A Parallel Strategy for Transitive Closure using Double Hash-Based Clustering

Jean-Pierre Cheiney *, Christophe de Maindreville **

* Ecole Nationale Supérieure des Télécommunications
46, rue Barrault, 75013 Paris, France
** Institut National de la Recherche en Informatique et Automatique
Rocquencourt, BP 105, 78153 Le Chesnay Cédex, France

network addresses: cheiney@inf.enst.fr
maindrev@madonna.inria.fr

Abstract

We present a parallel algorithm to compute the transitive closure of a relation. The transitive closure operation has been recognized as an important extension of the relational algebra. The importance of the performance problem brought by its evaluation brings one to consider parallel execution strategies. Such strategies constitute one of the keys to efficiency in a very large data base environment. The innovative aspects of the presented algorithm concern: 1) the possibility of working with a reasonable amount of memory space without creating extra Inputs/Outputs; 2) the use of on-disk clustering accomplished by double hashing; and 3) the parallelization of the transitive closure operation. The processing time is reduced by a factor of p, where p is the number of processors allocated for the operation. Communication times remain limited; a cyclic organization eliminates the need for serialization of transfers. The evaluation in a shared nothing architecture, shows the benefits of the proposed parallel transitive algorithm.

Proceedings of the 16th VLDB Conference
Brisbane, Australia 1990

1. Introduction

The efficient implementation of a transitive closure operator today appears to be one of the keys to the evaluation of recursive queries in a deductive DBMS. Numerous algorithms have been proposed [BANC86], [VALD86], [AGRA87], [HAN88], [GARD88], [IOAN88]. However, if these algorithms are examined in an environment of very large relations, two aspects are uncovered that until now have received little attention :

- the first concerns taking into account the memory space available for the operation. Tuples under manipulation are generally assumed to be held in main memory; the possibility of multiple read-operations due to memory saturation is too often either treated optimistically or not even considered.

- the second concerns the parallelization of the transitive closure operation. Even though this is a solution for executing the operation within acceptable time limits, parallel algorithms are rarely proposed.

This situation is all the more surprising because a lot of these proposed algorithms use multiple joins, either directly or implicitly. However, all of the recent works on efficient join implementation show the advantages of parallel processing and the need for taking into consideration available memory space [DEWI84]. Indeed, this guarantees a parallel execution in a single read-operation of on-disk relations. After a period of defining operations and algorithms, we feel that today it is crucial, for the sake of execution efficiency, to study physical implementations, the use of clustering and access

methods, and the unique advantages of multi-processor architectures [CHEI89].

Some recent papers approach the operator implementation problem by considering efficient transitive closure execution or using multi-processor architectures. [AGRA87] and [IOAN88] consider Input/Output minimization on direct algorithms (where transitive closure is considered as a problem of graphs). [VALD88a] proposes an execution of the operation in a parallel architecture. Transitive closure is executed in several passes and uses a two-way merge type operation from locally generated results. The relation is partitioned and, in n passes, with 2^n processor nodes, the total closure is calculated. However, the sequencing requires a coordinating node as well as a delicate balancing of overall system loading. [CHEI89] examines the efficient execution of a transitive closure that permits searches on a large number of rules.

In this paper we propose a multi-processor implementation of a transitive closure operator based on double hashing. This implementation aims to reconcile the processing of very large relations with acceptable response times. The framework is one of execution by join loops [BANC86]. Choosing a simple, well-known algorithm allows us to show more clearly the advantages of the "divide and conquer" strategy: the task is divided into a number of smaller tasks that can then be assigned to several processors. A very large transitive closure thus amounts to a collection of smaller operations. This decomposition provides:

(i) the guarantee that each operation takes place in main memory without requiring extra read and write operations due to a lack of available memory space;

(ii) the assignment of the set of operations to several parallel processors, each of which performs the same task on a section of data (multiple backend operation).

We propose an algorithm based on double hashing of a binary relation to be joined, which is named "Double Hash Transitive Closure" (DHTC). This algorithm uses direct clustering of the relation to be joined without overburdening the memory for the linearization of the transitive closure operation, and can be directly implemented in a parallel structure. In a multi-processor arrangement with a multiple backend configuration [HSIA85] in which each processor performs the same relational operation, one can expect to achieve a reduction factor of p in the processing time of large transitive closures on on-disk clustering. Data transfers between processors are minimized and a cyclic organization eliminates the need for serialization of tasks caused by an occupied bus.

After this introduction, section 2 presents the basic concept of the algorithm applied for a general transitive closure denoted R^*. Section 3 develops the parallel algorithm in a multi-processor architecture environment without shared memory. Finally, sections 4 and 5 present an evaluation of the algorithm, first from the point of view of memory space requirements, and then from the point of view of execution time. Section 6 concludes the paper.

2. A general algorithm for very large relations

In this paragraph we present the DHTC algorithm. The innovative aspects of the DHTC concern: 1) the possibility of working with a reasonable amount of memory space without creating too many Inputs/Outputs; 2) the use of on-disk clustering accomplished by double hashing; and 3) the parallelization of the transitive closure operation.

Using the same basic idea, a parallel algorithm has been proposed in [VALD88b]. In fact, this algorithm does not use a clustering technique and re-hashes the new tuples during each iteration. In this paper, no consideration is given to the main memory size.

Most of the evaluations published on transitive closures use very optimistic hypotheses for analysing the number of Inputs/Outputs needed by the execution. Indeed, for algorithms that use join loops, it is generally supposed that join operations take place in main memory. Unless additional strategies are employed, this would call for a very large amount of memory space: if one considers a join loop algorithm, the memory has to be able to accomodate the largest possible ΔR generated during the processing, where ΔR is an intermediate relation in which newly generated tuples are stored in one iteration. As for the R pages, they are read

one after the other. This hypothesis is especially overstated for certain distributions of the initial relation data. In addition, most algorithms use the set operations of union and difference for testing stop conditions. Such operations require sorts and suppression of duplicate tuples; furthermore, their efficient execution (i.e. calling for only one read operation of data from the disk) imposes severe constraints on available memory space.

When contemplating the manipulation of very large relations clustered on-disk, one can no longer consider the relation on which transitive closure is performed as a simple sequence of tuples. The main idea is to use clustering to reduce greatly the cost of the operation. Clustering characteristics, which are already largely used in the execution of other relational operators (selection, join, etc.) can likewise be exploited in transitive closure processing.

2.1. Basic concept

Let us consider a binary relation R(X, Y) where X and Y are defined on the same domain D. The relation R defines a graph G, where a node is an element of D and an edge (x,y) denotes a tuple (x,y) of R. The transitive closure R^* of the relation R consists of the transitive closure of its corresponding graph G, i.e. a tuple (x,y) is in R^* iff there exists a path from x to y in G.

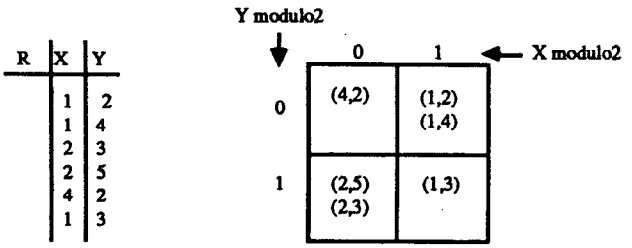
R is clustered on-disk. The size of this relation can be very large and thus, no optimistic hypothesis can be made regarding the comparison between this size and the size of available main memory. The join loop will be performed by a semi-naïve iterative algorithm [BANC86]. The major point we want to study is the limitation of Input/Output operations. In order to guarantee the linear aspect of the join operations, we want to reduce the size of the data which fits in main memory at a given time.

In an iterative algorithm, each iteration generates new tuples from R (stored on-disk) and ΔR tuples which were produced during the previous iteration. The latter may also possibly be re-written on the disk. During the initialization ΔR is composed of the set of R tuples. The generation of new tuples is based on joining the R relation with ΔR. In this configuration the use of a hash-based join algorithm is attractive [KITS83]: it permits efficient execution of the operator with reduced use of memory space [DEWI84]. In addition, the hash buckets used by the algorithm can correspond to an on-disk clustering of the relation. In order to use this possibility, the relation tuples are considered according to an on-disk clustering implemented with a hashing in n buckets, and the new ΔR tuples are considered as a set of n buckets that correspond to an identical hashing function.

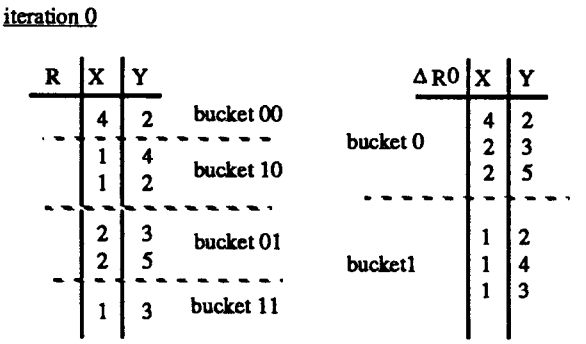
Let's look at an iteration: with a hash-based join algorithm, each relation is divided into n buckets obtained by the same function applied to the join attribute. Only buckets having the same index are joined two by two, (buckets having different indices cannot join together). However, these algorithms are insufficient for executing a join loop because the result of one step must be rehashed according to a different attribute in order to form the usable buckets for the next step. Thus, if R is only hashed (and clustered) according to Y and ΔR according to X, the join resulting from a step permits only the joining of buckets where (R.Y) modulo n = (ΔR.X) modulo n ; but in order to form the ΔR tuples used in the next step, it is necessary to rehash the result according to the new value of X (the projection on the attributes R.X and ΔR.Y is immediately computed after the join).

Our proposal permits this rehashing to be avoided. The idea is to use a multi-attribute clustering technique which provides suitable hash buckets for each iteration. In order to do this, a *double-hashed* clustering of the relation R is performed. First R is hashed by a modulo function in n buckets according to the value of X; then each of these buckets is rehashed by the same function according to the value of Y. For example, one can use a Predicate Tree technique [GARD84] which guarantees a multi-attribute, dynamic hashing (necessary in the case of expansion). The tuples of the permanent relation R are thus hashed simultaneously according to the values of both X and Y. This technique allows the relation to be looked at according to two different partitionings [CHEI86]. The first (according to the value of Y) will be used for the join algorithm for hash buckets having the same index; the second (according to the value of X) will prevent the loss of the hash value information of each tuple according to the value of X and will thus avoid the need for a write-operation hashing during the following step.

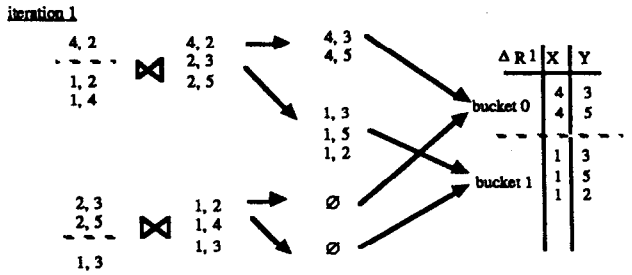
The algorithm is illustrated on the following relation R:



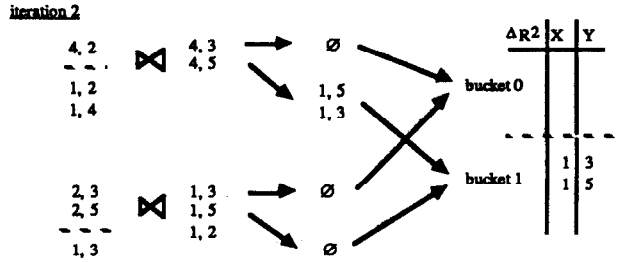
The relation is hashed in 4 buckets according to the values of X and Y. During initialization ΔR is composed entirely by R. Thanks to the on-disk double hashing, ΔR appears as 2 buckets (according to the values of X).



The first iteration of the algorithm will only perform joins between buckets 00, 10 of R and the bucket 0 of ΔR^0 on the one hand, and between buckets 10 and 11 of R and the bucket 1 of ΔR^0 on the other hand.



The results are stored directly in ΔR^1 without rehashing, according to the hash values of X which will be used during the following iteration. Iteration 2 can then proceed:



The stop condition is satisfied since no more new tuples are generated.

More generally, figure 1 represents, for the iteration p, the join between the buckets of index 3 of the R relation and the ΔR relation which has been obtained at the previous step. For this step, the join between the buckets of index 3 follows the join between the buckets of indices 0 to 2; it will be followed by the join between the buckets of index 4. The tuples of the ΔR relation which will be used at the next step are directly built without any rehashing, through accumulation of the tuples according to their hash values on X.

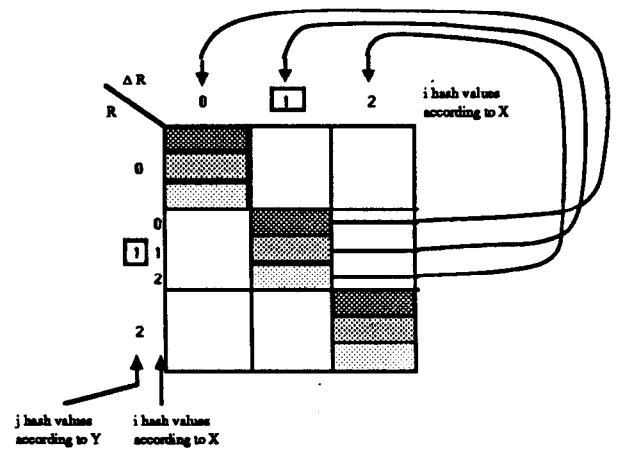


Figure 1 : *Join loop with double hashing technique*

2.2. Algorithm

We give in this section a more formal description of the algorithm.

- R is the permanent relation having the schema (X, Y)

- R is clustered on the two attributes X and Y. The hashing function used for this clustering is the same for both attributes; it is a modulo n function. The R

relation is recursively hashed into n^2 buckets. First, it is hashed in n buckets according to the X attribute, and then, each one of these bucket is hashed into n buckets according to the Y attribute. The integer i represents the hash value for X and j represents the hash value for Y. $R = \cup_{ij} R_{ij}$ $0 \le i \le n-1$, $0 \le j \le n-1$

- ΔR is a temporary relation composed of n buckets named k. ΔR_k store the tuples from ΔR having for k the hash value for the X attribute.

- $\Delta R = \cup_k \Delta R_k$ $\quad 0 \le k \le n-1$

- ΔR corresponds to one iteration of the construction of the transitive closure of R, denoted by R*. The schema of this relation is (X, Y).

The join operation is assumed to include the projection that eliminates the join attribute. The join followed by the projection, i.e. the composition of R with S, will be denoted R∘S :

$$R \circ S = \Pi_{1,4} (R \bowtie_{2=3} S)$$

```
function R* (R) : relation ;
   begin
      R* := R ;
      ΔR := R ;
      while ΔR contains new tuples do
            begin
            Z := Ø ;
            for k :=0 to n-1 do
            for i:=0 to n-1 do
               begin
                  RTik := ΔRk° Rik ;
                  Zi:= Zi ∪ RTik
               end ;
            ΔR := ∪i Zi ;
            R* := R* ∪ ΔR
         end
   end ;
```

The Z ($\cup Z_i$) relation is necessary to stack all the tuples built in one iteration: ΔR has to be assigned to $\cup Z_i$ only when all the tuples of the previous generation are generated. In fact, the tuples of Z_i are obtained from the ΔR_k , with k between 0 and n-1.

The stop condition test and the elimination of redundant processing make it necessary to determine the existence of new tuples during each iteration. The determination of tuples must always be made in pairs. By the distributive property of the union operation, this condition can be determined separately on each hash bucket. The result is formed by an AND operation of the results evaluated on each bucket. Since the buckets are composed in order to be kept in main memory, the cost of the stop condition test is greatly minimized in the DHTC algorithm. We can take advantage of the partitioning of ΔR into several ΔR_k's, thanks to the double hashing.

3. Parallel algorithm

The idea of decreasing execution times of relational operations by the intensive use of parallelism is very prevalent. Its effectiveness has been proven in numerous propositions and parallel implementations [KITS83], [DEWI86], [CHEI86]. The more recent data base machine architectures base their execution strategies around parallel algorithms. The multiple backend approach is very dominant [HSIA86]; besides the performance improvements, it provides other advantages such as reliability and possibilities for operating in a degraded mode. This approach lends itself to situations in which high-volume sequential processing is performed [GARD86], which is the case with transitive closures. One of the bases for this type of architecture is the division among the processors of the data to be processed. A simple, generally adopted way to perform this division is to form, using an appropriate function, as many hash buckets as there are processors available.

The DHTC algorithm is directly usable in such an environment. Thanks to the use of hashing, it lends itself naturally to parallelization. With a multiple backend configuration, as in [CHEI86], the join loop processing time can be divided by p, where p is the number of processors available.

The advantages of parallelization rest on the following principle: the initial relation is divided horizontally into several fragments, each of which is then processed by a separate processor. In an environment where memory is not shared ("shared nothing"), each processor processes the data from its own disk in its own memory. Each processor-

memory-disk set can be viewed as a node in a network. In a centralized multi-processor architecture, the network consists simply of a bus, whereas with a divided configuration it consists of a veritable communication network. The main problem for parallel execution of a relational operation in general, and of a transitive closure in particular, is to give a maximum amount of local tasks to each processor while limiting data and message transfers at the same time. Unlike joins, the execution of a transitive closure cannot be totally "localized" by simple horizontal partitioning of the initial relation. In fact, tuples newly-produced in a particular node at a particular moment in the processing can be required by another node in order to continue the task. Transfers are thus absolutely necessary *during* execution of the transitive closure. However, inter-processor transfers can be limited to the smallest set necessary for each step.

We maintain the idea of a division of tuples among several nodes thanks to a partitioning by hashing on the attribute that will be the join attribute. The second hashing permits the composition of the buckets to be transfered in preparation for the next iteration. The sets of tuples exchanged between processors are minimized and formed directly. The bucket indices (i.e. the hash values) represent, for each bucket, the receiving node. Without extra processing or special messages, transfers are thus performed in a minimum of time. No extra synchronization or master node is necessary.

In a multiple backend configuration, each processor performs the same action. The data are divided into as many subsets as there are processors available for the operation. If each processor has its own disk (we shall assume, for simplicity's sake, such a configuration), the data are divided among these disks by hashing. At the outset, the relation R is distributed among as many sub-relations as there are processors. This division is calculated by hashing on the X attribute. Each sub-relation is rehashed locally according to the Y values. Overall, n^2 sub-buckets are to be dealt with (hashing in n buckets on the X attribute followed by rehashing of each bucket in n buckets on the Y attribute). If p processors are available, n = p is chosen.

Let p be the processor index; the buckets of index ip from R, i between 0 and n-1, will be clustered on the disk assigned to processor p (it is supposed that there is one disk per processor). The buckets of index p from the different ΔR's, produced during each step, are likewise distributed. In this way, for one iteration, the join of buckets R_{ip} and ΔR_p remains local to each processor. No transfers whatsoever are necessary per iteration for the processing of each step of an algorithm. The processing time during this phase of the algorithm is thus divided by the number of processors p. Actually, a non-uniformity in the hashing can considerably reduce this factor because the total time used is that of the slowest processor, i.e. the processor processing the largest volume of data. This problem, shared by all hashing methods, emphasizes the importance of a careful choice of the partitioning function.

The results from each join R_{ip} |X| ΔR_p must next be processed by processor i. Indeed, these tuples have the i hash value according to X; they form the ΔR_i sub-relation for the next iteration, which will be processed by processor i. A transfer is thus necessary. However, no rehashing has to be performed. The tuples destined for processor i are already stored in a specific sub-relation, thanks to the values from the second hashing. An initial version of the parallel algorithm can be given:

```
for each processor p do
   R*p := Rp
   ΔRp := Rp ;
   while ΔRp contains new tuples do
      begin
         for i:=0 to p-1 do
            begin
               Zip := ΔRp ° Rip ;
               send Zip to node i
            end;
         receive (p-1) Zpk ;        k=0..p-1, k≠p
         ΔRp := ∪k Zpk ;            k= 0..p-1
         Rp* := Rp* ∪ ΔRp
      end ;
```

The final result (R*) will be formed simply by the union of the different R^*_p's. It must be pointed out that this union does not require a suppression of duplicate tuples. Indeed, the sets to be combined are hash buckets and are thus necessarily disjoint.

In this algorithm, each processor awaits the arrival of all of the new tuples produced during one

iteration in order to begin the next iteration (each of the p processors receives p-1 sub-relations coming from p-1 other nodes). A procedure that performs the "receive" function can easily take this role and verify that all the sub-relations have arrived.

One thing slows down this processing, however. A serialization, according to the order of write operations, can occur. In fact, if each processor p computes the buckets R_{ip} with the ordering (i:=0 to n), the serialization by the bus might slow down the entire computation: the processors compute all the buckets R_{ip} at the same time and try to send their results towards the same processor i (figure 2).

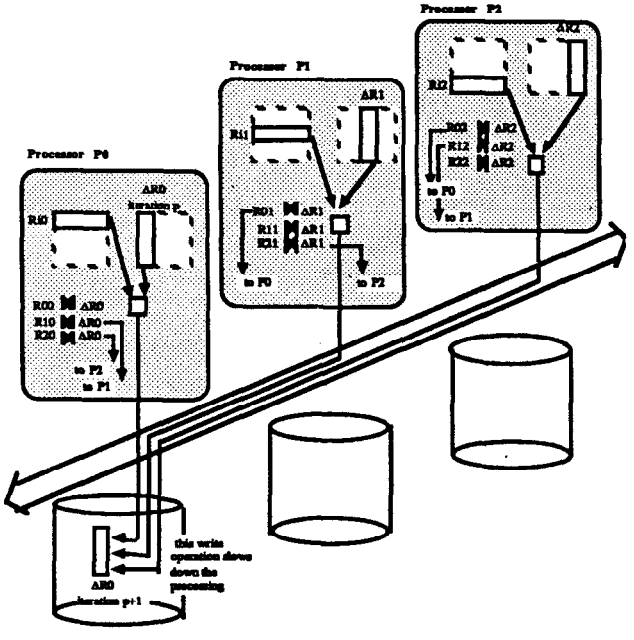


Figure 2 : *inter-processor transfers and serialization according to order of write operations*

This problem can be solved by a cyclic organization of the data transfers. Each processor first computes the bucket R_{pp} and thus forms its own result. Then it computes the following bucket ($R_{(p+1)p}$) and transfers the result to the processor p+1. Thus, the data transfer is done in a cyclic way, each processor receiving one subrelation at a time, without a slow-down caused by simultaneous write operations in one node.

The algorithm is, for an iteration :

```
{parallel join loop}
  for each processor p do
    repeat
      i:=p ;
      Z ip:= Rip ° ΔRp ;        {set of local joins}
      transfer Zip towards the disk used by the
          processor i ;
      i := (i+1) modulo p
    until i = p ;
```

The implementation can be further improved by pipelining the operations. Sub-relations transfered between nodes in fact don't need do be fully formed before being transfered. They can be transfered, page by page, as soon as they become available. Lost time due to possible loading imbalances between processors is thus minimized (such imbalances being due to a non-uniform hashing).

4. Single read execution conditions

The evaluation of the algorithm brings out two critical elements: available memory size, and the cost of data transfers between processors. In order to simplify things, we shall first give the constraints on memory size which guarantee that each local operation remains linear. Then, since these conditions are not severe, we shall assume that they are satisfied and shall proceed to evaluate the time-performance of the algorithm.

4.1. Main memory size required for a single readexecution in the case of a single processor

One difficult problem with large joins is making sure that their execution stays "linear"; each relation is read from disk only once. Therefore, this problem is very important with an iterative transitive closure algorithm, where a join is made in each iteration. The proposed algorithm reduces the memory requirement for a single read execution.

We use the following parameters to evaluate the main memory requirement:

|R| : size of R in pages ;
|M| : main memory size in pages ;
F : uniformity ratio of hashing function .

Thus, when one relation R (size |R|) is hashed in n buckets, the size of the largest bucket (or sub-relation) is F. |R| / n.

Let n be the number of different hash values on X and Y. We want to determine the size of main memory required to guarantee one single read of the R relation, for each iteration.

During the join step, the buckets of identical hash values R and ΔR must join in main memory. The smallest sub-relation (ΔRj) stays in F. |R| / n pages (the Rij pages for one i value, are read one after another). In order to have full pages, we must keep n pages for Z. They correspond to the stacking of the produced tuples during an iteration (according to the Y hash values). These pages are written on disk as soon as they are full. R is hashed in n^2 buckets.

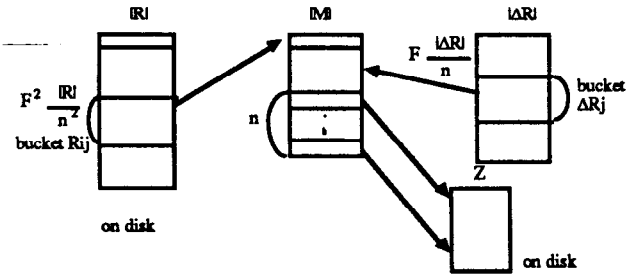


Figure 3 : *"single read" join*

The condition is :

$$|M| \geq F.|\Delta R| / n + 1 + n$$

With a cylindrical distribution of tuples [BANC86], the size of |ΔR| (new tuples generated during one step) can be considered equal to the size of the R relation. Thus, a sufficient condition is

$$|M| \geq F/n\ |R| + 1 + n$$

This condition is true when $(1 - |M|)^2 \geq 4\,F.|R|$ .This constraint is easily met, it is :

$$|M| \geq \sqrt{|R|} \qquad \textbf{condition 1}$$

The in-memory execution of each join loop step is guaranteed. If the memory is larger, the performance is improved because the Z result is kept in memory. The substitution of the old |ΔR| buckets by the new |ΔR| buckets is entirely performed in main memory. With this distribution, the relation has a maximum size |R| :

$$|M| \geq F/n\ |R| + 1 + |R|$$

let $|M| \geq (F/n + 1)\ |R| + 1$ **condition 2**

Condition 2 is harder to meet but guarantees an execution of the transitive closure in a single read.

4.2. Multi-processor case

An overall memory size of |B| = p.|M| is presently available, where |M| remains the local memory size available to a node in the network of processor-memory-disk sets. Since clustering is used, the hashing for partitioning is already done. At each node, the size of the sub-relations to be processed during each iteration is reduced, but the buckets remain the same as in the single-processor case. The memory size condition sought for is:

$$|B| \geq p\sqrt{|R|}$$
condition 3

We notice that the multi-processor configuration doesn't permit a reduction in local memory size. For a high number of processors, this situation brings about a significant increase in the cost of the architecture. However, if condition 3 is not true, it is possible to do a local re-hashing. This re-hashing implies two additional read-write operations. Indeed, they can be avoided by pipe-lining the transfers and the re-hashing [CHEI86]. On the other hand, it should be pointed out that each processor processes only one bucket out of the p buckets that form the relation R and the ΔR.

5. Analysis and comparisons

This evaluation concerns the parallel processing of transitive closure. In this section we analyze the DHTC algorithm's performance and compare it to the performance of: 1) a simple iterative algorithm (Iterative Transitive Closure or ITC); and 2) Valduriez and Khoshafian's Parallel Transitive Closure (PTC) algorithm [VALD88a]. In order to make the comparison simpler, we shall use the same model and hypotheses as [VALD88a]. We thus assume that new tuples are uniformly

produced, both by each processor and during each iteration of the join loop.

Likewise, we assume that the Input/Output times are identical for the three algorithms. This assumption is an optimistic hypothesis in favor of the ITC and PTC algorithms because they both require a large amount of memory space if on-disk re-read and re-write operations are to be avoided. In fact, the production time of new tuples taken into account in this evaluation includes both the processing and the necessary Inputs/Outputs. We consider this time as directly proportional to the number of new tuples produced, independent of the number of basic tuples processed (which are assumed to be read only once thanks to a sufficient amount of main memory). This hypothesis imposes much more substantial size requirements on the main memory for the ITC and PTC than for the DHTC.

5.1. Response time

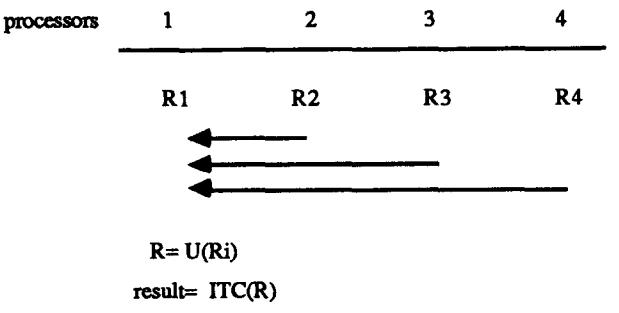
In the following analysis, we shall use the following parameters :

R	:	number of R tuples ;
t	:	time to produce a new tuple ;
Rnew	:	number of new tuples produced by the transitive closure ;
trf	:	time to transfer one tuple ;
p	:	number of processors or nodes ;
msg	:	time to transfer one message ;
d	:	number of join loop iterations ;

In this evaluation, we assume that the transfers do not saturate the network. The response time can be broken down into two parts: communication time and processing time. As for communication, messages between processors must be considered. These messages correspond to the operation of the algorithm's "send" and "receive" functions. Thus, the necessary time for transfering a bucket of n tuples equals n.trf + msg.

Algorithme ITC

The ITC algorithm is performed by a single processor, after all the sub-relations on this node have been returned to their respective nodes of origin.



The response time consists of the communication time plus the processing time. The communication time corresponds to (p-1) times the transfer time of one sub-relation. In fact the serialization of transfers is inevitable here because one single node must receive all of the buckets. The communication time is thus:

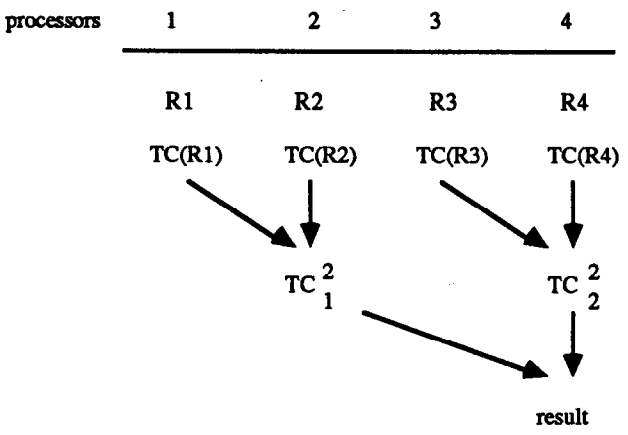
$$(p\text{-}1)\left(\frac{R}{p}.trf + msg\right)$$

The processing time corresponds to the the production time of new tuples, i.e. simply : Rnew.t
The response time RT of the ITC algorithm is thus:

$$(p\text{-}1)\left(\frac{R}{p}.\ trf + msg\right) + Rnew.t$$

PTC algorithm

Valduriez and Koshafian give in [VALD88a] a complete evaluation of their algorithm. Let us recall that the PTC algorithm performs the transitive closure of a relation R distributed among p nodes, in $\log_2 p$ passes. During each pass, a "fusion" of two previous local results is performed. Redundant processing is avoided in this "fusion". With the same parameters, the number of tuples produced during each pass is [Rnew / ([$\log_2$ p + 1]] where [x] indicates the integer part of x. The sequence of processing and transfers is illustrated below:



On the whole, the final cost is [VALD88a] :

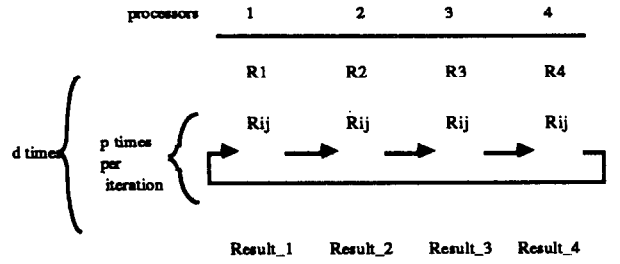
$$RT(ptc) = \sum_{i=1}^{\log_2 p} \left(\left(\frac{R}{p} + \frac{2^{i-1}}{p}.DL \right) trf + msg \right) + \sum_{i=1}^{\log_2 p} \frac{2^{i-1}}{p}.DL.t$$

{ transfer time } { processing time }

where DL is the number of new tuples produced in one pass.

DHTC algorithm

The number of new tuples generated during each iteration equals Rnew/d. Rnew/pd new tuples are generated in parallel per node and per iteration. These Rnew/pd tuples are seen in the form of p distinct sub-buckets which correspond to the p receiving nodes (actually p-1, since one sub-bucket stays where it is). The size of the sub-buckets sent is thus $Rnew/dp^2$. p of these buckets are sent together without serialization (cyclic organization). Overall, a small bucket of size $Rnew/dp^2$ is sent p times per iteration. It's this decomposition into small buckets that makes it possible to avoid the rehashing and the Inputs/Outputs.



The transfer time of all the tuples together is thus :

$$p.d\left(\frac{Rnew}{d.p^2}.trf + msg\right)$$

With the hypothesis of uniformity, all the processors produce new tuples in parallel, and the processing time is divided by the number of active processors :

$$\frac{Rnew.t}{p}$$

Overall the expression for the response time is :

$$RT(dhtc) = \frac{Rnew}{p}.\ trf + p.d.msg + \frac{Rnew}{p}.t$$

5.2. Performance comparisons

Two general remarks can be immediately formulated. First of all, it is noticed that DHTC requires more messages than the two other algorithms. This number equals the depth of the join loop times the number of parallel transfers during one iteration. Indeed, DHTC performs more transfers (hence the increased number of messages), but the buckets are smaller (hence the certainty of not having to write and re-read the buckets on-disk). And secondly, a close look allows one to notice that the result is localized differently according to the algorithm: for ITC and PTC, it's entirely located in one node, while for DHTC it's distributed in p disjoint buckets over p nodes. Two different architectures must therefore be considered. The first concerns a multiple backend operation, where each node constitutes a backend processor. A host processor submits the operation and receives the result. In this case, the response times mentioned in the previous section are directly applicable. The second concerns an operation where each node constitutes a site of a system. In this case, the result is requested at a particular site, and, for DHTC, the time to transfer the result to the final site must be added in. This case is of less interest for DHTC. This analysis will compare the performance of a multiple backend architecture implementing the three algorithms as well as the case of a utilization where the result is requested at a determined site.

The following values are chosen for the comparison:

R = 1,000,000 tuples	msg = 1ms
Rnew = 2,000,000 tuples	trf = 5 µs
	t = 0,2 ms

The first curve shows the effect of the depth d of the join loop. It is noticed that this effect is only significant for a very large number of iterations and for a considerable number of processors. In current situations (up to 100 processors with depths of 100 loops), d is not a determining factor. We shall therefore neglect its effect in the rest of the evaluation and we shall choose an average value (d=100) for the comparisons.

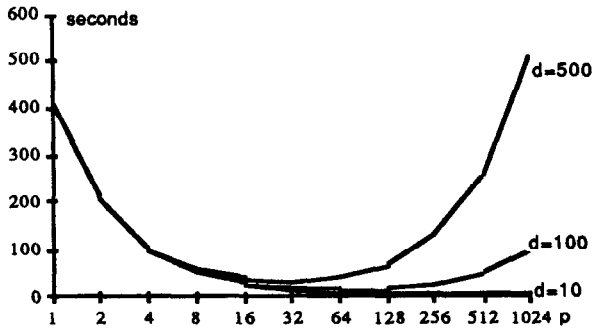


Figure 4 : *effect of the depth of the join loop*

The figure below permits a visual comparison of the response times of ITC, PTC and DHTC as a function of the number of processors.

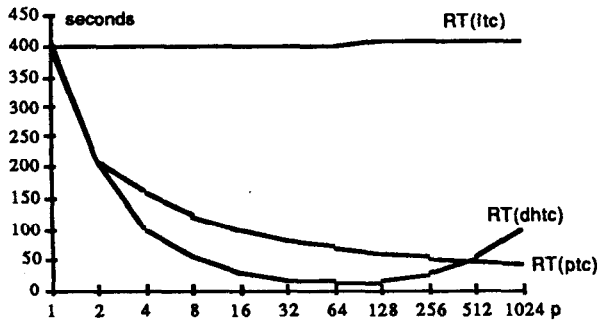


Figure 5 : *performance as a function of the number of nodes*
(multiple backend configuration, Rnew=2,000,000)

The elevated number of messages induced by the DHTC algorithm limits its possibilities when the number of processors becomes very large. The communication times between processors therefore become prohibitive. In fact, the message cost can be strongly reduced with, for example, new transputer-based machines [KUBL88]. A clear superiority is noticed however for the range in number of processors in current use. DHTC thus shows a performance improvement factor of two to four over PTC for a number of processors between 4 and 128, which are typical values in present multi-processor configurations.

In order to examine the performance of DHTC in an architecture where the result must be recomposed at a site, the transfer time for the (p-1) results available at the other sites can be added to RT(dhtc) :

$$(p\text{-}1)\left(\frac{Rnew}{p}.trf + msg\right)$$

The response times in a utilization where the result is composed at a single site can therefore be plotted:

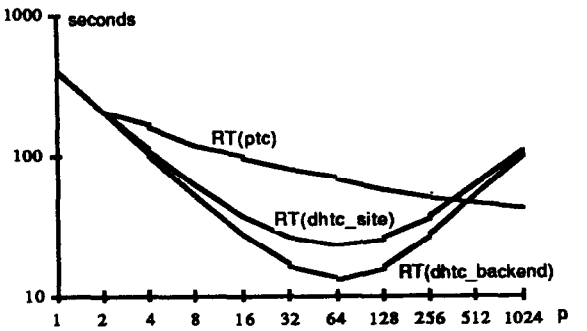


Figure 6 : *performance as a function of the number of nodes*
(result at one site, Rnew=2,0000,000)

As anticipated, RT(dhtc_site) is less performant than RT(dhtc_backend); however DHTC remains better than PTC for all configurations in current use.

6. Conclusions

We have presented in this paper a solution for efficiently implementing the transitive closure of a very large relation stored on-disk. The algorithm executes a join loop for which we propose an optimization based on clustering of the relations and parallelization. Thanks to a double hashing, the size of the sub-relations to be manipulated together in memory is reduced and a linearization of joins with substantially relaxed main memory size constraints can be guaranteed. The processing is

divided into p parallel operations divided among p processors. This article goes only so far as to consider a semi-naïve algorithm for transitive closure. Our future research will consist of doing a more in-depth analysis of other possibilities for parallel executions.

7. References

[AGRA87] R. AGRAWAL, H.V. JAGADISH: *"Direct Algoritms for Computing The Transitive Closure of Database Relations"*, 13th VLDB, Brighton ,1987.

[AGRA89] R. AGRAWAL, A. BORGIDA, H.V. JAGADISH: *"Efficient Management of Transitive Relationships in Large Data and Knowledge Bases"*, ACM SIGMOD, Portland, June 1989.

[BANC86] F. BANCILHON, R. RAMAKRISHNAN: *"An Amateur's Introduction To Recursive Query Processing Strategies"* , ACM SIGMOD, Washington, May 1986.

[CHEI86] J.P. CHEINEY, R. MICHEL, P. FAUDEMAY, J.M. THEVENIN : *"A Reliable Multiple Backend Using A Select-join Operator"*, 12th VLDB, Kyoto, Aug 1986.

[CHEI89] J.P. CHEINEY, C. DE MAINDREVILLE: *"Relational Storage and Efficient Retrieval of Rules in a Deductive DBMS"*, 5th Int. Conf. on Data Engineering, Los Angeles, 1989.

[DEWI84] D.J. DEWITT et al: *"Implementation Techniques for Main Memory Database Systems"*, ACM SIGMOD, Boston 1984.

[DEWI86] D.J. DEWITT et al: *"GAMMA- A Hight Performance Dataflow Database Machine"*, 12th VLDB, Kyoto, Aug 1986.

[GARD84] G. GARDARIN, P. VALDURIEZ, Y.VIEMONT: *"Predicate Trees: A Way for Optimizing Relational Queries"*, Computer Engineering Conf., Los Angeles, 1984.

[GARD86] G. GARDARIN: *"Efficient Processing of Very Large Databases : A Comparative Analysis of Architectures"* , IFIP, 1986.

[GARD88] G. GARDARIN, P. PUCHERAL: *"A Graph Operator to Process Efficiently Linear Recursive Rules in Main Memory Oriented DBMS"*, 3rd Database Bresilian Symposium, Recife-Pernambuco, March 1988.

[HAN88] J. HAN, G.QADAH, C. CHAOU: *"The Processing and Evaluation of Transitive Closure Queries"*, Proc. of EDBT, Venice, 1988.

[HSIA85] D.K. HSIAO, S. DEMURJIAN: *"Benchmarking Database Systems in Multiple Backend Configuration"*, A Quatterly Bulletin of the IEEE Computer Society, Technical Comitee on Database Systems, V8, N°1, 1985.

[IOAN88] Y. IOANNIDIS, R. RAMAKRISHNAN : *"Efficient Transitive Closure Algorithms"*, 14th VLDB, Los Angeles, 1988.

[KITS83] M. KITSUREGAWA et al : *"Application of Hash to Database Machines"*, New Generation Computing, N°1, 1983.

[KUBL88] F.D. KUBLER: *"Cluster Oriented Architecture for the Mapping of Parallel Processor Networks to High Performance Applications"*, Int. Conf. on Super Computing, St Malo, France, 1988.

[SCHN89] D. A. SCHNEIDER, D.J. DEWITT : *"A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment"*, ACM SIGMOD, Portland, June 1989.

[VALD86] P. VALDURIEZ, H. BORAL: *"Evaluation of Recursive Queries Using Join Indices"*, Int. Conf. on Expert Database Systems, Charleston, South Carolina, April 1986.

[VALD88a] P. VALDURIEZ, S. KHOSHAFIAN: *"Transitive Closure of Transitively Closed Relations"*, 2nd Int. Conf. on Expert Database Systems, Tysons Lorner, Virginia, April 1988.

[VALD88b] P. VALDURIEZ, S KOSHAFIAN: *"Parallel Evaluation of the Transitive Closure of a Database Relation"*, Int. Journal of Parallel Programming, Vol 17, N°1, Feb. 1988.