Simulation of join query processing algorithms for a trusted distributed database management system

H Rubinovitz and B Thuraisingham

The paper describes the simulation of join query processing algorithms for a trusted distributed database management system (TDDBMS). In a TDDBMS, users cleared at different security levels access and share a distributed database with data at different sensitivity levels without compromising security. The query processing algorithms should ensure that users only obtain the information classified at or below their security levels. The paper also describes two algorithms for the join operation and discusses the simulation of these algorithms. Some of the results obtained from the simulation with those obtained from a prototype implementation are then compared, and finally, the authors recommend an algorithm for join query processing in a TDDBMS.

In a trusted database management system (TDBMS) users cleared at different sensitivity levels are expected to access and share a database with data at different sensitivity levels without compromising security. A trusted distributed database management system (TDDBMS) consists of multiple TDBMSs which are interconnected with each other via a trusted network (TN)\(^*\). In previous articles, we have described our investigation on the design and development of TDDBMSs. In the first article\(^1\), we defined an architecture for a TDDBMS and subsequently discussed issues on developing a security policy, query processing strategies, and transaction management algorithms. In the second article\(^2\), we focused on the design and development of the query processing component of the TDDBMS. The query processing component was called the distributed query processor (DQP). We described the design and prototype implementation of the DQP. In this, the third article, we discuss the simulation of query processing algorithms and compare the results with those obtained from the prototype described in the second article. In particular, join algorithms for a TDDBMS, the simulation model and the experiments that were carried out, are detailed.

Simulation has been considered to be a significant cost-effective tool for modelling the behaviour of distributed systems\(^3\). Many implementation and simulation studies of distributed query processing algorithms have been carried out previously\(^4\). However, none of the previous studies has considered security issues in distributed query processing algorithms. Before building an operational TDDBMS, it is essential that the algorithms be simulated. This is because, although our DQP prototype described in Rubinovitz\(^5\) has shown to be a valuable tool for the validation of distributed query processing algorithms, due to resource constraints it was impossible for the prototype to depict the real-world operational scenario. For example, the TDDBMS in the real-world may consist of thousands of physical nodes while the number of nodes handled by our prototype was three. Therefore, simulation studies should also be carried out particularly for scenarios which cannot be handled by prototypes. Comparing the results of our simulation with those obtained from the prototype, the simulation model was validated and subsequently extended beyond the physical configuration of the prototype.

The organization of the paper is as follows: the next section describes issues on secure distributed query processing. Secure distributed query processing algorithms that we simulated are described in the third section, and details of the simulation are given in the fourth section. The fifth section concludes the paper. It is assumed that the reader is familiar with concepts in distributed DBMS (DDBMS) and TDBMS. An excellent discussion on DDBMS concepts is given in Cer\(^6\). A useful starting point for concepts in TDBMS is the Air Force Summer Study Report\(^7\).

**ISSUES ON MULTILEVEL DISTRIBUTED QUERY PROCESSING**

The functions of the distributed query processor (DQP) will depend on the way the multilevel data is distributed. In this section, multilevel data distribution issues and an architecture for distributed query processing are described.
Multiplex data distribution

A multiplex relational data model is utilized at the local and global levels of the TDBMS, and then a multiplex relational model is obtained by extending the relational model with support for multiplex security constructs. This section discusses the essential points of the multiplex relational data model being considered and the data distribution issues.

A multiplex relational database consists of a set of multilevel relations. A multiplex relation is defined to be a relation in which each tuple is assigned a security level. In a multilevel world, it is possible for users at different security levels to have different views of the same entity. This feature is supported in some relational TDBMSs by a mechanism called polyninstantiation. With this mechanism, it is possible for two tuples with the same primary key to exist at different security levels. Each tuple represents the view of the corresponding entity at the level specified. A user reads a tuple if his level dominates the level of the tuple. Therefore, it is possible for a user at level L to read different tuples with the same primary key at level L or lower if such tuples exist. A user updates a tuple if his level is the same as that of the tuple.

The result of a JOIN query is illustrated with an example. Consider the multiplex distributed database shown in Figure 1. The database consists of two relations, EMP and DEPT. EMP has attributes SS#, Ename, Salary, and D#. The key attribute of EMP is SS#.

DEPT has attributes Dept#, Dname, and Mgr. The key attribute of DEPT is Dept#. Also, SS#, Salary, D#, and Dept# are numbers, while Mgr, Dname, and Ename are words. In this example, we have assumed that a multiplex relation is decomposed and stored in single-level fragments. EMP has three fragments, EMP1-U, EMP2-U, and EMP3-S, at sites S1, S2, and S3, respectively. EMP1-U and EMP2-U are Unclassified while EMP3-S is Secret. DEPT has two fragments, DEPT1-U and DEPT2-S, at sites S1 and S2, respectively. DEPT1-U is Unclassified while DEPT2-S is Secret. EMP1-U is also replicated as REMP1-U at site S3. We assume that EMP1-U is the primary copy.

The results of the query ‘EMP JOIN DEPT where EMP.D# = DEPT.Dept#’ posed by Unclassified and Secret users are shown in Figure 2. In the case of the restricted join operation, the lower level polyninstantiated tuples in the Unclassified fragments are not involved in the join operation. In the case of the unrestricted join operation the lower level polyninstantiated tuples are not eliminated. The join query is processed by the DQP. The steps involved in processing the join query will be the subject of the third section.

Distributed query processing

The distributed query processor (DQP) is a module that augments a TDBMS at each site as shown in Figure 3. The DQPs at different nodes communicate with each other via the trusted network. The components of the DQP are shown in Figure 4. The user’s query is parsed by the request user interface manager (Request-UIM). The query transformer (QT) transforms the parsed
query at the logical level. That is, the query on a relation is transformed into queries on logical fragments. Discretionary access checks are also performed by the QT. The transformed query is then processed by the query optimizer, which determines the most efficient way to execute the query. The distributed execution monitor (DEM) monitors the execution of the query. The DEM interfaces to the local DBMS as well as to the trusted network. Response is given to the user via the response user interface manager (Response-UIM). A DQP process executing at level L services requests of users at level L. The system must ensure that two DQPs at different nodes communicate only if they both operate at the same level. The DQP does not perform any access mediation. The trust that must be placed on the DQP depends on whether label integrity and/or data integrity are required. Below we discuss various cases.

CASE 1. Untrusted DQP. Let us consider the case where the entire DQP is untrusted. In this case, we assume that there is a DQP process per security level. When a user operating at level L issues a request, the DQP at level L is invoked by the trusted operating system to act on behalf of the user. The DQP at level L obtains the relevant metadata classified at or below the level L in order to generate the execution strategy. The DEM component of the DQP, which carries out the execution strategy, acts as a user process to the TDBMS at that node. The TDBMS will ensure that only data at or below the DEM’s level is retrieved. The trusted network must ensure that the DEM operating at level L at a node N can only communicate with a DEM operating at level L at a node M. Finally, the DEM at the user’s site delivers the response assembled to the user.

As the DQP is untrusted, the labels displayed by the TDBMS are advisory. In some cases, this may not be a severe limitation if the labels displayed by the TDBMS at a node are themselves untrusted*. Another disadvantage with having an untrusted DQP is that one cannot guarantee the integrity of the data itself. For example, a user could request the names of employees working in department D1 and the query parser, transformer, or optimizer could modify the query to retrieve the

*Note that TDBMSs such as Sea View® and Lock Data Views® are designed in such a way that the labels are untrusted. Note also that TDBMS architectures such as the OS-provided MAC Architecture utilized by SeaView and the Replicated Distributed Architecture utilized by SINTRA® do not ensure label integrity. In order to ensure label integrity, the front-end must also be trusted. As a result, most TDBMSs do not provide label integrity.
employees working in department D2. An advantage of having an untrusted DQP is that it has no security critical components. This means that there are no security assurance requirements.

**CASE 2.** Trusting the Entire DQP. At the other extreme, the entire DQP could be trusted. Then not only can it be guaranteed that the labels are trusted, but also the integrity of the results can be ensured. However, this would mean trusting all of the other modules of the DQP. As a result the amount of trusted code will be quite large and with present verification technology it may be difficult to obtain high assurance.

**CASE 3.** Trusting the DEM and the response user interface manager. If trusting the integrity of the labels is a requirement for an application, then the DQP must preserve the integrity of the labels (assuming that the labels provided by the local TDBMSs are trusted). There are various ways to ensure this. In one of the approaches, all of the modules of the DQP except the DEM and the Response-UIM are untrusted. In this case, the DEMs can be guaranteed to preserve the integrity of the labels and the Response-UIM will guarantee that the labels accurately reflect the classification of data. The trusted network must also ensure that the labels are not tampered with. Although this approach will reduce the amount of trusted code compared to that of Case 2, there is still a large part of the code that needs to be trusted. Furthermore, since the Request-UIM, the QT, and the QO are untrusted, the integrity of the data returned cannot be guaranteed.

For the simulation we have assumed the first case where the entire DQP is untrusted. Therefore, the integrity of the labels or the data cannot be guaranteed.

### JOIN ALGORITHMS FOR A TDBDBMS

#### Overview

For the join operation two algorithms have been simulated. They are the non-distributed join algorithm and the distributed join algorithm. Both algorithms use the semi-join algorithm as the query processing tactic. In the non-distributed join algorithm, the fragments of a relation are merged, the semi-join operation is performed in order to reduce the size of the merged fragments, and finally the join operation is performed. In the distributed join algorithm, the semi-join operation is performed between the fragments and finally the join operation is performed between the reduced fragments. Both algorithms handle polyninstantiation, horizontal fragmentation, and replication.

To compute the cost and benefit of executing the various semi-join operations, the method proposed in Ceri is used, and the essential points stated here. When there are multiple semi-join operations that have to be executed in order to process a join query, the query optimizer has to determine which semi-join operation to execute. It usually selects the semi-join which is most profitable. In order to compute the profitability of a semi-join, its cost and benefit are computed. The cost of a semi-join $R S_{ij}$ is $C_0 + C_1 \times \text{size}(A \text{ in } S) \times \text{val}(A,S)$ where $C_0$ and $C_1$ are constants, $\text{size}(A \text{ in } S)$ is the number of bytes of all the attributes of $A$ in $S$, $\text{val}(A,S)$ is the number of distinct
A values appearing in $S^\dagger$. The benefit of a semi-join is calculated in terms of avoided future transmission costs. The benefit of $R S_J A$ is $(1 - \pi) \cdot \text{size}(R) \cdot \text{card}(R) \cdot C_1$ where $\pi$ is the selectivity factor. Note that the selectivity factor of a semi-join $R S_J A$ is estimated to be the number of different $A$ values in $S$ divided by the number of values in the domain of $A$. Profit is set to be $(\text{Benefit} - \text{Cost})$, therefore, the most profitable semi-join among a list of semi-joins is the one with the highest profit.

In the next section the non-distributed join algorithm is described, and in the following section the distributed join algorithm is described, accompanied by examples to illustrate the details of the algorithm. A version of these algorithms is given in Rubinfonitz\textsuperscript{2}.

Non-distributed join algorithm

In this algorithm, it is assumed that the relations are fragmented and also replicated. Further, the tuples are also polyaninstantiated across security levels, and polyaninstantiation can occur within as well as across sites. Suppose a user poses a query at a security level $L$. The user can issue either a restricted or an unrestricted request. Let the query be:

$$R_1 \text{JOIN}_{R_1.A_1 = R_2.A_1} R_2 \text{JOIN}_{R_2.A_2 = R_3.A_2} R_3 \text{JOIN} \ldots \text{JOIN}_{R_{n-1}.A_{n-1} = R_n.A_n} R_n$$

In the case of a non-distributed join, the first step is to generate a strategy for performing the union of all the fragments of a relation $R_i$ ($1 \leq i \leq n$). The next step is to determine a strategy for performing the join operation. The third phase is to carry out the execution. Thus the algorithm has three phases.

Description of the algorithm:

Output: Result of the join).

Phase 1:

Output: Profiles on the relations $R_i$ ($1 \leq i \leq n$) and the site which stores the union of the fragments of $R_i$.

For each relation $R_i$ ($1 \leq i \leq n$) do the following:

- Find the sites which store at least one fragment of $R_i$ whose tuples are classified at or below $L$. Let $S_1, S_2, \ldots, S_t$ be the sites.
- At each site $S_j$ ($1 \leq j \leq t_i$), if the query is restricted, eliminate from consideration the lower level polyaninstantiated tuples in the local multilevel database.
- Select among the sites $S_1, S_2, \ldots, S_t$, the site $S_k$ which has the greatest number of tuples associated with the relation $R_i$ at or below $L$.

Sk is the site which is responsible for the union operation.

- From each site $S_p$ ($p \neq k$) belonging to the set of sites $S_1, S_2, \ldots, S_t$, send the tuples associated with the relation $R_i$ at $S_p$ (at or below $L$) to the site $S_k$. Also, for each tuple, attach its security level if the query request is restricted.
- At site $S_k$, perform the logical union of all the fragments of $R_i$ (at or below $L$). If the query is restricted, then eliminate the lower level polyaninstantiated tuples from the various fragments.
- Output profile on relation $R_i$ at site $S_k$.

End (for every relation $R_i$).

End (of Phase 1)

Phase 2:

Input: Query: query specification, profiles on relations $R_1, R_2, \ldots, R_n$ and their sites. 
Output: Query execution strategy.

Construct the join graph for the query. For each join operation between $R_i$ and $R_j$, include $R_i \text{SJ} R_j$ and $R_j \text{SJ} R_i$ in a list. Let the list of semi-joins constructed be $LSJ$.

Set counter $= 1$.

Repeat

- For each unmarked semi-join in the list $LSJ$, compute its profit.
- Select the semi-join which is most profitable and execute it logically.
- Replace the relation that was reduced by the semi-join with the reduced version of the relation (at the logical level).
- Mark the semi-join that was just executed from the list $LSJ$ with the value given by the counter.

Increment counter by 1. Until all semi-joins have been marked.

Determine the site at which the query should be executed. (Query execution amounts to performing a join between all the reduced relations. The number of tuples that have to be transmitted in order to do this determines the site at which the query is executed.)

Eliminate any semi-joins in the list which reduce the relations at the query execution site (this is post-optimization).

End (of Phase 2)

Phase 3:

Input: Query: query specification, profiles on relations $R_1, R_2, \ldots, R_n$ and their sites, query execution strategy. 
Output: Result of join.

First, physically perform the union of all the relations. Next, physically execute the semi-joins in the list $LSJ$. The order is determined by the numerical value (or mark) associated with the semi-join. That is, if semi-join $P$ has mark $n_1$ and semi-join $Q$ has mark $n_2$ and if $n_1 \leq n_2$, then $P$ is executed before $Q$.

\textsuperscript{\dagger}Note that by $R \text{SJ} A$, is meant that the relation $R$ is reduced. Some references have reversed the order.
Send the reduced relations to the query execution site and perform the final join operations between the reduced relations.
Transfer the result to the site at which the query was posed.

End (of Phase 3)

**Example**

We illustrate the algorithm with the employee-department example discussed previously. Suppose the number of tuples in EMP1-U, EMP2-U, EMP3-S, DEPT1-U, and DEPT2-S are 30, 40, 50, 30, and 60, respectively. In order to simplify the discussion in this example, it is assumed that 30% of all tuples in a fragment are polyninstantiated at the higher level. Let a Secret user at site S1 pose a restricted query to perform the join between EMP and DEPT on the attribute D# of EMP and Dept.# of DEPT.

**Execution of Phase 1**

For relation EMP, first eliminate lower level polyninstantiated tuples if possible. As EMP3-S and REMPI-U are both at site S3, some of the tuples in REMPI at site S3 could be eliminated. The resulting number of tuples in both EMP3-S and REMPI-U is 71 (assuming 30% polyninstantiation). The site to assemble EMP is S3 as it has the greatest number of tuples (i.e., EMP3-S and REMPI-U are both at site S3). EMP2-U is moved to site S3. The lower level polyninstantiated tuples are removed. The resulting relation EMP is at site S3. The number of tuples in EMP is 99.

For relation DEPT, since no two fragments are at one site, no local processing is performed. The site to assemble DEPT is S2. The tuples in DEPT1-U are transmitted to site S2. The lower level polyninstantiated tuples are removed during the union operation. The resulting relation DEPT is at site S2. The number of tuples in DEPT is 81. We also assume that the total number of Dept.# values in the domain is 81.

**Execution of Phase 2**

The semi-joins EMP SJ DEPT and DEPT SJ EMP are listed in LSJ. The profit of each semi-join is computed as follows. It is assumed that C0 = C1 = 10, cost to transfer a number is 8 and the cost to transfer a word is 16. Number of distinct D# values in EMP is 40% of the total which is equal to 40 (approx). The number of distinct Dept.# values in DEPT is 100% of the total (because Dept.# is the primary key) which is equal to 81.

Cost of EMP SJ DEPT = 10 + 10 * 81 * 8 = 6490
Benefit of EMP SJ DEPT = (1 - 81/81) * 40 * 99 * 10 = 0
Cost of DEPT SJ EMP = 10 + 10 * 40 * 8 = 3210
Benefit of DEPT SJ EMP = (1 - 40/81) * 40 * 81 * 10 = 16 200 (approx)

Therefore, DEPT SJ EMP is selected for execution. This will reduce the number of tuples in DEPT to 40. Also, DEPT SJ EMP is marked 1.

Next, the semi-join EMP SJ DEPT is selected for execution, and it is marked 2. Note that the algorithm recomputes the profit of this semi-join, although this is not necessary as it is the only semi-join left. Also, it can be seen that this semi-join will not reduce the relation EMP. This is because (i) we assume referential integrity (i.e., all the D# values in EMP are also in DEPT), and (ii) the relation DEPT has not been reduced because it was reduced earlier by the D# values in EMP. Therefore, the semi-join EMP SJ DEPT is useless.

Next, the algorithm will determine the site of execution. Now, EMP has 99 tuples and DEPT has 40 tuples each. The cost to transmit EMP to site S2 is:

10 + 10 * 99 * 40 = 39 610.

The cost to transmit DEPT to site S3 is 10 + 10 * 40 * 40 = 16 010. Therefore, it is more beneficial to perform the join at S3. The result of the join is transferred from S3 to S1.

**Execution of Phase 3**

It is during this phase that the query execution strategy (both for the union and join) is carried out. The result of this phase is the response to the query.

**Distributed join algorithm**

In this algorithm, it is assumed that the relations are fragmented and also replicated. Further, the tuples are also polyninstantiated across security levels, and polyninstantiation can occur within as well as across sites.

Suppose a user poses a query at a security level L. The user can issue either a restricted or an unrestricted request. Let the query be:

R1 JOIN R2 JOIN R3 JOIN Rn

In the case of a distributed join, the fragments are not merged to form a complete relation. That is, semi-join operations are performed between the fragments in order to reduce them. Finally, the join operation is performed between the reduced fragments. The algorithm is a two-phase one. During the first phase, the query execution strategy is generated. During the second phase, the strategy is physically executed.

**Description of the algorithm:**

**DQP2 (Input: Query: query specification, L: security level, restricted: Boolean.**

**Output: Result of the join.**

**Phase 1:**

**Input:** Query: query specification, L: security level, restricted: Boolean.

**Output:** Query execution strategy.

Construct the join graph for the query. For each join operation between Ri and Rj, include Ri SJ Rj and Rj SJ Ri in a list. Let the list of semi-joins constructed be LSJ.
Set counter = 1.
Repeat
For each unmarked semi-join in the list LSJ, compute its profit.
For each semi-join Ri SJ RJ, its profit is determined by executing the following operations logically and determining its benefit and cost.
For each fragment Fjk of RJ, project it on the semi-join attributes, its primary key. Attach the security level of the primary key value beside the primary key. Send the projected values to each site which has a fragment Fik of Ri.
At a site which has a fragment of Ri, do the following:
Collect all values of fragments of RJ that were received. Eliminate the lower level polynstantiated values if the join is restricted. This can be done as the security level is attached to the primary key. Perform the union of the resulting values to form Ri*J. Reduce the fragment of Ri by Ri*J. Replace the fragment that was reduced by the semi-join with the reduced version of the fragment (at the logical level).
Mark the semi-join that was just executed from the list LSJ with the value given by the counter. Increment counter by 1.
Until all semi-joins have been marked.
Determine the site at which the query should be executed.
(Query execution amounts to performing a join between all the reduced fragments. The number of tuples that have to be transmitted in order to do this determine the site at which the query is executed.)
Eliminate any reductions from the strategy which reduce the fragments at the query execution site (this is post-optimization).
End (of Phase 1).

Phase 2:
Input: Query: query execution strategy.
Output: Result of the join.
Physically execute each step specified in the query execution strategy.
Return result to the site where the query was posed.
End (of Phase 2)

Example
We illustrate the algorithm with the same example discussed for the non-distributed join operation. Let a Secret user pose a restricted query at site S1 to perform the join between EMP and DEPT on the attribute D# of EMP and Dept# of DEPT.

Execution of Phase 1
The semi-joins generated are DEPT SJ EMP and EMP SJ DEPT.
It is assumed that C0 = C1 = 10, the cost to transmit a number is 8 and the cost to transmit a word is 16.

The cost of DEPT SJ EMP is computed as follows:
The SS #, D # attributes of EMP1-U have to be sent to site S2; SS #, D # attributes of EMP2-U have to be sent to site S1; SS #, D # attributes of EMP3-S have to be sent to site S1 and S2. Note that the security label of a tuple has to be transmitted also. We assume that the label is an integer. The cost for these transmissions is:

\[(10 + 10 \times 30 \times 3 \times 8) + (10 + 10 \times 40 \times 3 \times 8) + (10 + 2 \times 50 \times 3 \times 8) \times 2 = 40,840\]

The benefit of DEPT SJ EMP is computed as follows:
Since the tuples in the three EMP fragments will be merged to eliminate the polynstantiated lower level tuples, the number of remaining tuples in EMP is 99. We need to compute the benefit of the semi-joins DEPT1-U SJ EMP and DEPT2-S SJ EMP.

Benefit of the semi-join DEPT1-U SJ EMP = \((1 - 12/81) \times 40 \times 30 \times 10 = 10,500\) (approximately) (note that 40% of the D# values in DEPT are in EMP).
Benefit of semi-join DEPT2-S SJ EMP = \((1 - 24/81) \times 40 \times 60 \times 10 = 16,900\) (approximately)
Total benefit = 27,400 (approximately)
Profit of executing DEPT SJ EMP = Benefit - Cost = -13,440 (approximately)

The cost of EMP SJ DEPT is computed as follows:
The D# attribute of DEPT1-U has to be sent to site S2 and site S3, the D# attribute of DEPT2-S has to be sent to sites S1 and S3. Note that the security label has to be transmitted also.
The cost for these transmissions is:

\[(2 \times (10 + 10 \times 30 \times 8)) + (2 \times (10 + 10 \times 60 \times 2 \times 8)) = 28,840\]

The benefit of EMP SJ DEPT is computed as follows:
First the tuples in the two DEPT fragments have to be merged in order to eliminate the lower level polynstantiated tuples. The remaining number of tuples in DEPT is 81. Benefit of the three semi-join operations EMP1-U SJ DEPT, EMP2-U SJ DEPT, and EMP3-S SJ DEPT is computed as follows:
Benefit of semi-join EMP1-U SJ DEPT = \((1 - 1) \times 40 \times 30 \times 10\)
Benefit of semi-join EMP2-U SJ DEPT = \((1 - 1) \times 40 \times 40 \times 10\)
Benefit of semi-join EMP3-S SJ DEPT = \((1 - 1) \times 50 \times 10\)
Total benefit = 0
Therefore, semi-join DEPT SJ EMP is selected for execution first. Then EMP SJ DEPT is selected. Note that EMP SJ DEPT is useless and need not be computed.

\(^{\dagger}\)Also, note that since EMP1-U is replicated at site 3, DEPT2-S does not have to be transmitted to site 1. That is, our algorithms can be optimized further.
Next, the execution site is determined as follows:
The fragments EMP1-U, EMP2-U, and EMP3-S remain
the same. The fragment DEPT1-U is reduced to 13
tuples and the fragment DEPT2-S is reduced to 27
tuples. This is because 30% of the tuples in EMP1-U
and EMP2-U are polymonotivated. Furthermore, only 40%
of the D≠ values in DEPT are in EMP. Site S3 has 50
EMP3-S tuples and 30 REMP1-U tuples. Site S2 has 27
DEPT2-S tuples and 40 EMP2-U tuples. Site S1 has 30
EMP1-U tuples and 13 DEPT1-U tuples. It can be seen
that the query execution site will be determined to be S3.

Next, if any of the semi-joins between the fragments
have to be performed at site S3, then these semi-joins
can be eliminated from the list. This is the post-optimization
process.

Phase 2 of the execution
During this phase, the execution is physically carried
out. The result is then transferred from site S3 to the site at
which the query was posed.

SIMULATION OF SECURE DISTRIBUTED QUERY PROCESSING
ALGORITHMS

In this section the details of the simulation of the join
algorithms as previously described are discussed, and in
the following subsections the simulation language and the
simulation model are described, followed by discussion
of the experiments carried out.

The simulation language

We used the language HGPPS in the simulation.
HGPPS is a simulation language which extends GPSS by
supporting many new primitives that are desirable for
simulation. Although HGPPS is adequate for the develop-
ment of distributed database models, it still requires
substantial mental manipulations on the part of the
programmer in order to develop a complex model. Hence,
HGPPS has emerged from the GPSS system. HGPPS is
written in C thus providing machine independence.
It has been compiled on Sun Workstations, Vaxs, PCs,
and Amigas. The code produced by HGPPS is faster
than GPSS, as most of the variables and labels have been
defined at a compile time, rather than at run time.

HGPPS supports all the features of GPSS, such as
statements, creating and destroying entities, standard
numerical attributes, random number distributions, and
functions. In addition, several other primitives are
supported by HGPPS. These additions lower the com-
plexity of the simulation task. The additional list of
features supported by HGPPS include the following:

GPSS compatibility. HGPPS is upward compatible
with GPSS allowing users of GPSS to keep the same

†For a discussion on GPSS refer to Gordon*. A detailed discussion
of HGPPS is given in Rubinovitz*.

code that they have already developed for previous
simulations.

C-Interface. HGPPS allows the user to code portions
of the simulation model in C in addition to offering its
set of explicit modelling constructs. This allows more
complex algorithms to be coded in C while allowing
access to HGPPS variables within the C function.

Subscripted variables. While GPSS does not allow all
variable types to be subscripted, HGPPS allows several
more variable types to be subscripted, thereby permitting
a more flexible model to be designed.

Definition of transaction. In GPSS all transactions
must be homogeneous. HGPPS has a more elegant
method of defining transactions. Parameters of a trans-
action are defined in a style similar to that used to define
structures in C.

Subroutines. HGPPS has improved the method used in
accessing subroutines. In GPSS, the user is responsible
for maintaining the stack pointer which is used to point
to the next location on the stack to save the return
address. HGPPS creates a stack, stack pointer,
parameter stack pointer, for each transaction. As HGPPS
maintains the stack for the user, the statements used to
access subroutines are easier to use.

Additional constructs. In addition to many other
extra features, HGPPS supports the essential software
engineering constructs IF-THEN, IF-THEN-ELSE, and
WHILE.

Simulation model

The two secure distributed query processing algorithms
for the join operation have been simulated. This section
describes the basic methodology used for the simulation.
Some general remarks will be given about the actual
implemented prototype system which was used as a
source of timing information for the simulation model.
The experiments that were simulated are then outlined.
Some of the simulated models were validated against
the results from the actual prototype. These experi-
ments were designed to determine the effects of various
parameters on the overall system performance.

The logical architecture considered in the simulation
was shown in Figure 3. It is assumed that the TDBMS
consists of several nodes interconnected by a trusted
network. It is also assumed that all the nodes are designed
identically. Each node is capable of handling multilevel
data, that is, each node has a TDBMS. The TDBMS at
each node is augmented by a DQP which is responsible
for query processing at the global level. The system must
ensure that two DQPs at different nodes communicate
with each other only if they both operate at the same
level.

Prior to the simulation of the algorithms, a prototype
DQP was implemented. This way it was possible to com-
pare some of the simulation results with those obtained
from the prototype. The prototype was demonstrated
using two physical machines. One machine was the
Microvax running Ulitrix (both products of Digital Equip-
ment Corporation). The local TDBMS in this machine
was the Secure SQL DataServer (a product of Sybase Inc). The second machine was a SUN-3 (product of Sun Microsystems) running Unix (trademark of AT&T Bell Laboratories). The local TBMS in this second machine was the non-multilevel version of Sybase DataServer (product of Sybase Inc.). We implemented a front-end to this DBMS so that the security features of the Secure SQL DataServer could be emulated‡.

The implemented DQP is composed of two main components consisting of about 9000 lines of ‘C’ code. The first component, the Query Analyzer (QA), handles the actual decomposition of the query. It consists of the Request-UIM, Response-UIM, QT, and QO as illustrated in Figure 4. The second component is the DEM and is responsible for the actual physical execution of the query.

The QA decomposes the query and generates a list of script commands to be actually executed by each participating site. The following are examples of three commands:

(1) SELECT INTO EMP-DEPT * FROM EMP, DEPT WHERE EMP.D# = DEPT.D#
(2) MOVE 2 EMP-DEPT
(3) SAVE EMP-DEPT

The first command performs the SQL join operation between the EMP and DEPT relations joining on the attributes EMP.D# and DEPT.D#. The second command moves the relation EMP-DEPT to site number 2. The third command saves the in-memory relation EMP-DEPT to permanent storage.

Because a substantial amount of code had been previously written to determine the necessary actions to execute the two join algorithms by the QA, it was decided to invoke the same code for the simulation model also. This condition requires the simulation language to provide access to external ‘C’ routines. The developed simulation model consisted of both a number of ‘C’ routines and code written in HGPSS. The following presents the high level pseudo code for the simulation.

Ask user for his query
Invoke Query Analyzer to generate command script
While there are commands to be executed
   Based on current command, simulate appropriate action
Endwhile
Print performance information

A number of actual experiments were conducted to determine performance levels of the implemented prototype. For each experiment a log was maintained which contained overall CPU time in seconds as well as the amount of CPU time to execute each command. From the collected logs of timing information it is possible to develop an accurate simulation model based on the actual prototype. For example, in order to simulate the MOVE command, the number of tuples contained in the EMP relation and the data transfer time were used to determine the amount of processing time. A function was written to return the processing time based on the actual numbers captured from the prototype.

Discussion of the experiments
This section provides a description of the five sets of experiments that were conducted. The first, second, and third sets of experiments were executed by the implemented prototype as well as simulated. Having validated the simulation model for sets of experiments one, two, and three, the model was extended in sets of experiments four, and five. The configuration of the distributed database for the first and second sets of experiments was based on the Employee-Department example described in the second and third sections. The number of tuples and the number of sites were varied depending on the set of experiments carried out. Due to resource constraints, the fourth and fifth sets of experiments were only simulated. The five sets of experiments are described next‡.

First set of experiments: variable number of tuples for non-distributed join algorithm
Several experiments were carried out for the non-distributed join algorithm with the number of tuples per node as the variable. Execution times were obtained for number of tuples per node equalling 200, 400, 600, 800 and 1000. For each experiment we obtained two sets of results for the algorithm. The first set of results was for the case of the restricted join operation and the second set of results was for the case of the unrestricted join operation. The simulation model used measurements from the various actions of the implemented prototype. An example of an action would be the amount of CPU time required to polyinstantiate a relation. The simulation of the non-distributed join algorithm matched very closely to the results obtained from the actual prototype. This validated the simulation model for the non-distributed join algorithm.

![Figure 5. Non-distributed restricted join](image)

‡A preliminary discussion of the simulation is given in Rubinovitz².

†Details of the prototype are given in Rubinovitz².
Simulation of join query processing algorithms for a trusted distributed database management system

Figure 6. Non-distributed unrestricted join

The results from the simulation indicate that the restricted join required more time than the unrestricted join for a smaller number of tuples, but the difference became less marked as the number of tuples increased. This would indicate that it is less expensive, in terms of CPU time, to polyninstantiate the tuples and subsequently remove the lower level polyninstantiated tuples and then perform a join, than to perform a join on all of the tuples in the relation. (See Figures 5 and 6.)

Second set of experiments: variable number of tuples for distributed join algorithm

The characteristics of the model for the second set of experiments were very much like those of the first set of experiments but used the distributed join algorithm instead of the non-distributed join algorithm. The results obtained from the experiments belonging to this second set matched very closely to the results obtained from the actual prototype. As before, for each experiment, two sets of results were obtained. The first set of results was for the case of the restricted join operation and the second set of results was for the case of the unrestricted join operation. The simulation results indicate that the run time is approximately the same for both cases: that is, removing the lower level polyninstantiated tuples did not have any major impact. In comparison to the non-distributed join algorithm, the distributed algorithm is approximately 20% slower. (See Figures 7 and 8.)

Third set of experiments: variable number of sites for non-distributed and distributed join algorithms

For the third set of experiments we simulated both the non-distributed join algorithm and the distributed join algorithm. However, instead of using just 3 sites as in the case of the first two sets of experiments, the number of sites used for the experiments in the third set varied from 2 to 10 in increments of 2. That is, the execution times were obtained for the two algorithms for number of sites equalling 2, 4, 6, 8, and 10. We did not vary the number of tuples. The results of the simulation matched the results measured from the actual prototype fairly closely for the non-distributed join algorithm. However, for the distributed join algorithm, the simulation results did not match the results obtained from the actual prototype as well as in the case of the non-distributed join algorithm. The main reason for this could be attributed to the fact that the implemented prototype configured the system in such a way that the logical number of sites was placed in two physical machines. That is, the actual prototype, limited by the physical number of database servers, mapped N logical nodes onto two physical nodes. In the prototype, each of the N/2 clients (each on a separate node) had to be serviced by one of the possible two database servers. This would have increased contention for the database server which was now serving additional nodes. The simulation model was configured for N number of nodes, where each node was modelled as a separate physical entity. The simulated model's behaviour would be closer to the actual results of a prototype which had N nodes with one database server per node.

The results from these experiments indicate that the overall CPU times are less for the non-distributed join algorithm than the distributed join algorithms regardless of whether the tuples were polyninstantiated or not. There is up to a 50% increase in CPU time for the non-distributed join algorithm for restricted join versus the same algorithm for the unrestricted join. The time difference, between the case of the restricted join and the case
Figure 9. Non-distributed restricted join

Figure 10. Non-distributed unrestricted join

Figure 11. Distributed restricted join

Figure 12. Distributed unrestricted join

Figure 13. Non-distributed join (restricted and unrestricted)

of the unrestricted join, is no more than 10% for the distributed join algorithm. (See Figures 9–12.)

Fourth set of experiments: large and variable number of sites for non-distributed and distributed join algorithms
This set of experiments could not be handled by the prototype due to the limited amount of resources available. The variable was the number of sites. Execution times were obtained for both the non-distributed and distributed join algorithms with number of sites equalling 10, 20, 40, 60, 80, and 100. We did not vary the number of tuples. As before, for each experiment, two sets of results were obtained. The first set of results was for the case of the restricted join and the second set of results was for the case of the unrestricted join. As the model was validated for the first and second sets of experiments, the results obtained from this model should closely approximate those obtained from an actual system.

The results of these experiments indicate that the run time for the non-distributed join algorithm grows linearly while the run time for the distributed join algorithm grows exponentially. Eliminating the lower level polynstantiated tuples made no significant impact on the CPU times for both algorithms. (See Figures 13 and 14.)

Fifth set of experiments: large and variable number of tuples for non-distributed and distributed join algorithms
This set of experiments was similar to the first and second sets of experiments. The only difference is that a large number of tuples were used. Execution times were obtained for number of tuples per node equaling 10 000, 20 000, 40 000, 60 000, 80 000 and 100 000. As with the
fourth set of experiments, this set of experiments could not be handled by the actual prototype.

The results from these experiments show that once again the non-distributed join algorithm runs faster than the distributed join algorithm, but unlike the results obtained from the previous set of experiments, the performance does not show such a large degree of discrepancy between the two algorithms. (See Figures 15 and 16.)

SUMMARY, CONCLUSIONS, AND FUTURE CONSIDERATIONS

In this paper, we first described issues on distributed query processing in a TDDBMS. Then we described two secure distributed query processing algorithms for the join queries using semi-join as a query processing tactic. Each algorithm handles N (N ≥ 1) nodes, polynormalization, fragmentation and replication. These algorithms are: (i) the non-distributed algorithm, and (ii) the distributed algorithm. The simulation of these algorithms, details of the simulation language, the simulation model and the experiments carried out were then described. Some of the results of the simulation experiments were compared with the prototype that was implemented.

It was found that, on the whole, the non-distributed algorithm performed better than the distributed algorithm for a variable number of tuples. Even when the number of sites were varied, the non-distributed algorithm gave a better performance. For each algorithm, the performance results between the restricted and unrestricted cases was compared, and here it was found that the restricted case required more time than the unrestricted case for a small number of tuples. However, as the number of tuples grew larger, it was found that the restricted case gave better performance. This is because a join operation with larger number of tuples is more expensive than one with a fewer number of tuples, even if some time is spent on eliminating lower level polynormalized tuples.

Based on the simulation results obtained, we recommend the non-distributed join algorithm for a TDDBMS. The choice between the restricted and unrestricted cases is left to the user.

The design, implementation, and simulation of join algorithms is just the first step towards developing a TDDBMS. The next step is to simulate and subsequently implement the entire DQP. In addition, update processing algorithms and transaction management algorithms have to be simulated and subsequently implemented. In the case of multi-user concurrent updates, it has been found that the traditional concurrency control algorithms, such as locking and timestamping, introduce covert channels. Such algorithms have to be adapted to function in a multilevel environment and subsequently be simulated in order to determine their performance under different conditions. The final step will be to integrate the various distributed processing components, the local TDBMSs, and the trusted network in order to obtain an operational TDDBMS.

A covert channel occurs when information is passed from a higher level subject to lower level subjects by other than normal means of communication.
The discussion in this paper has focused on a homogeneous environment where all the local TDBMSs are assumed to be designed and to operate identically. Different types of TDBMSs are being introduced into the marketplace and the secure interoperability of such systems will soon become a necessity. Therefore, future research should also be directed towards the design and development of heterogeneous TDBMSs.

ACKNOWLEDGEMENT

We gratefully acknowledge Rome Air Development Center for sponsoring the work described in this paper under contract F19628-89-C-0001. We thank John Faust and Joe Giordano for sponsoring this work.

REFERENCES

4 Thuraisingham, B 'Multilevel security issues in distributed database management systems—II' Computers and Security Vol 10 No 8 (December 1991)
5 Rubinovitz, H and Thuraisingham, B 'Design and implementation of distributed query processor for a trusted distributed database management system' J. Systems and Software Vol 21 No 1 (April 1993)
11 Gordon, G The application of GPSS system V for discrete event simulation Prentice-Hall (1975)
Third European Workshop on Software Process Technology  
EWSPT'94

February 7-8, 1994
Grenoble, FRANCE  

CALL FOR PAPERS

ORGANIZING COMMITTEE

Workshop Chair:  
Christen Fernström  
Cap Gemini Innovation, France

Programme Chair:  
Brian Warboys  
University of Manchester, UK

Programme Committee:  
Vincenzo Ambriola  
University of Pisa, Italy

Reidar Conradt  
NTH / SINTEF, Norway

Jean-Claude Devriame  
CRIN, France

Mark Dowson  
Marlstone ST, USA

Jacky Estublier  
LOI-IMAG, France

Alfonso Fuggetta  
Politecnico di Milano/CEFRIEL, Italy

Peter Henderson  
University of Southampton, UK

Claus Lewerenz  
FZI, Germany

Wilhelm Schäfer  
University of Dortmund, Germany

Ian Sommerville  
University of Lancaster, UK

The Software Process is the total set of software engineering activities to develop and maintain software products. Software Process Technology (SPT) deals with methods, formalisms and tools for supporting the Software Process. SPT is connected to disciplines such as product modelling, configuration management, groupware, cooperating transactions, interpretative systems, and rule-based systems.

The field of SPT started over 10 years ago. There are international and national workshops, symposia and conferences on SPT, but too little organized contact in the European arena. The first two European workshops arranged in Milan and Trondheim during 1991 and 1992 attracted over 40 international delegates and established a tradition, which we intend to continue at Grenoble, for state of the art technical discussions within the framework of a very informal and friendly atmosphere. The proceedings of the second workshop were published by Springer-Verlag as Lecture Notes in Computer Science 635.

The goal of this third European Workshop is to bring together SPT experts primarily from Europe to discuss:

- Novel and innovative approaches to and formalisms for SPT
- SPT implementation and interplay with other tools
- SPT system experiences, evaluations, and comparisons
- SPT and its relationships to other fields

The workshop will be an interactive forum with presentations of papers, panel sessions, and discussion groups. Papers should emphasize what is new and significant about the chosen approach, and adequately compare it with similar work. Attendance will be limited to 40 people and all participants must have written accepted full or position papers. The workshop language is English.

Participants will be selected on the basis of submitted full papers (6000 words maximum) or position papers (3 pages maximum), all of which will be reviewed by the programme committee. Proposals for panel or discussion sessions are also solicited. Full papers must be original contributions, not accepted or submitted elsewhere. It is intended, once again, to publish the proceedings with Springer-Verlag in the Lecture Notes in Computer Science series.

Send 4 copies of your submission to:

Brian Warboys  
Department of Computer Science  
University of Manchester  
Manchester M13 9PL, UK

Phone +44 61 275 6182, Fax +44 61 275 6236, Email brian@cs.man.ac.uk

Submission Deadline: 15 July 1993  
Acceptance Notification: 1 October 1993  
Camera ready paper due: 1 November 1993