Integrity Constraint Checking in Distributed Nested Transactions over a Database Cluster*

Stephane Gançarski[1], Claudia León[2], Hubert Naacke[1], Marta Rukoz[2] and Pablo Santini[2]

[1]Laboratoire d'Informatique Paris 6, Université Pierre et Marie Curie.
8 rue du capitaine Scott, 75015, Paris, France.
{Stephane.Gancarski,Hubert.Naacke}@lip6.fr

[2]Centro de Computación Paralela y Distribuida, Universidad Central de Venezuela.
Apdo. 47002, Los Chaguaramos, 1041 A, Caracas, Venezuela.
{cleon,mrukoz}@ciens.ucv.ve, psantini@eda.com

Abstract

This paper presents a solution to check referential integrity constraints and conjunctive global constraints in a relational multi database system. It also presents the experimental results obtained by implementing this solution over a PC cluster with Oracle9i DBMS.

The goal of those experimentations is to measure the time spent to check global constraints in a distributed systems. The results show that the overhead induced by our distributed constraint checking is reduced by 50% compared to a centralized checking of constraints.

Keywords: Multi Databases, Integrity Constraints, Transaction Systems, Distributed Systems.

Introduction

Multidatabase Systems (MDBS), where different sites (or nodes), connected by large scale (e.g. Internet) or local (e.g. Fast Ethernet) network, store parts of a global database are nowadays widely used: distributed databases, database clusters, mediation based systems, databases over peer-to-peer networks. A major issue with such systems is to maintain data quality with respect to transactions addressed to the MDBS. One of the most popular way of maintaining data quality is to define and maintain integrity constraints, which are logical assertion which must be satisfied by the database. An integrity constraint may be local, i.e. it relates data stored on a single node. Otherwise, it is a global constraint, i.e. it relates data distributed over different sites. Maintaining global constraints is a challenging performance issue, since it involves a distributed evaluation for checking the constraints. Data involved by such a constraint being distributed over various nodes, it is necessary to determine on which node(s) the constraint must be checked to minimize the checking process duration, by minimizing the quantity of information transferred through the network, and thus optimize transaction response time. In [2], we propose a classification of integrity constraint and give, for each constraint type, strategies for integrity checking in an object oriented multidatabase, according to the nodes on which updates are sent.

In this paper, we propose a mechanism for checking global constraints in a relational Multidatabase. As in [2], user transactions as well as the integrity checking process are nested transactions, according to the model proposed by Moss [6]. We first focus on referential integrity constraints, then, we address global conjunctive constraints which are a generalization of referential integrity constraints. The core idea of our solution is to distribute and parallelize the checking process whenever a local checking is not sufficient.

* This work was financed by CDCH-UCV, FONACYT - Venezuela and CNRS - France

We also show the performance evaluation results obtained on a database cluster where each node runs an instance of Oracle9i

The remainder of the paper is structured as follows. Section 1 presents the main concepts used in our approach. Section 2 describes our checking strategies while implementation and experimentations are presented in Section 3. In Section 4, we present and discuss the results we obtained. Section 5 concludes.

1. Context and main concepts

A Multidabase system allows for data manipulation over different database parts, each one being managed on a node by a local DBMS. There are different possible architectures, with different levels of integration of the local DBMSs, corresponding to different levels of global services [8]. In this paper, we propose mechanisms for maintaining global integrity constraints over a relational homogenous multidatabase system which includes the following features:

- All the transactions are global and handled by a global transaction manager based on a global schema describing data and its localization. In other words, we do not consider the case where local transactions can be executed by local transaction managers out of the control of the global transaction manager. This latter manages the concurrency control between transactions and between sub-transactions of a nested transaction.

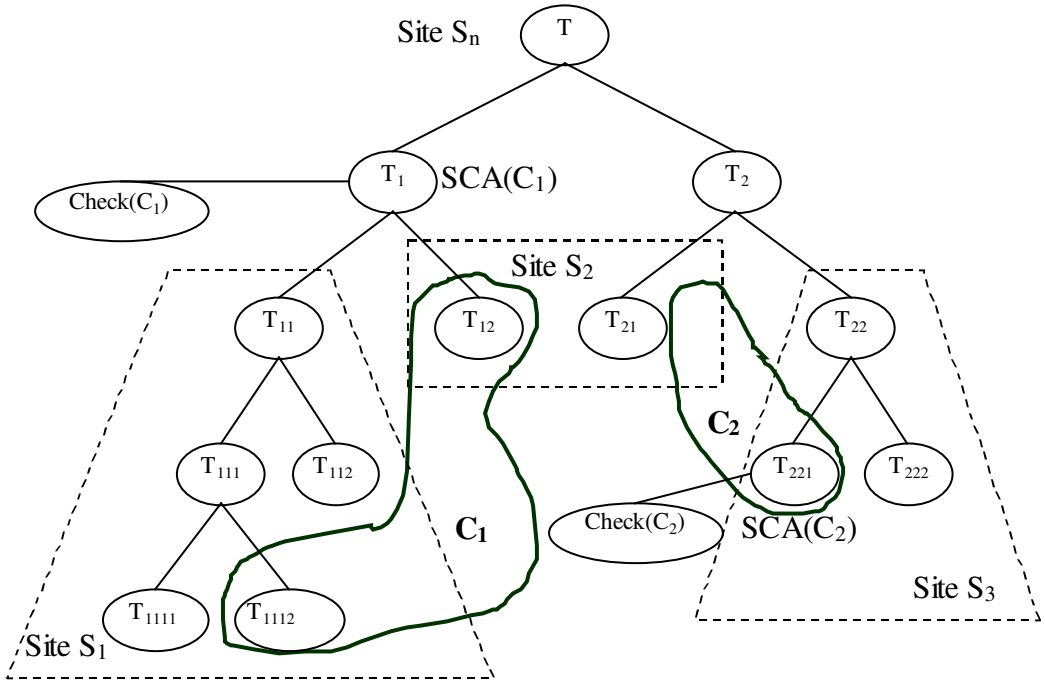


Figure 1. Distributed nested transaction and "touched" integrity constraints

- We use the (closed) nested transaction model originally proposed by Moss [6]. A nested transaction can have as many as necessary sub-transactions which themselves can be composed of sub-transactions and so on. Thus, a nested transaction can be represented by a transaction tree which represents the parent-child hierarchy among sub-transactions. Moss described the properties of the nested transaction model and algorithms for concurrency control and validation. It also proposed a simplified version of the model, where only leaf sub-transaction can access stored data, each leaf sub-transaction being executed on a single node and thus, only accessing data stored on that node. We use

this simplified version of the model, since it simplifies the transaction control and, as demonstrated by Moss, it is as expressive as the original version.

Sub-transactions of a nested transaction are assigned to the different nodes according to the distribution of the data. When a nested transaction is initiated on a site S_i its sub-transactions are recursively initiated on the same site. When one of those sub-transactions is the root of a sub-tree the leaves of which must be executed on a same site S_j, different from S_i, then the whole sub-tree is sent to S_j where it will be fully executed. This assignation policy is illustrated on Figure 1.

2. Checking global integrity constraints over relational multidatabases manipulated by nested transactions

As mentioned in the introduction, the main issue for checking global constraints in a multidatabase environment is to determine where to check the constraint while minimizing the amount of data transferred through the network. In [2], we address this issue in the context of OO multidatabase with nested transactions, and propose a strategy based on the constraint type, update type and update localization. As the goal of this paper is to adapt this strategy to relational multidatabases, we recall here the main principles of our strategy proposed in [2]:

- The checking of a constraint C_i against the effects of a nested transaction T is controlled by the smallest common ancestor, within the tree representing T, of all the leaf sub-transactions *touching* the constraint. Briefly speaking, a transaction touches a constraint if it is likely to violate it, i.e. it updates data involved by the constraint. The set of subtransaction that touch C_i is denoted $Touch(C_i)$, and their smallest common ancestor which controls C_i is denoted $SCA(C_i)$. The checking of C_i starts as soon as no more subtransaction likely to violate it is running, i.e. as soon as all the subtransactions in $Touch(C_i)$ have finished.

- Since each subtransaction controls the execution of its subtree [6], it is possible to guarantee that in case of violation of C_i, we abort not only the subtransactions in $Touch(C_i)$, but also the subtransactions that have already used the "dirty data" produced by subtransactions of $Touch(C_i)$. On opposite to flat transactions systems, the nested transaction can continue its execution with its subtransactions which are not related with the constraint, i.e. the subtransactions which are not in $Touch(C_i)$ and which have not used results produced by subtransactions in $Touch(C_i)$.

- There is a direct communication between elements of $Touch(C_i)$ and $SCA(C_i)$,which allows checking C_i as soon as possible, i.e. as soon as all the subtransactions have finished. In case of violation, a message is propagated downwards through the subtree of $SCA(C_i)$ to provoke the abort of relevant subtransactions in order to maintain the database integrity.

- The checking of C_i is performed against the objects modified by a nested transaction T_j, i.e. the process $Check(C_i)$ which checks C_i uses the effects of the transactions in $Touch(C_i)$ that have decided to commit. We denote $Effect(T_j,C_i)$ the set of such effects. $Check(C_i)$ is nested transaction, child of $SCA(C_i)$. Due to the way subtransactions are assigned to sites, $Check(C_i)$ is initiated either on the site $SCA(C_i)$ is initiated or on a site S_k, if all the subtransaction in $Touch(C_i)$ are assigned to S_k (see Figure 1).

For sake of efficiency, we must determine on which site the processes triggered by check will be executed, in order to minimize the amount of data to be transferred through the network. We must take into account the nature of constraint C_i as well as the location of the subtransactions in $Touch(C_i)$. This issue is illustrated on Figure 1 where the "area" concerned by a constraint (data involved by the constraint and subtransactions touching the constraint) is delimited by a bold line. For instance, C_1 involves data on sites S_1 and S_2 and is touched by T_{1112} on S_1 and by T_{12} on S_2. C_2 involves data on sites S_2 and S_3 and is touched by T_{221} on S_3. This example shows that it is necessary to minimize the quantity of data to transfer between the sites concerned by the checking of a constraint. More precisely:

- For constraint C_2, though the unique leaf subtransaction that touches it is located on S_3, it is likely that the checking process should be executed on S_2. For instance, if C_2 is a referential integrity constraint, its checking requires comparing the data modified on S_3 by T_{221} with data stored on S_2 related with the constraint. Evaluating C_2 on S_2 only requires transferring data modified on S_3 to S_2, while evaluating it on S_3 could require transferring all the data of S_2 involved by the constraint, even the data that is not related, directly or indirectly, by the transaction.
- For constraint C_1 the situation is even more complex. Its evaluation can be executed on S_1, on S_2, or on both sites, depending on the nature of the constraint and on the operations performed by T_{1112} and T_{12}.

2.1 Using intersite sets

An usual method for optimizing the checking process of universally quantified constraint is the following. Objects modified by a transaction and involved by a constraint are collected at transaction execution time, so that the constraint is only checked against those objects. In a distributed context, it is necessary to minimize the transfers of such objects through the network, to avoid creating a bottleneck during the execution of distributed transactions. To this end, we proposed in [2] to parse each global constraint C in order to determine intersite sets which minimize the number of objects to be transferred for checking the constraint. Generating intersite sets is made through the two following steps:

First step: the first step is inspired by the work of Gupta and Widom [4] which consists in deriving, wherever possible, a *locally computable predicate*. This predicate is locally evaluated on a single node with object locally modified. If this predicate is satisfied, this is sufficient to guarantee the satisfaction of the global constraint C. From now on, we denote PL_i such a predicate, where S_i is the site on which PL_i is defined and C is the constraint to check.

Second step: the second step is based on the approach of Grufman et al. [3]. They propose to decompose each universally quantified constraint into a conjunction of intersite predicates. The idea is to obtain a sub-constraint for each site, which describes the responsibility of the site with respect to the satisfaction of the global constraint, and an additional constraint which relates all the intersite predicates with the constraint to check. This additional constraint is stored on the site where the final checking of the constraint will occur. In [2], we propose such a decomposition where $C\_intersite_i$, the set associated with the intersite predicate of S_i, only contains objects modified on S_i and which do not satisfy the locally computable predicate associated with S_i and C.

2.2 Generating nested transactions to implement the checking process

The process $Check(C)$ which implements the checking of constraint C with respect to nested transaction T, is initiated by $SCA(C)$, the subtransaction of T in charge of controlling the checking of C, on the site assigned to $SCA(C)$, denoted $Site_{SCA(C)}$. $Check(C)$ initiate the root of a distributed nested transaction on $Site_{SCA(C)}$, having a child $Check(CS_i)$ for each site S_i assigned to a leaf sub-transaction of T touching C.

Each $Check(CS_i)$ has the following behavior:

1. Check the locally computable predicate PL_i , if exists, through $Check(PL_i)$ process. If PL_i is satisfied, the process ends since this means that C can not be violated by the modifications made on S_i.
2. If PL_i is not satisfied, $C\_intersite_i$ is built. $C\_intersite_i$ contains the objects of S_i required to continue the checking of C on every site S_j containing data involved by C and is thus sent to S_j.
3. Last, $Check(S_i)$ initiate on S_j the process $Check(RG_{ij})$ which checks the satisfaction of C with respect to the objects in $C\_intersite_i$ and to the data stored on S_j.

Nested transaction *Check*(*C*) is distributed over different sites: sites containing leaf sub-transactions of *T* touching *C*, and sites where *C* must be checked because they contain data involved by *C*. Figure 2 shows such a nested transaction for the constraint C_1 of Figure 1. Note that the nested transaction for checking C_2 would have only one subtree (instead of two for C_1) since C_2 is touched by leaf sub-transactions all executed on a single site, S_3.

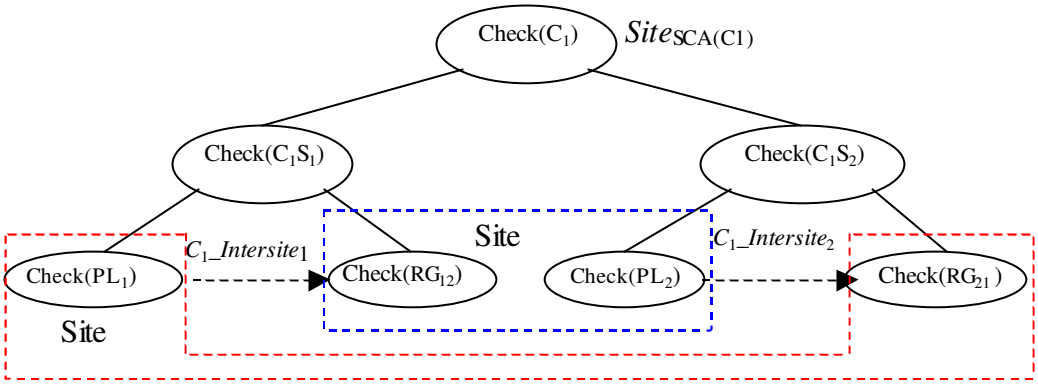


Figure 2. Checking of a global constraint touched by sub-transactions on two sites

A global constraint may contain predicates and quantifiers requiring to process checking on several sites, through transactions exchanging information. Such a constraint may have an arbitrary complexity which makes it difficult (almost impossible) to define a generic algorithm for generating the nested transaction that checks the constraint. Thus, we restrict our solution to two constraint classes: global foreign key constraints (referential integrity) and global conjunctive constraints. Global conjunctive constraints are more general constraints with relate data located on different sites. We present the method for checking those two constraint type in the next two subsections.

2.3. Referential integrity constraints

Referential integrity constraints are defined between the primary key of a table and a set of columns, called foreign key, of another table. This type of constraint is usually maintained by existing DBMS in client/server architectures, we present here a method to maintain them in a distributed environment.

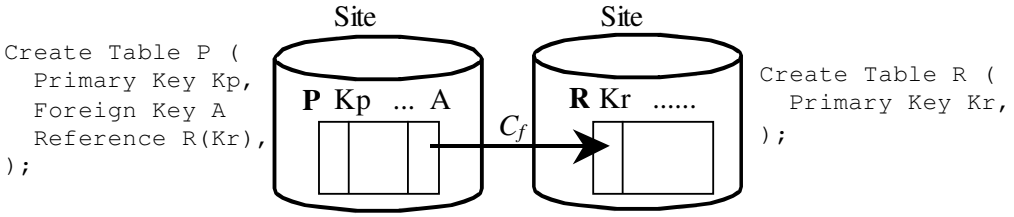


Figure 3: Referential integrity constraint between tables P and R.

Figure 3 shows an example of a referential integrity constraint Cf , which can be expressed as follows:

$$\forall\, p \in P, \exists\, r \in R,\ r.\mathrm{Kr} = p.\mathrm{A}.$$

Note that the only operations likely to violate C_f are update or delete on table R and update or insert on table P. In the following, we show our method applied to two examples: a delete on R and an insert on P.

2.3.1. Removing a primary key

Figures 4 and 5 show the pseudo-SQL code of the nested transaction that maintains the constraint C_f when a operation DELETE R WHERE <condition> is executed. We assume in Figure 4 (resp. Figure 5) that C_f is defined with the Restrict (resp. Cascade) option. In both cases, tuples deleted on site 2 may be

referenced by tuples on site 1. In case of the Restrict option, the transaction is aborted if there exist such tuples (Figure 4). In case of the Cascade option, such tuples are deleted on site 2.

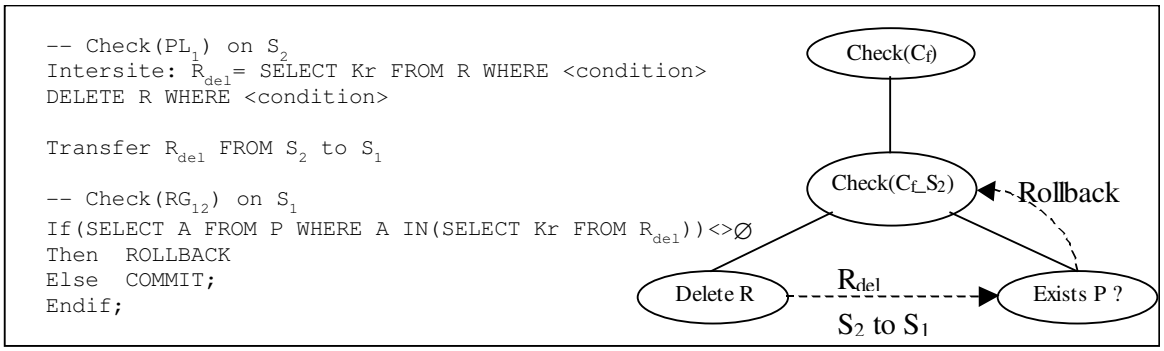


Figure 4: Checking an referential integrity constraint with RESTRICT option upon deletion of tuples in R

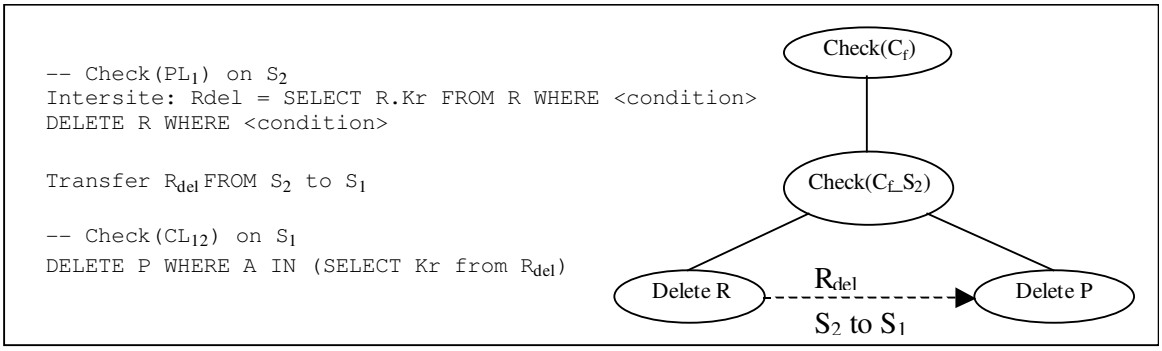


Figure 5: Checking an referential integrity constraint with CASCADE option upon deletion of tuples in R

2.3.2. Inserting a foreign key

When a tuple is inserted into table P, we must guarantee that there exists a tuple in R with the corresponding primary key value. Figure 6 shows the pseudo-SQL code of the nested transaction that maintains the constraint C_f when a operation INSERT INTO P is executed. For sake of simplicity, we assume that the inserted tuples correspond to the result of query SELECT * FROM P WHERE <condition>.

To check referential integrity constraints, we take into account the following points:

- In case of a *Delete* on R with cascade option, we need not only to check the constraint, but also maintain it by deleting all the tuples in P with a foreign key equal to the primary key of the deleted tuple in R.
- The case where two concurrent transactions (belonging to the same nested transaction or not) introduce inconsistency by respectively inserting a tuple into P with A=x and deleting a tuple in R where Kr=x is impossible, at least if the DBMS concurrency control is based on (write) locks, as it is the case with Oracle*9i*.

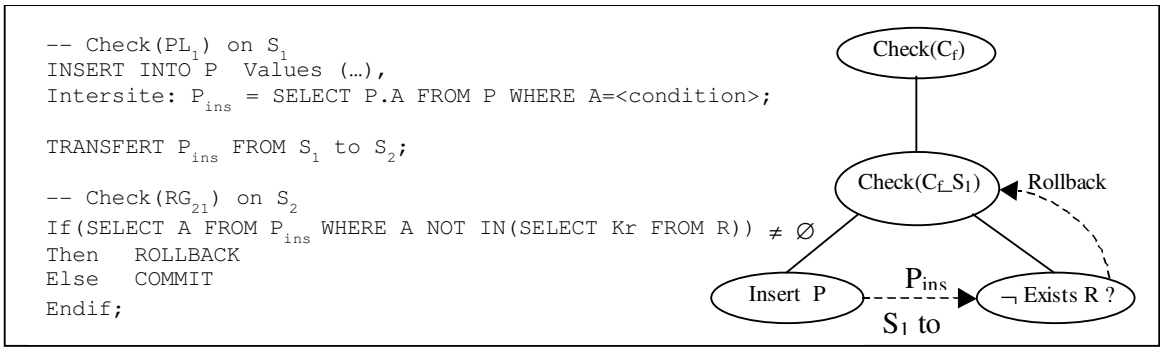


Figure 6: Checking an referential integrity constraint upon insertion of tuples in P

2.4. Global conjunctive constraints

Global conjunctive constraints are an important class of constraint useful to relate data possibly located on different sites. We give here an example, the reader may refer to [2, 3] for a more detailed description. Consider the distributed database of Figure 7, where the arrow illustrates the fact that attribute F on S_1 is a reference to the table R on S_2. A global conjunctive constraint C_f that uses this reference could be:

$$\forall(p \in P, r \in R),\ p.D = value_1 \ \wedge\ p.\mathrm{F} = r.\mathrm{Kr} \Rightarrow r.\mathrm{N} = value_2$$

C_f imposes a defined value for attribute N for every tuple in R related through F with a tuple in P with a given value for attribute D. For instance, assume that P describes products and R describes Firms, such a global constraint may be that a firm which produces a given product type must have a security level above a given threshold.

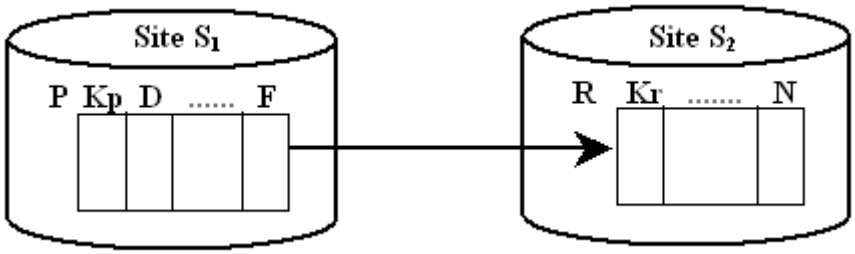


Figure 7: Relation between R and P through reference F

Figure 8 presents the pseudo-SQL code of the nested transaction *Check(C_f)* that maintains the constraint C_f when a transaction that updates table *P* is executed on node S_1. The execution of another transaction that updates table *R* on node S_2 would generate a similar nested transaction, by inverting the role of S_1 and S_2. The execution of a transaction that updates both tables would raise a nested transaction for checking C_f similar to the one shown on Figure 2.

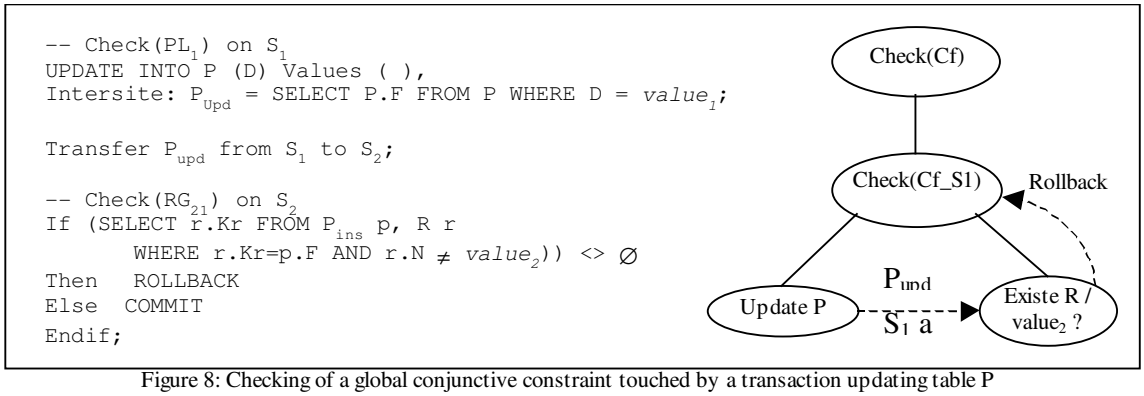


Figure 8: Checking of a global conjunctive constraint touched by a transaction updating table P

3. Performance evaluation of checking global conjunctive constraints

In order to evaluate our integrity checking mechanism, we focus on the checking of global conjunctive constraints. We measure the time spent by the system to execute the checking process, including data transfers between sites to prove the feasibility of our method in distributed environments. For sake of simplicity, we assume that the nested transaction that may violate a constraint is reduced to the subtree formed by the set of leaf subtransactions touching the constraint, noted *T(C)* and their common ancestor *SCA(C)*. We also consider that the leaf sub-transactions are all executed in parallel and use the concurrency control method provided by Oracle*9i* [7].

In this context, the whole response time of a nested transaction is decomposed into the following three parts that we measure separately:

- **Update time:** this time corresponds to the execution time of the leaf subtransactions, including the updates made and the generation of intersite sets. Since leaf subtransactions are executed in parallel, the update time is the maximum of the leaf subtransaction execution times.
- **Transfer time:** this is the time spent to transfer intersite sets from one site to another. Again, since intersite sets are transferred in parallel, the total transfer time is the maximum execution time off all threads performing intersite set transfers.
- **Checking time**: this is the time spent to check the constraint. It corresponds to the maximum time of all the thread performing the checking of the constraint. It is worth noting that we consider the worst case, i.e. the case where the constraint is not violated and thus the checking process must be executed for all the involved data.

The **total time** is thus the sum of the three times defined above. We choose it as the metric for our experiments, since each part includes actions related with the checking process.

3.1. Experimental environment

We ran our experiments on a seven nodes PC cluster at Central University of Venezuela of Caracas. Each experiment used three nodes simultaneously. Each node has a Intel Pentium Pro III 1 GHz processor, 256 MB SDRAM core memory and a 40 gigabytes hard-disk. Nodes are connected by a fast Ethernet 100 Mb/s network. Each node runs Red Hat Linux Advanced Server Release 2.1 AS (Pensacola) Kernel 2.4.9-e.3. The local DBMS is Oracle9*i* Release 2 (9.2.0.1.0) on each node. Programming and query languages are Java 2 Standard Edition J2EE version 1.4.2, Oracle9*i* PL/SQL, Oracle SQL and Java Database Connectivity JDBC 3.0 API.

3.2. Case study

Experiments were led on the example introduced in Section 2.4 and represented on Figure 9. The database is composed of a table *Products* stored on one site S_1 and a table *Firms* stored on another site S_2. Attribute *ProdBy* of table Products is a reference to the firm which produces the product. There is one global conjunctive constraint over this database, which states that a firm which produces a product with restricted delivery must have a (security) level equal to 33. It can be expressed by the following formula:

$$C_f : \forall(p \in \text{Products}, f \in \text{Firms}),\ p.\text{Deliv} = \text{“restricted”} \wedge p.\text{Prodby} = f.\text{Kl} \Rightarrow f.\text{Level} = 33$$

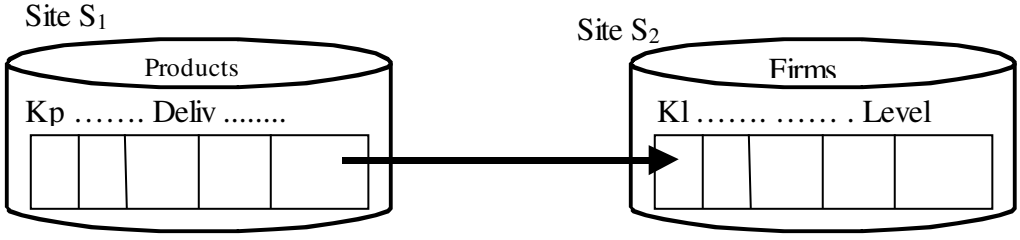


Figure 9: a sample distributed database

Each table has one million of tuples randomly generated. The simplicity of the database design does not simplify the checking algorithm. Moreover, as shown by Grefen and Widom [5], global constraints involving more than two sites are very rare and almost every constraint involving more than two sites can be rewritten as a conjunction of constraints involving only two sites.

3.3. Prototype global design

As shown on Figure 10, our prototype is composed of three layers: a global manager, a local controller and a local DBMS. The global manager is used to control the whole nested transaction on the site it is initiated (here Site1). Communications between sites are made between local controllers.

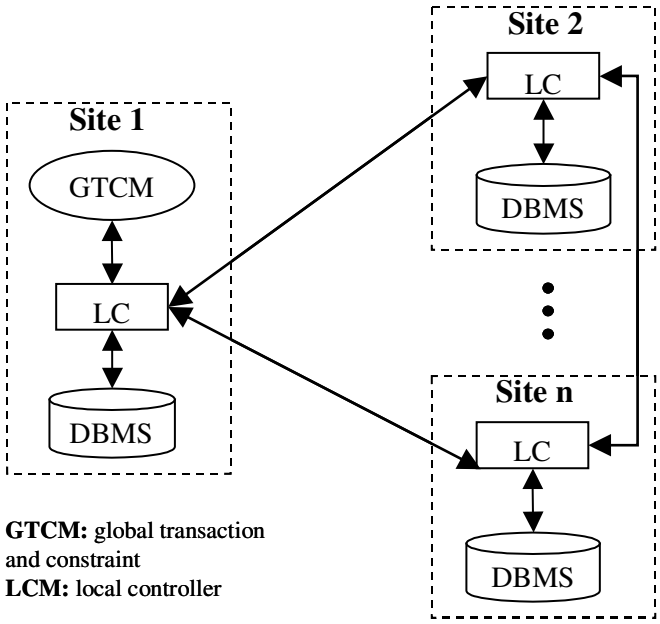


Figure 10.System Architecture

The global manager is in charge of controlling the execution of the nested transaction *SCA(C)* and of the processes that check constraint C. Its main functions are: initiate the prototype, coordinate the execution of operations according to the sites touched by the transaction, define and launch the processes required for checking the constraint and decide whether the transaction commits or abort. It establishes a connection with the local controller on the same site in order to schedule the different processes.

The local controller schedules and sends operations to the local DBMS: DML operations, operations that create the intersite sets and operations that check the locally computable predicates. It also sends data (intersite sets) to remote local controllers.

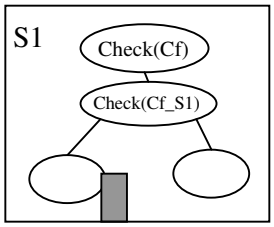
The local DBMS stores and manages local data. It takes advantage of the features of Oracle9i [7].

3.4. Experiments description

We ran four different experiment types, according to the different ways of distributing the data (one or two nodes), the distribution of updates over those nodes, the nested transaction type and the checking process. In the following, we describe those four experiment types. Grey rectangles represent updated tables; the dashed arrows represent data transfers.

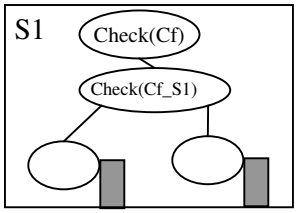
Experiment 1: updating one table on one site

This experiment is considered as a reference, since is it fully centralized, i.e. there is no data transfer from one node to another. It allows comparing the behavior of our mechanism in a distributed environment and in a centralized environment with similar conditions. There is one leaf subtransaction that updates table *Products*.



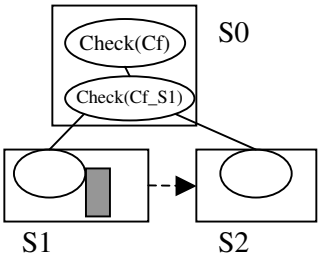
Experiment 2: updating two tables on one site.

This Experiment type is similar with the preceding one, but has two leaf subtransactions, one updating table *Products* and the other updating table *Firms*. Both tables are located on node S_1. It is also a reference scenario since it is fully centralized.



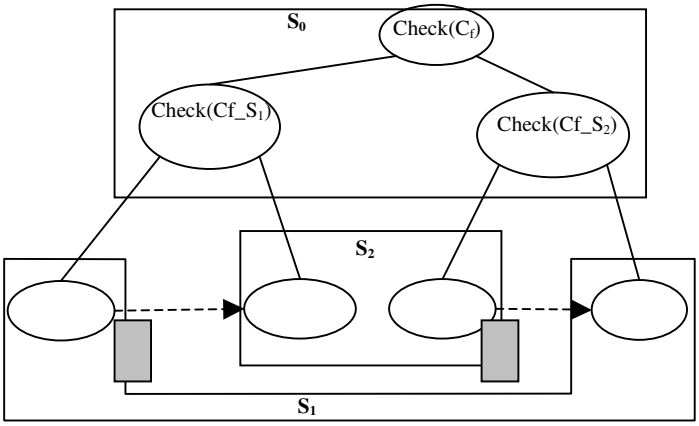
Experiment 3: updating one table on two sites from a third site

This scenario uses three cluster nodes. The nested transaction SCA(C_f) is initiated on node S_0. It has one leaf subtransaction sent to S_1 to update table *Products.* The three nodes are required for checking the constraint C_f. The execution control of SCA(C_f) and of the checking process is made on S_0, by performing the following tasks : initiate leaf subtransactions on the corresponding site (S1), coordinate the data transfer from S1 to S2, check the constraint and, in case of satisfaction, initiate the 2PC mechanism to validate the nested transaction.



Experiment 4: updating two tables on two sites from a third site

This experiment is similar to the preceding one but has two leaf subtransactions, one updating table *Products* and the other updating table *Firms*.



3.5. Experiment size

In order to observe the behavior of our mechanism with different workloads, each experiment is executed with six different sets (of increasing sizes) of tuples to update. More precisely, for each experiment, the UPDATE operation has been executed over 20.000, 50.000, 100.000, 200.000, 500.000 and 1.000.000 tuples. Each experiment is repeated ten times, in order to increase the reliability of our results. As mentioned above, the updates operation are chosen so that the constraint is never violated, which is the worst case in terms of response time. Furthermore, if the constraint could be violated, then the interpretation of the results would have been more difficult, since the checking time would depend on the time when the first tuple which violate the constraint is encountered.

4. Experiment results

Figure 11 compares the total time, defined in Section 3, obtained for experiments 1 and 3, where only one table is updated. It shows that distributing the checking process does not create overhead, even when updates are all performed on the same site and thus the checking does not take much advantage from the parallelization.

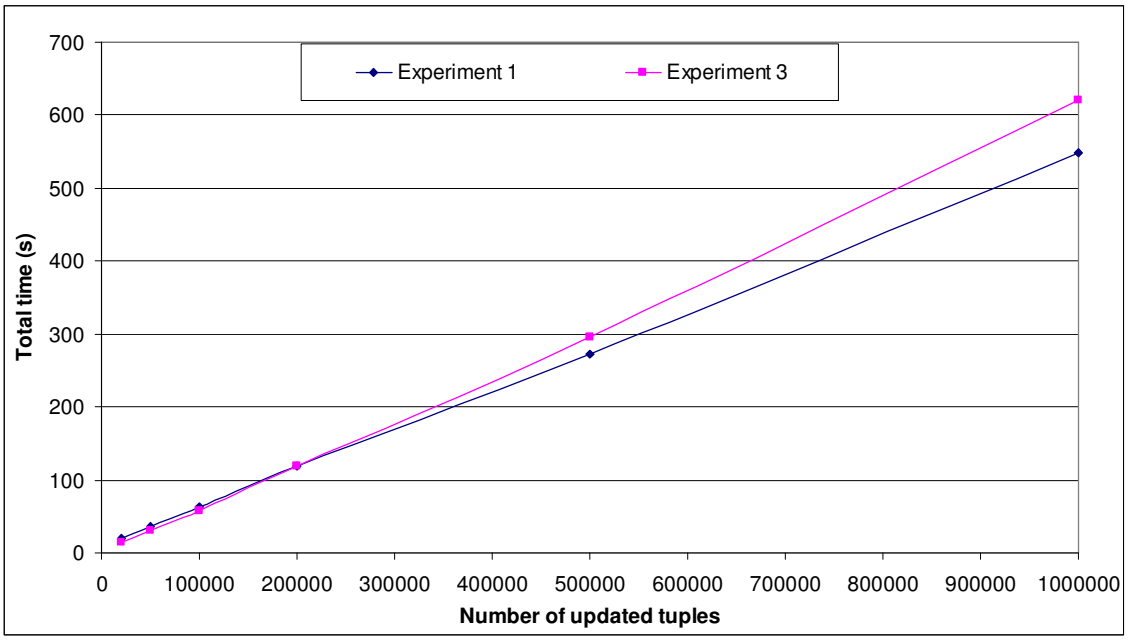


Figure 11. Experiment results when only one table is updated

Figure 12 compares the results obtained by experiments 2 and 4, when the nested transaction performs updates on both tables. We observe that experiment 4 raises better results than experiment 2, despite this latter does not require data transfer. This is mainly due to a better parallelism, since two nodes are used, instead of one, to perform the same tasks. Those results must be considered carefully, since they have been obtained on a rather fast network, which reduces the impact of data transfers on the overall execution time.

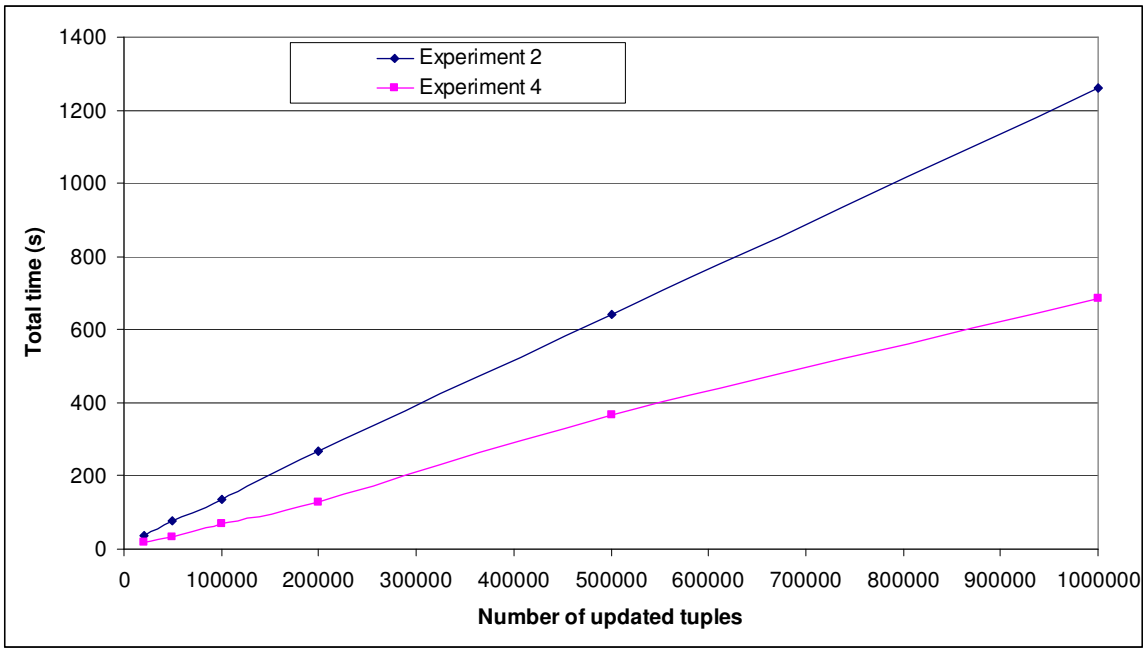


Figure 12. Experiments results when both tables Products and Firms are updated

5. Conclusions

In this paper, we present an efficient solution for maintaining global constraints in a relational multidatabase. This solution has been introduced in [2], so this paper focuses on implementing the solution and a first performance evaluation. We study two constraint types, referential integrity constraints and global conjunctive constraints. We analyze the operations likely to violate a constraint of such type and propose a mechanism that maintain integrity for each constraint type.

This mechanism has been implemented over a database cluster running Oracle9*i* on each node. Experiments have been led for global conjunctive constraints. The results we obtained show the feasibility of our approach, i.e. it is possible to efficiently maintain integrity in a multidatabase accessed through distributed nested transactions, by using a mechanism that uses itself distributed nested transactions. In other words, the benefits of a distributed nested transactions mechanism, execution control and parallelism, can be extended to the integrity maintenance issue.

To go further, we plan to experiment with more complex cases, where several nested transactions are executed in parallel and where several constraints can be violated by the same transaction. We also plan to experiment the mechanism for referential integrity checking presented in this paper, to see if it yields the same conclusions.

References

[1] J. Akoka and I.Comyn-Wattiau. *Conceptions des Bases de Données Relationnelles* Vuibert. Paris. 20001.

[2] A. Doucet, S. Gançarski, C. León and M. Rukoz. *Integrity Constraints in Multi-DB with Nested Transactions,* LNCS. Volume 2172. Proceedings 9th International Conference of Cooperative Information Systems, COOPIS'2001, Italia, 2001, Springer-Verlag.

[3] S. Grufman, F. Samson, S. M. Embury, P. M. D. Gray and T. Risch. *Distributing Semantic Constraints Between Heterogeneous Databases.* Proceedings of the Thirteenth International Conference on Data Engineering, ICDE 1.997, April 7-11, pages 33-42, Birmingham U.K., April

1-997. IEEE Computer Society.

[4] A. Gupta and J. Widom. *Local Verification of Global Integrity Constraints in Distributed Databases.* Proc. of the 1.993 ACM SIGMOD Int. Conf. on Management of Data, volume 22 of ACM SIGMOD Record, pages 49-58, Washington, USA, May 1.993. ACM Press.

[5] P.W.P.J. Grefen and J. Widom. *Protocols for Integrity Constraints Checking in Federated Database*. Distributed and Parallel Database, 5(4):327-355, 1997.

[6] J.E.B. Moss. *Nested Transactions: An Approach To Reliable Distributed Computing*. MIT Press, Cambridge, USA, 1.985.

[7] Oracle. *Oracle9i Database Administrator's Guide Release 2 (9.2).* Part No. A96521-01. Oracle Corporation. March, 2002.

[8] A. P. Sheth and J. A. Larson. *Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases.* ACM Computing Surveys, 22(3):183-236, September 1.990.