# A Framework for Global Constraint Checking Involving Aggregates in Multidatabases Using Granular Computing

Praveen Madiraju, Rajshekhar Sunderraman and Haibin Wang, *Member, IEEE*

*Abstract* - **We have earlier introduced constraint checker, a general framework for checking global constraints using an agent based approach. In this paper, we complement the constraint checker with algorithms for checking global constraints involving aggregates in the presence of updates. The algorithms take as input an update statement, a list of global constraints involving aggregates, and granulizes each global constraint into sub constraint granules. The sub constraint granules are executed locally on remote sites and then the algorithm decides if a constraint is violated based on these sub constraint executions. The algorithms are efficient as the global constraint checks are carried before the update; hence we save time and resources spent on rollbacks.**

***Index Terms*— Global Integrity Constraints, Multidatabases, Aggregate Constraints**

## I. Introduction

Aggregate queries and their optimisations have long been recognised as an important area in advanced database applications, such as data warehousing and decision support systems [5,15].Naturally, these kinds of applications enormously utilise constraints involving aggregates. Hence, we need to check for such aggregate constraint violations under updates. Granular computing (GrC) [10,11,17] has received much attention during recent years. The different issues and perspectives of granular are well explained in [16]. One of the basic ideas of GrC is to decompose a computing problem into sub granules. These sub granules are either aggregated or decomposed further into new sub granules and this process repeats until we find the solution to the computing problem. We apply the decomposition idea of GrC to check for global constraint violations in multidatabases.

Most of the commercial database systems and previous research has considered checking for constraint violations after executing an update statement. However, this leads to extra time and resources being spent on rollbacks, when the constraints are violated. This situation is further exacerbated in a multidatabase setting, when an update statement causes global constraints to be violated. Therefore, we design an agent based general framework, propose algorithms, and implement prototype of the system for checking global constraints *without* having to execute the update statement. This saves time and resources spent on rollbacks.

Praveen Madiraju (praveen@mscs.mu.edu) is an Assistant Professor in the Department of Mathematics, Statistics and Computer Science at Marquette University, Milwaukee, USA. Rajshekhar Sunderraman (raj@cs.gsu.edu) is an Associate Professor in the Department of Computer Science at Georgia State University, Atlanta, USA. Haibin Wang is a research scientist in Winship cancer institute at Emory University, Atlanta, USA.

We have earlier proposed a general framework for checking global semantic integrity constraints using mobile agents [12]. To our knowledge, we have not come across of any research using mobile agents for checking global semantic integrity constraints. These constraints are mainly classified as constraints involving arithmetic and aggregate functions. In [13], we have proposed algorithms for checking constraints involving arithmetic predicates. Here, we extend our on-going work by proposing algorithms for checking constraint violations involving aggregates. Due to space limitations, we are not able to describe the implementation details.

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 basic notations for integrity constraints. The aggregate constraint checking algorithms are discussed in Section 3. We compare our work with other peer's work and offer our conclusions in Section 4.

## II. PRELIMINARIES

We give an example healthcare multidatabase system. We also introduce the basic notations for integrity constraint representation.

### *A. Example Database*

Consider our example of a health care multidatabase as shown in Figure 1. 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: `Si:table t`, where t is the name of the table stored on site $S_i$. To make the problem interesting and generic, we consider both vertical and horizontal distribution of data. *CLAIM* table is horizontally distributed across all the three sites, $S_1$, $S_2$ and $S_3$. A patient can make multiple claims uniquely identified by their *CaseId*. For example, John is associated with multiple claims (with CaseId's - 1, 3, and 4) on sites $S_1$ and $S_3$. We avoid the description of the tables and columns as they are self explanatory from their names.

### *B. Constraints*

We consider integrity constraints in the form of range-restricted denials (datalog style notation).

$\leftarrow A_1 \wedge A_2 \wedge \ldots \wedge A_n$

Where each $A_i$ is a literal or an aggregate literal involving a base predicate and global variables are assumed to be universally quantified over the whole formula [1]. An aggregate literal is expressed as

$$A_i(\hat{s},\ \alpha(y):v) :\text{-}\ B$$

**S₁:CASE**

| CaseId | SSN | InjuryDate |
|---|---|---|
| 1 | 123 | 06/16/2004 |
| 2 | 234 | 06/24/2004 |
| 3 | 123 | 10/12/2004 |
| 4 | 123 | 01/09/2005 |
| 5 | 123 | 02/09/2005 |

**S₁:PATIENT**

| SSN | Pname | HealthPlan |
|---|---|---|
| 123 | John | B |
| 234 | Mike | C |
| 345 | King | A |
| 456 | Mark | B |

**S₁:CLAIM**

| CaseId | ClaimDate | Amount | Type |
|---|---|---|---|
| 1 | 07/10/2004 | 10,000 | Inpatient |
| 3 | 11/14/2004 | 40,000 | Emergency |

**S₂:CLAIM**

| CaseId | ClaimDate | Amount | Type |
|---|---|---|---|
| | | | |

**S₃:TREATMENT**

| CaseId | Dname | Tdate | Disease |
|---|---|---|---|
| 1 | Clark | 06/18/2004 | Leg Injury |
| 3 | Blake | 10/15/2004 | SmallPox |
| 4 | Clark | 01/16/2005 | Flu |
| 5 | Harry | 02/10/2005 | Allergy |

**S₃:CLAIM**

| CaseId | ClaimDate | Amount | Type |
|---|---|---|---|
| 4 | 02/11/2005 | 30,000 | Prescription |
| | | | |

**Figure 1: Example healthcare multidatabase**

Where (i) *B* is a conjunction of predicate atoms that represent relations, (ii) $\hat{s}$ is the grouping list of attributes that must appear some where in the body of the rule - B, (iii) α is aggregate function such as *avg, count, max, and min*, (iv) *y* is the aggregate variable, and (v) *v* is the result of applying the aggregate function. We assume that the aggregate literals are not recursive, just as in [14].

Say integrity constraint $C_1$ states "the sum of claim amounts for each patient with healthplan 'B' may not be more than 100000". 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.

```
A(SSN,SUM(Amount):v1)  :- S1:PATIENT(SSN,-,'B'),
                          S1:CASE(CaseId,SSN,-),
                          S1:CLAIM(CaseId,-
                          ,Amount,-).
B(SSN,SUM(Amount):v2):-   S1:PATIENT(SSN,-,'B'),
                          S1:CASE(CaseId,SSN,-),
                          S2:CLAIM(CaseId,-
                          ,Amount,-).
C(SSN,SUM(Amount):v3):-   S1:PATIENT(SSN,-,'B'),
                          S1:CASE(CaseId,SSN,-),
                          S3:CLAIM(CaseId,-
                          ,Amount,-).
PanicC1 ← A(SSN,v1),B(SSN,v2),C(SSN,v3),
          v1+v2+v3 >10000.
```

For convenience, we will refer $PanicC_1$ as just $C_1$.

## III. CONSTRAINT PLANNING INVOLVING AGGREGATES

The basic idea of constraint planning is to decompose a global constraint into a conjunction of sub constraints (or granules), where each conjunct represents the constraint check as seen from each individual database [4]. 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 without 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 involving aggregates is to scan through the global constraint $C_i$(involving aggregates), update statement U and then generate the conjunction of sub constraints, $C_{ij}$'s ($C_{ij}$ indicates the sub constraint corresponding to constraint $c_i$ on site $s_j$). 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.

### *A. CPAggreg-insert*

Algorithm `CPAggreg-insert` (constraint planning involving aggregates for an insert statement) gives constraint decompositions ($C_{ij}$'s), corresponding to global constraint $C_i$ (involving aggregates) and an insert statement (decomposition is based on the locality of sites). Algorithm `CPAggreg-insert` takes as input the insert 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 $t_1 \ldots t_n$. The constraint data source table, *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 6-40*), 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), $p_1 \ldots p_r$.

**Algorithm** `CPAggreg-insert`

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

2: **OUTPUT**: list of sub constraints $< C_{i_1},\ldots,C_{ik_i} >$ for each $C_i$ affected by U

3: DOL (U) = < R ($a_1$= $t_1$,…,$a_n$= $t_n$) >

4: CDST(C,DOL(U)) = <<$C_1$, ($S_{11},\ldots,S_{1n_1}$)>,…,<$C_q$, ($S_{q1},\ldots,S_{qn_q}$)>>

5: *let* θ = {$x_1 \leftarrow t_1,\ldots,x_n \leftarrow t_n$}be obtained from DOL(U) where $x_1 \ldots x_n$ 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* A be all arithmetic sub goals associated with $S_j$, *Aggreg* be all Aggregate literals associated with site $S_j$ (atleast one of the predicates in the body of aggregate literal belongs to $S_j$) and $S_j$: $p_1(X_1)$, $p_2(X_2)$… $p_r(X_r)$ be sub goals of $C_i$ associated with $S_j$

9: **if** (j <> m) **then**
/* site where update is not occurring */

10: **for** each Aggregate literal, aggreg($\hat{s}$,$\alpha$(y):v):- B **do**
11: $A_{ijd}$ = select $\hat{s}$,$\alpha$(y)
from predicates in the
Body B
where <cond1>
group by $\hat{s}$
12: if all the predicates in B belong to same site $S_j$, <cond1> is obtained by standard joining of tables from B using variables from θ; else semi-join operation is employed for distributed tables. It includes any arithmetic sub goal conditions. $A_{ijd}$ is the value of the aggregate literal corresponding to constraint $C_i$, site $S_j$ and d is the nth such literal. $V_{ijd}$ is the value of aggregate operation corresponding to $A_{ijd}$
13: **end for**
14: **else if** (j=m) **then** /* site where update is occurring */
15: **for** each Aggregate literal, aggreg($\hat{s}$,$\alpha$(y):v):- B **do**
16: $A_{ijd}$ = select $\hat{s}$,$\alpha$(y)
from predicates in the
Body B where <cond2>
group by $\hat{s}$
17: **if** α = "sum" **then**
18: $v_{ijd}$ = θ(y)+ $v_{ijd}$
/*$v_{ijd}$ is the value calculated from $A_{ijd}$ of line 16 */
19: **else if** α = "min" **then**
20: $v_{ijd}$ = min(θ(y),$v_{ijd}$)
21: **else if** α = "max" **then**
22: $v_{ijd}$ = max(θ(y),$v_{ijd}$)
23: **else if** α = "count" **then**
24: **if** θ(y) is not null **then** $v_{ijd}$ = $v_{ijd}$ + 1 /* we are assuming single row inserts */
25: **else if** α = "avg" **then**
26: add θ(y)to the sum aggregate and divide by total count
27: **end if**
28: **end for**
29: **if** (there exists variables in A that do not appear in Aggreg or θ ) **then**
30: **for** each variable var in A that do not appear in Aggreg or θ **do**
31: *let* k be the site where var appears in a sub goal, S:t(X) in $C_i$
32: $IP_{ikd}$ = (select Col(var) from S:t where <cond3> )
33: Col(var) is the column name corresponding to var . *<cond3>* is obtained from joining X and θ . d is nth *intermediate predicate*
34: **end for**
35: **end if**
36: $C_{ij}$ = return 1 if (*<cond4>* and (*logical and*) *A´*) else return 0.
37: *<cond4>* is obtained from θ and $X_1 \ldots X_r$. *A´*is A with IP's replacing corresponding variables and $v_{ijd}$'s replacing corresponding aggregate values
38: **end if** /* end of the "else if" on line 12 */
39: **end for**
40: **end for**
41: apply the substitution θ(U) to all $C_{ij}$

A critical feature of the algorithm is the generation of $v_{ijd}$'s (lines 15-28) at the site where update is happening. Also, an *intermediate predicate* (IP) is 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, var (*line 30*) from a different site. $IP_{ikd}$ (*line 32*) means the *d*th intermediate predicate corresponding to constraint $C_i$ and site $S_K$.

**Example 3.1 :** Here, we show the working of the algorithm CPAggreg-insert on the example database and constraints introduced in Section 2. Consider the initial multidatabase state as shown in Figure 1

*Input*: $U_1$ = insert into $S_2$:CLAIM values (5,'02/20/2005',25000,'Emergency');
C = list of all global constraints
*Output*: list of sub constraints $Ci_1$ ,…,$Cik_i$ for each $C_i$ affected by $U_1$
DOL = {$S_2$:CLAIM (CaseId=5,ClaimDate='02/20/2005',

Amount=25000,Type='Emergency'}.
CDST = <$C_1$, ($S_1$, $S_2$, $S_3$)> /* $C_1$ is given in Section 2.2 */
θ = {$S_2$:CLAIM(CaseId1=5,ClaimDate1='02/20/2005', Amount1 = 25000,Type1 = 'emergency') }
/* $A_{111}$ and $A_{112}$ are generated from CPAggreg-insert from line 11 */
$A_{111}$ = select PA.SSN,sum(CL.Amount) "$v_{111}$"
from S1_PATIENT PA, S1_CASE CA, S1_CLAIM CL
where PA.SSN = CA.SSN and PA.HealthPlan = 'B'
and CA.CaseId = CL.CaseId and CA.CaseId = CaseId1
group by PA.SSN;
$A_{112}$ = select PA.SSN,sum(CL.Amount) "$v_{112}$"
from S1_PATIENT PA, S1_CASE CA, S3_CLAIM CL
where PA.SSN = CA.SSN and PA.HealthPlan = 'B'
and CA.CaseId = CL.CaseId and CA.CaseId = CaseId1
group by PA.SSN;
/* $A_{121}$ is generated from CPAggreg-insert from line 16 */
$A_{121}$ = select PA.SSN,sum(CL.Amount) "$v_{121}$"
from S1_PATIENT PA, S1_CASE CA, S2_CLAIM CL
where PA.SSN = CA.SSN and PA.HealthPlan = 'B'
and CA.CaseId = CL.CaseId and CA.CaseId = CaseId1
group by PA.SSN;
$V_{121}$ = amount1 + $v_{121}$; /* from line 18 */
$C_{12}$ = return 1 if {$V_{111}$+$V_{112}$+$V_{121}$ > 100000} /* line 36 */
θ($C_{12}$) = return 1 if { θ($V_{111}$)+θ($V_{112}$)+θ($V_{121}$) > 100000 }
/* θ($V_{111}$) is obtained by substituting CaseId1=5 in $A_{111}$ and similarly we calculate θ($V_{112}$) and θ($V_{121}$) */

```
Hence, θ(C12) = return 1 if (50000+30000+25000
> 100000)
Therefore, C1 = C12 = 1 (true). Hence,
constraint C1 is violated by the given update.
```

### B. Discussion

Due to space constraints, we are not able to report algorithm for delete statements: CPAggreg-delete (Constraint Planning involving Aggregates for a delete). CPAggreg-delete proceeds in a similar fashion as the CPAggreg-insert. The only difference is the site where delete is occurring. Line 16 of CPAggreg-insert is modified in the where clause and <cond2> is obtained by negating the variables from θ (negation is done because it's a delete statement). The constraint planning for a modify can be modeled as a delete followed by an insert.

The constraint planning algorithm 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 algorithm is not lost. Also, note that we have not considered the issue of constraint checking in the presence of transactions.

If we have a template of possible update statements, most of the steps of the algorithm can be executed in compile time and when an actual update statement is given, a template match can occur and only the last line of the algorithm (line 41 of CPAggreg-insert) happens at run time. By pushing most of the processing at compile time, we gain efficiency at run time. Hence, constraint checking before the update statement saves lot of time and resources spent on rollbacks and also uses very less time at run time.

## IV. RELATED WORK AND CONCLUSIONS

**Related Work:** Grufman et al. [4] provide an excellent formal description of distributing a constraint check over a number of databases. In their constraint distribution model, constraint check is carried after executing an update statement. They consider semantic integrity constraints involving simple arithmetic predicates. However, our algorithms are much more sophisticated as we perform constraint checks before the updates and thus saving time and resources on rollbacks. Also, we consider semantic integrity constraints involving both arithmetic and aggregate predicates. 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.

Grefen and Widom [3] 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.

**Conclusions:** We have presented constraint checker, an agent based framework for checking global semantic integrity constraints. We proposed algorithms for checking global semantic integrity constraints involving aggregates in the presence of updates. The algorithms check for constraint violations before the update happens; hence, we save on time and resources spent on rollbacks. Most of the processing of the algorithm could happen in compile time; hence we save on the time spent at run time.

Constraint optimizations are part of our on-going future work. We plan to give a performance cost model for our constraint optimizations. We also intend to evaluate the performance of the system under varying conditions.

## V. BIBLIOGRAPHY


[1] Das, S.K. and Williams, M.H. Extending integrity maintenance capability in deductive databases. In the proceedings of the UK ALP-90 Conference, pp.75-111, 1990.

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

[3] Grefen, P. and Widom, J. Protocols for integrity Constraint Checking in Federated Databases. International Journal of Distributed and Parallel Databases, 5(4)

[4] Grufman, S., Samson, F., Embury, S.M., Gray, P.M.D and Risch T. Distributing Semantic Constraints Between Heterogeneous Databases. Proceedings of the Thirteenth International Conference on Data Engineering (ICDE), April, 1997

[5] Grumbach, S.,Rafanelli, M., and Tininini, L. Querying Aggregate Data. PODS 1999.

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

[7] Gupta, A. and Widom, J. Local Verification of Global Integrity Constraints in Distributed Databases. Proceedings of the ACM SIGMOD, 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] Lange,D.B., and Oshima, M. Mobile Agents with Java: The Aglet API. World Wide Web 1(3): 111-121 (1998).

[10] Lin, T. Y. Granular Computing: Fuzzy Logic and Rough Sets, In Computing with words in information/intelligent systems, L.A. Zadeh and J. Kacprzyk (eds), Physica-Verlag (A Springer-Verlag Company), 183-200, 1999

[11] Lin, T. Y. Data Mining and Machine Oriented Modeling: A Granular Computing Approach, Journal of Applied Intelligence, Kluwer, Vol 13, No 2, 2000, 113-124.

[12] 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.

[13] Madiraju, P. and Sunderraman, R. An Efficient Constraint Planning Algorithm for Multidatabases, Proceedings of 2005 ACS/IEEE International Conference on Computer Systems and Applications (AICCSA 2005), Cairo, Egypt, January 3-6, 2005

[14] Martinenghi, D. Simplification of integrity constraints with aggregates and arithmetic built-ins. In proceedings of Flexible Query Answering Systems (FQAS), 2004.

[15] Tan, K-L., Goh, C.H. and Ooi, B.C. Progressive evaluation of nested aggregate queries. The VLDB Journal, Volume 9, Issue 3 (December 2000).

[16] Yao, Y.Y., Perspectives of Granular Computing , 2005 IEEE Conference on Granular Computing, to appear

[17] Zadeh, L.A. Some reflections on soft computing, granular computing and their roles in the conception, design and utilization of information/intelligent systems, Soft Computing, 2, 23-25, 1998.