An Efficient Constraint Planning Algorithm for Multidatabases

Praveen Madiraju and Rajshekhar Sunderraman
Department of Computer Science
Georgia State University
Atlanta GA 30302
{cscpnmx,raj}@cs.gsu.edu

Abstract

We have earlier proposed a general framework of an agent based constraint checker for checking global integrity constraints. The constraint checker consists of: update parser, metadatabase extractor, constraint planner, constraint optimizer and constraint executor. One of the important steps in checking for global constraints is to generate sub constraints from global integrity constraints. These sub constraints can be locally executed on remote sites. The constraint planner devises an effective plan for generating such sub constraints from each of the global constraints. In this paper, we focus on the constraint planner module. We propose an efficient algorithm that takes as input an update statement and the list of all global constraints and outputs sub constraints to be executed on remote sites. Our algorithm is efficient since the algorithm does not require the update statement to be executed before the constraint check is carried out. In this way, we save lot of time and resources avoiding any potential rollbacks. Our vision for implementation exploits parallelism by virtue of employing mobile agents.

Keywords: Multidatabases, Integrity Constraint Checking, Mobile Agents, Constraint Planning

1. Introduction

A multidatabase system consists of autonomous component database systems. Data is distributed among multiple sites. The reasons for data distribution may be inherent nature of the data, performance reasons or individual sites being incapable of hosting large amounts of data (mobile environment). Data distribution is quite natural in the healthcare database system considered in this paper. Patient information is stored at site S_1. Insurance company stores patient's claim information at site S_2 and a different agency stores doctor's information at site S_3. It is difficult to enforce a centralised scheme as we have different agencies operating at their own rules. In some cases, where large volumes of data with millions of records are stored, it is just not possible to have centralised data due to performance factors. Data at these individual sites are not necessarily independent, but may participate in a relationship with data from other sites. An update statement issued on a single site might cause a global constraint to be violated, essentially endangering the consistency of the database. The consistency of the database is preserved by imposing global integrity constraints on such interrelated data.

We have earlier proposed a general framework for checking global integrity constraints using mobile agents [9]. The system architecture is shown in Figure 1. A *constraint checker* module resides on each of the data sources. This module is responsible for interfacing with the global metadatabase. In Figure 1, say, an update statement U_1 is issued on site S_1. It modifies/updates some of the database objects. Constraint checker on S_1 sends out mobile agent on to the global metadatabase. The global metadatabase is a repository of site and domain information. Site information gives description of sites where data sources reside. Domain information gives metadata description of database objects of all data sources and global constraints, say $C_1 \ldots C_n$. The mobile agent at the global metadatabase is equipped with the knowledge of database objects being modified and also data processing code. The mobile agent computes the list of global constraints being affected by U_1, say $C_1 \ldots C_m$. The mobile agent returns this list to the constraint checker. Constraint checker takes as input one global constraint at a time, C_1. For each global constraint, sub constraints corresponding to remote sites are generated. Mobile agents $rmagent_2$, $rmagent_3$, $rmagent_4$ are spawned to individual sites S_2, S_3, S_4. Constraint checker gathers results from these mobile agents and makes a decision if a global constraint is violated. This process is repeated for all remaining constraints $C_2 \ldots C_m$.

We give a very brief overview of the *constraint checker*. The constraint checker has five major modules: update parser, metadatabase extractor, constraint planner, constraint optimiser and constraint executor.

- *Update parser*: parses an update statement input by the user and identifying the database objects involved in the update statement.
- *Metadatabase extractor*: extracts all the global constraints being affected by the update statement.

0-7803-8735-X/05/$20.00©2005 IEEE

- *Constraint planner*: devises an effective plan for generating sub constraints on remote sites.

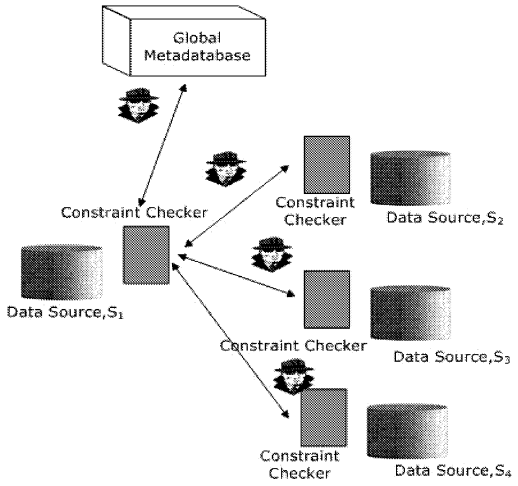

Figure 1: Constraint checking architecture

- *Constraint optimiser*: optimises sub constraints for efficient constraint checking.
- *Constraint executor*: generates and spawns mobile agents. The mobile agents execute the sub constraints and with the summarized information gathered from all the mobile agents, a decision is made if a global constraint is violated.

In the general framework, we proposed [9], we are using mobile agents for global constraint checking. To our knowledge, we have not seen any research work on using mobile agents for global constraint checking.

Our focus in this paper is the *constraint planner* module. *We propose an efficient algorithm to automatically decompose a global constraint into sub constraints to be executed on individual sites.* We limit our constraints to semantic integrity constraints.

The rest of the paper is organised as follows: In Section 2, we give an example healthcare multidatabase system that will be referred throughout the paper. We also give the basic notations and definitions for integrity constraints. The overall picture of the constraint planner module is explained in Section 3. In Section 4 we present the constraint planning algorithm. In Section 5 we compare our work with other peer's work and conclusions can be found in Section 6.

2. Preliminaries

We give an example healthcare multidatabase system. We also introduce the basic notations and definitions for integrity constraints.

2.1. Example database

Consider a typical health care multidatabase management system [9]. It is a very natural scenario to have patient's information distributed across multiple sites. In such a database setting, it is possible to have same predicate (table) names at two different sites. Hence, we need a notation that distinguishes one predicate from the other. We use the notation of: (S_i: table t), where t is the name of the table stored on site S_i.

At site S_1: Patient information is stored. A *PATIENT* relation with attributes *name* and type of *healthplan* is recorded. *S_1:PATIENT (name, healthplan).* A *PATIENTDETAILS* relation with attributes *name*, *address* where the patient lives, *employer* name and *salary* of the patient is recorded. *S_1:PATIENTDETAILS (name, address, employer, salary).*

At site S_2: Health insurance companies store patient's claim information. A *CLAIM* relation (table) with attributes *name* (patient name), *amount* of claim, date of claim and *type* of claim is recorded. *S_2:CLAIM (name, amount, claimdate, type).* A *CLAIMREVIEW* relation records patient's *name*, *date* of claim and *reviewer* name. *S_2:CLAIMREVIEW (name, claimdate, reviewer).*

At site S_3: Doctor's office maintains patient's name, doctor treating the patient and disease for which the patient is being diagnosed. A DOCTOR relation with attributes *name* (patient name), *doctorname* and *disease* is recorded. *S_3: DOCTOR (name, doctorname, disease).*

2.2. Constraints

In order to represent integrity constraints in the context of a database as query evaluation in the database, we consider integrity constraints in the form of range-restricted denials (datalog style notation).

$$\leftarrow L_1 \wedge L_2 \wedge \ldots \wedge L_n$$

Where each L_i is a literal or an aggregate literal involving a base predicate and global variables are assumed to be universally quantified over the whole formula [2].

Say integrity constraint C_0 states, the sum of all claim amounts of a patient with healthplan 'C' may not be more than 200000.

```
←S1:PATIENT(name,'C'),CLAIM(name,-,-,-),
 Sum(amount,S2:CLAIM(name,amount,-,-),s),
 s>200000.
```

This can be conveniently represented using the approach of [6]. A constraint is a query whose result is either 0 or 1([6] calls it "panic"). If the query produces 0 on the multidatabase D, then D is said to satisfy the constraint, or the constraint is violated on D.

```
PanicC0:-
 S1:PATIENT(name,'C'),CLAIM(name,-,-,-),
 Sum(amount,S2:CLAIM(name,amount,-,-),s),
 s> 200000.
```

For convenience, we will refer *PanicC0* as just C_0.

3. Constraint planner

The *constraint planner* module is responsible for generating a set of sub constraints from each of the global constraints. Figure 2 shows the overall procedure of generating sub constraints from global constraints.

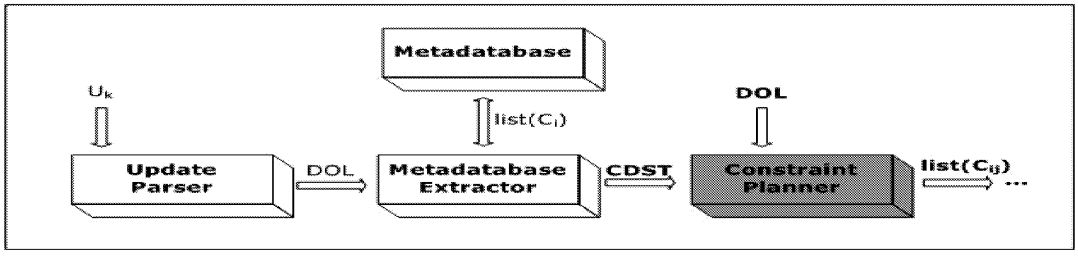

Figure 2: Overall procedure of generating sub constraints

list (C_i) is the list of global constraints being effected by a single update U_k .*DOL* is the database object list. The *DOL* contains the database objects being modified by U_k. *DOL* = {table *t* list *(colname, value)}*, where *t* is the table name being updated and *value* is the updated values of the corresponding column names (*colname*). *CDST* is the Constraint Data Source Table. The *CDST* gives the list of constraints *(list* (C_i)) being affected by U_k and also the list of sites *(list* (S_j)) affected by each C_i. *list*(C_{ij}) is the list of sub constraints corresponding to each C_i and site S_j . The constraint planner takes as input *CDST, DOL* and produces the output *list* (C_{ij}).

We will illustrate the overall procedure of generating the sub constraints using an example. Consider the initial multidatabase state as shown in Figure 3

S_1:PATIENT

name	healthplan
john	B

S_2:CLAIM

name	amount	claimdate	type

S_3:DOCTOR

name	doctorname	disease
john	mike	smallpox

Figure 3: Initial multidatabase state

Let

```
U1=insert into S2:CLAIM values
   ('john',25000,
   to_date('06/10/2003','MM/DD/YYYY'),
   'emergency');
```

The update parser parses the given the update statement and identifies the database objects being modified. The output from this step is the *database object list (DOL)*

```
DOL =
{S2:CLAIM (name='john',amount=25000,
claimdate='06/10/2003',type='emergency'}
```

The metadatabase extractor takes as input database object list. It contacts the metadatabase and gets the list of constraints being affected by the update statement and also the list of sites involved for each such constraint. The metadatabase extractor constructs the Constraint Data Source Table (CDST) as shown in Figure 4.

CDST (C_i) = <C_i, *list* (S_j)> where, C_i is the constraint identifier and *list*(S_j) is the list of data sources being affected by C_i

CDST

C_i	list(S_j)
C_5	(S_1,S_2,S_3)
C_6	(S_1,S_2)

Figure 4: The constraint data source table

```
C5:-
   S1:PATIENT(name,'B'),
   S2:CLAIM(name,amount,_,_),
```

```
  S3:DOCTOR(name,_,'smallpox'),
  amount > 20000.

C6:-
  S1:PATIENT(name,healthplan),
  S2:CLAIM(name,  _,  _,  'emergency'),
  healthplan = 'B'.
```

Constraint C_5 states that a patient with healthplan 'B' diagnosed with 'smallpox' may not claim more than 20000 dollars. Constraint C_6 states that a patient with healthplan 'B' may not file a claim of type 'emergency'. The *constraint planner* takes as input the *DOL* and *CDST*. It outputs the list of sub constraints *(list (C_{ij}))* for each global constraint. The value of each C_{ij} is either 0 or 1. The constraint planning algorithm given in the next section decomposes a global constraint C_i in to a set of sub constraints C_{ij} to be executed locally on remote sites.

For the running example, for C_5, the corresponding sub constraints generated are: C_{51}, C_{52}, C_{53} and for C_6, the sub constraints generated are: C_{61}, C_{62}. The values of these sub constraints are given in the next section (*Example 4.1*).

The "**...**" in Figure 2 represents that the list of sub constraints are fed to the next stage of constraint checking. The list (C_{ij}) is sent to the *constraint optimiser* and any optimisations that improve the constraint checking process are carried out. Finally the sub constraints are executed at remote sites by *constraint executor* module. In the above example, $C_6 = C_{61} \wedge C_{62}$ and $C_5 = C_{51} \wedge C_{52} \wedge C_{53}$. If the value of C_5 or C_6 is 1, it implies a constraint has been violated.

4. Constraint Planning Algorithm (CPA)

The basic idea of constraint planning is to decompose a global constraint in to a conjunction of sub constraints, where each conjunct represents the constraint check as seen from each individual database [5]. Given an update statement, a brute force approach would be to go ahead and update the database state from D to D' and then check for constraint violation. However, we want to be able to check for constraint violation with out updating the database. Hence, the update statement is carried out only if it is a non constraint violator.

The approach of the constraint planning algorithm (*CPA*) is to scan through the global constraint C_i, update statement U and then generate the conjunction of sub constraints, C_{ij}'s[+] .The value of each conjunct (C_{ij}) is either 0 or 1 and if the overall value of the conjunction is 1, constraint is violated, otherwise not. An update U can be an update involving an insert or a delete or a modify statement. Hence we have three different cases for the algorithm. They are given in Sections 4.1, 4.2 and 4.3.

4.1. CPA-insert

Algorithm `CPA-insert(`constraint planning algorithm for an insert statement) gives constraint decompositions (C_{ij}'s), corresponding to global constraint C_i and an update statement involving an insert statement. Algorithm CPA-insert takes as input the update statement U and the list of all global constraints C and outputs the list of sub constraints (C_{ij}) for each C_i being affected by U.

DOL, database object list, identifies the database objects being modified by the update statement, U. *DOL* (*line 3*) identifies, the table R with attributes (column names) $a_1 \ldots a_n$ inserted with values $t_1 \ldots t_n$. *CDST* (*line 4*) gives the list of sites involved, for each constraint being affected by the update statement. The outer for loop variable *i* (*line 6*) loops through all the constraints $C_1 \ldots C_q$ affected by the update U. The inner for loop variable *j* (*line 7*) loops through each site ($<S_{11},\ldots,S_{1n_1}>,\ldots,<S_{q1},\ldots,S_{qn_q}>$) for each constraint *i*. Inside the for loop (*lines 6-25*), all the sub constraints C_{ij}'s are generated. S_j:$p_1(X_1), p_2(X_2), \ldots, p_r(X_r)$ (*line 8*) denotes, for a particular site S_j, $X_1 \ldots X_r$ are the vector of variables corresponding to the predicates (table names), $p_1 \ldots p_r$.

A critical feature of the algorithm is the generation of intermediate predicate *(IP)*. *IP*'s are generated only at the site where update is occurring. In concept, *IP*'s represent information that needs to be shared from a different site. Implementation wise, *IP* is a SQL query returning value of the variable, v (*line 14*) from a different site. IP_{ikd} (*line 16*) means the *d*th intermediate predicate corresponding to constraint C_i and site S_K. The table dual (*line 10*) is like a "dummy" table provided by the ORACLE. It is a convenience table which has exactly one column and only one row and similarly, other database vendors provide different tables for this purpose.

Algorithm CPA–insert
1: **INPUT**
(a) U: insert S_m:R($t_1,\ldots,t_n$)
(b) C: list of all global constraints
/* Note: insert is occurring on site S_m */
2: **OUTPUT**

[+] Recall that C_{ij} indicates the sub constraint corresponding to a global constraint C_i on site S_j

list of sub constraints < C_{i_1},…,C_{ik_i}> for each C_i affected by U
3: DOL (U) = < R (a_1= t_1,…,a_n= t_n) >
4: CDST(C,DOL(U)) = < <C_1, (S_{11},…,S_{1n_1})>,… ,<C_q, (S_{q1},…,S_{qn_q})> >
5: *let* θ = {x_1 ← t_1,…,x_n ← t_n}be obtained from DOL(U) where x_1…x_n are variables corresponding to the columns of table R
6: **for** each *i* in {*1… q*} **do**
7: **for** each *j* in {*1…n_i*} **do**
8: *let* S_j:p_1 (X_1) ,p_2 (X_2),…,p_r (X_r) be the sub goals of C_i associated with S_j and A be all arithmetic sub goals associated with S_j
9: **if** (j <> m) **then** /* site where update is not occurring */
10: C_{ij} = select 1 from dual where exists (select * from p_1 …p_r where <*cond1*>)
11: <*cond1*> is obtained from X_1…X_r using standard method of joining tables. It also includes any arithmetic sub goal conditions
12: **else if** (j=m) **then** /* site where update is occurring */
13: **if** (there exists variables in A that do not appear among X_1…X_r) **then**
14: **for** each variable v in A that do not appear among X_1…X_r **do**
15: *let* k be the site where v appears in a sub goal, S:t(X) in C_i
16: IP_{ikd} = (select *Col(v)* from S:t where <*cond2*>)
17: *Col (v)* is the column name corresponding to v
18: <*cond2*> is obtained from X_1…X_r and X. d is nth *intermediate predicate*
19: **end for**
20: **end if**
21: C_{ij} = return 1 if (<*cond3*> and *A′*) else return 0.
22: <*cond3*> is obtained from θ and X_1…X_r and *A′* is A with IP's replacing corresponding variables
23: **end if**
24: **end for**
25: **end for**
26: apply the substitution θ(U) to all C_{ij}

Theorem 4.1: *The conjunction of sub constraints C_{ij}'s, generated from Algorithm CPA-insert conclusively determines, if an update statement involving an insert statement violates a global constraint C_i.*

Proof:

Consider an update statement on site S_m, global constraint C_i and the list of sub constraints, C_{ij}'s generated from algorithm `CPA-insert`. The generation of each C_{ij} needs to achieve the same affect as sub goal corresponding to S_j. Let S_j:$P_1(X_1)$,$P_2(X_2)$,…,$P_r(X_r)$ be the sub goals of C_i associated with S_j and A be all arithmetic sub goals associated with S_j. At this point each C_{ij} falls in one of the two cases. We will show that each C_{ij} in both the cases achieves the same affect as the sub goal corresponding to site S_j.

Case I (j<>m): This is the case where sub goal is associated with a site other than where update is occurring (*lines 9-11*). The generation of C_{ij} in this case is rather straight forward as it generates a sub constraint check from all the predicates involved on site *j* using appropriate join conditions and it also includes any arithmetic sub goal conditions. Hence C_{ij} naturally achieves the exact same result as the sub goal corresponding to site S_j.

Case II (j=m): This is the case where sub goal is associated with a site where update is occurring (*lines 12-23*). The generation of Cij's in this case consists of two parts. Part 1 consists of information from the same site – trivial case (just as Case I). Part 2 relates to information acquired from a remote site. For each such variable a unique *intermediate predicate* is generated. IP's are SQL queries returning values of such variable by computing appropriate joins and arithmetic conditions involved with such variables. Hence, IP guarantees correct exchange of information from a different site. The reason we are generating unique IP's is we can either store all the IP's at a global directory such as the metadatabase or we can generate IP's at run time.

Hence, from both the cases, we observe that the conjunction of C_{ij}'s entails the original global constraint, C_i. Therefore, if C_i determines whether an update involving an insert statement violates a global constraint C_i, then the conjunction of its sub constraints C_{ij}'s also determines if the constraint C_i is violated. In other words, if the conjunction of C_{ij}'s evaluates to 0 (false), constraint C_i is not violated, otherwise C_i is violated (*concepts from Section 2.2*). ■

We show the working of the Algorithm `CPA-insert` on some sample examples given below:

Example 4.1

This example considers constraint defined on the healthcare multidatabase system from Section 3. It showcases how sub constraints are generated in a simple case, when intermediate predicates are not involved.

Input:

```
U=insert into S2:CLAIM values('John',
25000, '06/10/2003', 'emergency')
C = list of all global constraints
```

Output:

```
List of sub constraints <Ci1 … Cik_i >
for each Ci affected by U
```

```
DOL(U) =
S2:CLAIM{name='john',amount=25000,claimda
te='06/10/2003',type='emergency'}

CDST(C,DOL(U)) = <C5,(S1,S2,S3)>

where

C5:-S1:PATIENT(name,'B'),
    S2:CLAIM(name,amount,_,_),
    S3:DOCTOR(name,_,'smallpox'),
    amount > 20000.
```

/* C_5 states that "A patient with health plan 'B' diagnosed with 'smallpox' may not claim more than 20,000 dollars". */

```
θ={S2:CLAIM(name1='john',amount1=25000,
  claimdate1='06/10/2003',
  type1='emergency') }

/* C51 is generated from algorithm CPA-
insert (lines 9-11)*/

C51 = select 1 from dual where exists
(select * from patient where name = name1
and healthplan = 'B')

/* C52 is generated from algorithm CPA-
insert (lines 12-23) */

C52 = return 1 if {name=name1 and amount1
> 20000} else return 0.

/* C53 is generated from algorithm CPA-
insert (lines 9-11) */

C53 = select 1 from dual where exists
(select * from DOCTOR where name=name1
and disease = 'smallpox')
```

Apply θ to each of the sub constraints

```
θ(C51) = select 1 from dual where exists
(select * from patient where name =
'john' and healthplan = 'B')

θ(C52)= return 1 if {'john' = 'john' and
25000 > 20000} else return 0

θ(C53)= select 1 from dual where exists
(select * from DOCTOR where name='john'
and disease = 'smallpox')
```

$C_5 = C_{51} \wedge C_{52} \wedge C_{53}$. In this example, $\theta(C_{51}) = 1$ (true), $\theta(C_{52}) = 1$ (true) and $\theta(C_{53}) = 1$ (true). The values of the sub constraints are calculated against the sample multidatabase state shown in Figure 3. The conjunction of C_{51}, C_{52} and C_{53} evaluates to true. Hence, C_5 is violated (*from Theorem 4.1*)

Similarly, for the example constraint C_6 from Section 3, we generate:

```
θ(C61) = select 1 from dual where exists
(select * from patient where name='john'
and healthplan = 'B')

θ(C62)=return 1 if {'john'='john' and
'emergency'='emergency'} else return 0
```

$C_6 = C_{61} \wedge C_{62}$. In this example, $\theta(C_{61}) = 1$(true), $\theta(C_{62}) = 1$(true). The conjunction of C_{61} and C_{62} evaluates to true. Hence, C_6 is also violated (*from Theorem 4.1*). Note that we do not need to evaluate other constraints, if one of the constraints is violated by an update statement. In this example, since C_5 is violated, we do not need to evaluate/check for C_6. We show the evaluation of C_6 simply for illustrative purposes.

Example 4.2

This example considers a constraint defined on a credit card, loan and a car database system. CREDITCARD table stores information of a person's name and his credit card balance, LOAN table stores information of a person's name and his loan balance and CAR table stores information of a person's name and his car balance. This example showcases generation of sub constraints when intermediate predicates are involved.

Input:

```
U = insert into S8:CAR values ('mark',
10000)

C = list of all global constraints
```

Output:

List of sub constraints $<C_{i_1} \ldots C_{ik_i}>$ for each C_i affected by U

```
DOL(U) = S8:CAR{name='mark',
         carbal=10000 }

CDST(C, DOL (U)) = <C7, (S6, S7, S8)>
```

Where,

```
C7:-
  S6:CREDITCARD(name, cdbal),
  S7:LOAN(name, loanbal),
  S8:CAR(name,carbal),
  (cdbal + loanbal + carbal  > 75000).
```

/* C_7 can be stated as: The sum of customer's credit card balance, loan balance and car balance should be less than 75000 */

```
θ={S8:CAR(name1='mark',carbal1 = 10000}

/* IP761 is obtained from algorithm CPA-
insert (line 16) */

IP761 = select cdbal from CREDITCARD
        where name = name1

IP771 = select loanbal from LOAN where
        name = name1
```

```
/* C78 is obtained from algorithm CPA-
insert (line 21) */

C78 = return 1 if {name = name1 and
(IP761 + IP771 + carbal1 > 75000)}
      else return 0.

apply θ to each of the sub constraints

θ(IP761) = select cdbal from CREDITCARD
           where name = 'mark'

θ(IP771)=select loanbal from LOAN where
         name = 'mark'

θ(C78) = 1 if {name = 'mark' and
  (θ(IP761) + θ(IP771) + 10000 > 75000)}
  else return 0
```

4.2. CPA-delete

Here, we make an important observation that an update statement involving a delete can only violate referential integrity constraints, semantic integrity constraints involving aggregate predicates (sum, max, min, avg and count), state transition and state sequence constraints involving aggregate predicates. It does not violate semantic integrity constraints involving arithmetic predicates considered in this paper.

4.3. CPA-modify

The constraint planning algorithm for a modify statement can be modelled as a delete followed by an insert statement.

4.4. Discussion

The CPA considers only elementary update statements. The elementary update statements are statements affecting only one row of a table at a time. However, note that any update statement can be translated equivalently to a set of elementary updates. Hence the generality of the CPA is not lost.

The CPA can generate sub constraints for constraints having universally quantified variables over a simple conjunction of predicates as discussed in Section 2.2. We are currently working to extend CPA for sub goals of the global constraint involving aggregate predicates (sum, max, min, avg and count). The extensions to the paper would be: a) modified Algorithm CPA-insert to deal with Aggregates and b) new Algorithm CPA-delete

We have not considered the issue of constraint checking in the presence of transactions. Let a transaction T change the current database state D to D'. A naïve approach would be to check for constraint violations in D' and if any constraints are violated, we rollback to the previous state D.

5. Related work

Grufman et al. [5] provide a formal description of distributing a constraint check over a number of databases. They propose that the problem of generating sub constraint from a global constraint is same as rewriting a predicate calculus expression of the constraint check in to a form in which the distribution of the data is respected. The rewritten predicate can be seen as a conjunction of sub constraints, where each sub constraint may be visualised as the constraint check as seen from each individual database. During the process of rewriting the constraint check predicate, they introduce the concept of intermediate predicate. The idea of intermediate predicates used in this paper has been borrowed from Grufman et al. [5]. In their constraint distribution model, an update statement is first carried out and the new database state is checked for constraint violation. If the constraint is violated, the update is rolled back. Our work differs from theirs by giving an algorithm that automatically decomposes a global constraint in to a conjunction of sub constraints. We also give *Theorem 4.2* as a proof of correctness to our algorithm. Our approach is much more sophisticated, as we check for constraint violation with out actually updating the database. The update is executed only when there are no constraint violations. Hence our algorithm is efficient as there are no problems involved with rollbacks as such.

Ibrahim [8] proposes a strategy for constraint checking in distributed database where data distribution is transparent to the application domain. They propose an algorithm for transforming a global constraint into a set of equivalent fragment constraints. However, our algorithm coverage is much broader as we can have different tables on different sites. In our approach, the constraint planning algorithm generates the sub constraints, which can be readily implementable on oracle database system. With minor changes, it can be implemented on any commercial database.

Much of the research concerning integrity constraint checking has been done in the area of relational database systems. Grefen and Apers [3] provide an excellent survey of constraint checking and enforcement methods in relational database systems. Grefen and Widom [4] give an exhaustive survey of protocols for integrity constraint checking in federated database systems. Gupta and Widom [7] give approaches for constraint checking in distributed databases at a single site. They show how a class of distributed constraints can be broken down into local update checks. Some of the approaches for distributed databases and federated databases can be easily applied to multidatabases with some minor

changes. Ceri and Widom [1] propose inter-database triggers for maintaining equality constraints between heterogeneous databases. Their approach relies on active rules and assumes a persistent queue facility between sites. Widom and Ceri [11] mention research on active databases and constraints.

6. Conclusions

In this paper, we have presented the constraint planning algorithm, which takes as input an update statement and the list of all global constraints and outputs the sub constraints to be executed on remote sites. Our main contributions in this paper are two fold:

1) An algorithmic description of the constraint planning module
2) An efficient constraint planning algorithm (CPA). The algorithm is efficient since the CPA does not require us to update the database and then check for constraint violations. Instead, the update is carried out only in the event of no constraint violations. Hence the multidatabase does not have to deal with any rollbacks as such.

Future Work

We plan to implement the CPA discussed in the paper. Our vision for implementation considers using mobile agents. The advantages of using agents for our approach are as follows: 1) For each sub constraint generated from CPA, a mobile agent would carry the data processing code and execute the sub constraint check on the remote site. Agents on the remote site process the data and only filtered data is transported to the base site. Thus, we save on the network bandwidth. 2) Constraint checking mechanism is much faster as the sub constraint checks on remote sites are executed in parallel by mobile agents.

We are making progress to extend the constraint panning algorithm for aggregate predicates (sum, max, min, avg and count). We would like to investigate further to extend the CPA to the class of state transition and temporal constraints. We also plan to give a performance cost model for the constraint optimiser module.

References

[1] Ceri, S. and Widom, J. Managing Semantic Heterogeneity with Production Rules and Persistent Queues. Proceedings of VLDB, pages 108-119, Dublin, Ireland, August 1993.

[2] Das, S.K. and Williams, M.H. Extending integrity maintenance capability in deductive databases. In the proceedings of the UK ALP-90 Conference (Bristol, England, January), Oxford, England: Intellect, pp.75-111, 1990.

[3] Grefen, P. and Apers, P. Integrity Control in Relational Database Systems - An Overview, Journal of Data and Knowledge Engineering, 10 (2), 187-223, 1993

[4] Grefen, P. and Widom, J. Protocols for integrity Constraint Checking in Federated Databases. International Journal of Distributed and Parallel Databases, 5(4): 327-355, October 1997

[5] Grufman, S., Samson, F., Embury, S.M., Gray, P.M.D and Risch T. Distributing Semantic Constraints Between Heterogeneous Databases. Proceedings of ICDE, April 7-11, pages 33-42, Birmingham U.K., April 1997

[6] Gupta, A., Sagiv, Y., Ullman, J.D. and Widom, J. Constraint Checking with Partial Information. Proceedings of the Thirteenth ACM PODS, pages 45-55, Minneapolis, Minnesota, May 1994.

[7] Gupta, A. and Widom, J. Local Verification of Global Integrity Constraints in Distributed Databases. Proceedings of ACM SIGMOD , pages 49-58, Washington, D.C., May 1993.

[8] Ibrahim, H. A Strategy for Semantic Integrity Checking in Distributed Databases. Proceedings of the ninth International Conference on Parallel and Distributed Systems, ICPADS 2002, pages 139-144.

[9] Madiraju, P. and Sunderraman, R. Mobile Agent Approach for Global Database Constraint Checking. Proceedings of ACM Symposium on Applied Computing (SAC'04), Nicosia, Cyprus, 2004, pp. 679-683.

[10] Türker, C., and Gertz, M. Semantic integrity support in SQL: 1999 and commercial (object) relational database management systems. VLDB Journal 10(4): 241-269 (2001)

[11] Widom, J. and Ceri, S. Active Database Systems: Triggers and Rules for Advanced Database Processing. Morgan Kaufmann, San Francisco, California, 1996.