data," IEEE Transactions on Knowledge and Data Engineering, vol.4, no.6, Dec.1992.

[4] Davison, S. B., Garcia-Molina, H., and Skeen, D., "Consistency in partitioned networks," ACM Compututing Survey, vol.17, No.3, Sep. 1985.

[5] Gilfford, D. K., "Weighted Voting for Replicated data", Proceedings of the 7th Symposium on Operating Systems Principles, 1979.

[6] Goldring, R. "A Discussion of Relational Database Replication Technology," InfoDB, Spring, 1994.

[7] Gorelik, A., Wang, Y. and Deppe, M. "Sybase Replication Server" Proceedings of ACM-SIGMOD Int. Conf. Management of Data, May 1994.

[8] Hanson, E. R., " A Performance analysis of view materialization stratigies, " Proceedings of ACM-SIGMOD Conf. Management of Data, May 1987.

[9] Horowitz, S. and Teitelbaum, T., "Generating editing environment based on relations and attributes," ACM Transactions on Programming Language and Systems, vol. 8, oct. 1986.

[10] Jajodia, S. and D. Mutchler, "A Hybrid Replica Control Algorithm Combining Static and Dynamic Voting," IEEE Transactions on Knowledge and Data Engineering, vol.1,no.4,Dec.1989.

[11] Kahler,B. and Risnes,O.,"Extending logging for database snapshot refresh," Proceedings of Very Large Data Bases, Brighton England, Sept.1987, pp.389-398.

[12] Kumar M., "Performance Analysis of a Hierarchical Quorum Consensus Algorithm for Replicated Objects" Proceedings of Distributed Computing System, 1990.

[13] Lindsay, B. G., Hass, L. ,Mohan C., Pirahesh H., and Wilms H., "A snapshot differential refresh algorithm," Proceedings of ACM-SIGMOD, June 1986, pp.53-60

[14] Ozsu, M. T., and Valduriez, P., Principles of Distributed Database Systems, Prentice-Hall,1991.

[15] Roussopoulos, N. and Kang, H. "Principles and Techniques in the design of ADMS+/-, " IEEE Computer, Dec. 1986.

[16] Segev, A. and Fang, W., "Optimal update policies for distributed materialized views," Department of Computer Science Reseach, Lawrence Berkeley Lab., Technical Report. LBL-26104, 1988.

[17] Segev, A. and Park, J., "Updating Distributed Materialized Views," IEEE Transactions on Knowledge and Data Engineering, Vol.1, No.2, June 1989.

[18] Shmueli, O., and Itai, A., "Maintenance of views," Proceedings of ACM-SIGMOD, Boston, MA, 1984.

[19] The, L., " Distribute Data Without Choking The Net" Datamation, January, 1994.

[20] Tompa,F.W. and Blakeley,J.A., "Maintaining Materialized Views without Accessing Base Data," Information Systems, Vol. 13, 1988.

[21] Wookey Lee, S. Kang, and J. Park, "Refreshing Distributed Multiple Views and Replicas," Journal of the Korean OR/MS Society, Vol. 21, No. 1, April 1996, pp. 31- 50.

[22] Wookey Lee, S. Kang, and J. Park, "Replication Server Scheme in Distributed Database Systems," Proceedings of the APDSI Conference, Hong Kong, Vol. 3, No. 1, June, 1996, pp. 1275- 1281.

[23] Yao, S. B., " Approximating block accesses in database organizations, " Communications of the ACM, Vol. 20, April 1977.