6

A Framework for Localizing Integrity Constraints Checking in Distributed Database

Ali Amer Alwan, Hamidah Ibrahim and Nur Izura Udzir
Department of Computer Science
Faculty of Computer Science and Information Technology
Universiti Putra Malaysia, 43400 Serdang,
Malaysia

1. Introduction

The validity, accuracy, and semantic of data are significant requirements in modern database applications. Semantic data in database is normally represented under the form of integrity constraints. Integrity constraints are properties, typically depending on the nature of the application domain, which must always be satisfied for the data to be considered *consistent*. Maintaining obedience of data with respect to integrity constraints is an essential requirement, since, if some data lacks integrity, then answers to queries cannot be trusted. Databases usually contain massive collections of data that rapidly evolve over time; this makes perfect checking at each update too time consuming a task to be feasible. In this regard, DBMS needs to be extended with the ability to automatically verify that database updates do not introduce any violation of integrity (Martinenghi, 2005; Christiansen & Martinenghi, 2006). The way we pursue here is the so-called *simplification* of integrity constraints. Simplification means to generate a set of integrity tests from the initial constraints whose satisfaction implies the satisfaction of the original constraints in the updated state. The main interest of the simplification process is to obtain a set of integrity tests (simplified forms) that are as easy to evaluate as possible. In this sense, simplification technique is feasible in terms of the cost of evaluating the constraints. *Integrity constraint checking* is the process of ensuring that the integrity constraints are satisfied by the database after it has been updated. Checking the consistency of a database state will generally involve the execution of integrity tests on the database which verify whether the database is satisfying its constraints or not. The problem of checking integrity constraints in database system has been addressed by many researchers, and has been proved to be extremely difficult to implement, particularly in distributed database. This chapter presents a framework for checking integrity constraints in a distributed database by utilizing as much as possible the information at a local site. This is achieved by considering several types of integrity tests and not focusing only on certain type of test as suggested by previous researchers. In addition, an approach for ranking and selecting suitable integrity tests that reduces the amount of data transferred across the network, the amount of data accessed, and the number of sites involved is also presented. The remainder of this chapter is organized as follows. In Section 2, the previous works related to this research are reported.

Source: Convergence and Hybrid Information Technologies, Book edited by: Marius Crisan,
ISBN 978-953-307-068-1, pp. 426, March 2010, INTECH, Croatia, downloaded from SCIYO.COM

In Section 3, the basic definitions, notation and examples, which are used in the rest of the chapter, are set out. In Section 4, the components of the proposed framework are described followed by some examples. Conclusion and further research are presented in the final section, 5.

2. Related work

For distributed databases, a number of researchers have looked at the problem of semantic integrity checking. Although many research works have been conducted concerning the issues of integrity constraint checking and maintaining in distributed databases but these works failed to exploit the available information at the target site and explore the various types of integrity tests to ensure local checking can always be achieved. This is briefly shown in Table 1, where column labeled 1, 2, 3, 4, 5, 6, 7, and 8 represent the work by Simon and Valduriez (1986), Qian (1989), Mazumdar (1993), Gupta (1994), Ibrahim *et al* (2001), Ibrahim (2002), Madiraju *et al* (2006), and Soumya *et al* (2008) respectively.

Criteria		1	2	3	4	5	6	7	8
Types of integrity constraints	Domain	√	√	√		√	√		
	Key	√	√	√		√	√		
	Referential	√	√	√	√	√	√		
	Semantic	√	√	√		√	√	√	√
	Transition						√		
Types of tests	Complete	√	√						
	Sufficient		√	√	√	√	√	√	√
	Support								

Table 1. Summary of the Previous Work

The work presented in Simon and Valduriez (1986) constructed a simplification method for integrity constraints expressed in terms of assertions for central databases and extended it to distributed databases. This method produces at assertion definition time, differential pre-tests called compiled assertions, which can be used to prevent the introduction of inconsistencies in the database. The cost of integrity checking is reduced because only data subject to update are checked in this approach.

Qian (1989) argued that most approaches derive simplified forms of integrity constraints from the syntactic structure of the constraints and the update operation without exploiting knowledge about the application domain and the actual implementation of the database. Qian (1989) shows that distributed constraints can be translated into constraints on the fragments of a distributed database, given the definition of the fragmentation, and offers a framework for constraint reformulation. The constraint reformulation algorithm used to derive sufficient conditions can potentially be very inefficient because it searches through the entire space of eligible reformulation for the optimal one. Using heuristic rules to restrict the reformulation step may miss some optimal reformulation.

The work presented by Mazumdar (1993) aims at minimizing the number of sites involved in evaluating the integrity constraints in a distributed environment. In his approach the intention is to reduce the non locality of constraints by deriving sufficient conditions not only for the distributed integrity constraints given, but also for those arising as tests for

particular transactions. His method relies on a standard backchaining approach to find the sufficient conditions.

Gupta (1994) presents an algorithm to generate parameterized local tests that check whether an update operation violates a constraint. This algorithm uses the initial consistency assumption, an integrity constraint assertion that is expressed in a subset of first order logic, and the target relation to produce the local test. This optimization technique allows a global constraint to be verified by accessing data locally at a single database where the modification is made. However, this approach is only useful in situations where each site of a distributed DBMS contains one or more intact relations since it does not consider any fragmentation rules.

Ibrahim *et al* (2001) contribute to the solution of constraint checking in a distributed database by demonstrating when it is possible to derive from global constraints localized constraints. They have proved that examining the semantics of both the tests and the relevant update operations reduces the amount of data transferred across the network. The simplified tests have reduced the amount of data that needed to be accessed and the number of sites that might be involved. Ibrahim (2002) extends the work in Ibrahim *et al* (2001) by considering the transition constraints.

The work proposed by Madiraju *et al* (2006) focuses on checking global constraints involving aggregates in the presence of updates. The algorithm takes as input an update statement, a list of global constraints involving aggregates and granules. The sub constraint granules are executed locally on remote sites and the algorithm decides if a constraint is violated based on these sub constraint executions. The algorithm performs constraints checking before the updates and thus saves time and resources on rollback. This approach is limited as they only consider semantic integrity constraints involving both arithmetic and aggregate predicates. Other types of integrity constraints that are important and are frequently used in database applications are not being considered.

Soumya *et al* (2008) proposed a technique to achieve optimization of constraint checking process in distributed databases by exploiting technique of parallelism, compile time constraint checking, localized constraint checking, and history of constraint violations. The architecture mainly consists of two modules: Constraint Analyzer and Constraint Ranker for analyzing the constraints and for ranking the constraints, respectively for systems with relational databases. They achieved optimization in terms of time by executing the constraints in parallel with mobile agents.

From these works, it can be observed that most of the previous works proposed an approach to derive simplified form of the initial integrity constraint with the sufficiency property, since the sufficient test is known to be cheaper than the complete test and its initial integrity constraint as it involved less data to be transferred across the network and always can be evaluated at the target site, i.e. only one site will be involved during the checking process. The previous approaches assume that an update operation will be executed at a site where the relation specified in the update operation is located, which is not always true. For example, consider a relation R that is located at site 1. An insert operation into R is assume to be submitted by a user at site 1 and the sufficient test generated is used to validate the consistency of the database with respect to this update operation, which can be performed locally at site 1. But if the same update operation is submitted at different site, say 2, the sufficient test is no longer appropriate as it will definitely access information from site 1 which is now remote to site 2. Therefore, an approach is needed so that local checking can be performed regardless the location of the submitted update operation. Also, the approach must be able to cater the important and frequently used integrity constraint types.

3. Preliminaries

Our approach has been developed in the context of relational databases. A database is described by a database schema, D, which consists of a finite set of relation schemas, $<R_1, R_2, \ldots, R_m>$. A relation schema is denoted by $R(A_1, A_2, \ldots, A_n)$ where R is the name of the relation (predicate) with n-arity and A_i's are the attributes of R. A relational distributed database schema is described as (D, IC, AS) where IC is a finite set of integrity constraints and AS is a finite set of allocation schemas.

Database integrity constraints are expressed in prenex conjunctive normal form with the range restricted property. A conjunct (literal) is an atomic formula of the form $R(u1, u2, \ldots, uk)$ where R is a k-ary relation name and each ui is either a variable or a constant. A positive atomic formula (positive literal) is denoted by $R(u1, u2, \ldots, uk)$ whilst a negative atomic formula (negative literal) is prefixed by ¬. An (in)equality is a formula of the form $u1\ OP\ u2$ (prefixed with ¬ for inequality) where both $u1$ and $u2$ can be constants or variables and $OP \in \{<, \leq, >, \geq, <>, =\}$. Throughout this chapter the company database is used, as given in Figure 1. This example has been used in most previous works related to the area of constraint checking (Feras, 2006; Ibrahim, 2006; Ibrahim *et al*, 2001; Gupta, 1994).

Schema:
emp(eno, dno, ejob, esal); dept(dno, dname, mgrno, mgrsal); proj(eno, dno, pno)

Integrity Constraints:
Domain Constraint
(*IC*-1) 'The *salary* in relation *emp* must be greater than 0'
$(\forall w \forall x \forall y \forall z)(emp(w, x, y, z) \rightarrow (z > 0))$
Key Constraints
(*IC*-2) '*eno* is the primary key of *emp*'
$(\forall w \forall x1 \forall x2 \forall y1 \forall y2 \forall z1 \forall z2)(emp(w, x1, y1, z1) \wedge emp(w, x2, y2, z2) \rightarrow (x1 = x2) \wedge (y1 = y2) \wedge (z1 = z2))$
(*IC*-3) 'Every department has a unique *dno*'
$(\forall w \forall x1 \forall x2 \forall y1 \forall y2 \forall z1 \forall z2)(dept(w, x1, y1, z1) \wedge dept(w, x2, y2, z2) \rightarrow (x1 = x2) \wedge (y1 = y2) \wedge (z1 = z2))$
Referential Integrity Constraints
(*IC*-4) 'The *dno* of every tuple in the *emp* relation exists in the *dept* relation'
$(\forall t \forall u \forall v \forall w \exists x \exists y \exists z)(emp(t, u, v, w) \rightarrow dept(u, x, y, z))$
(*IC*-5) 'The *eno* of every tuple in the *proj* relation exists in the *emp* relation'
$(\forall u \forall v \forall w \exists x \exists y \exists z)(proj(u, v, w) \rightarrow emp(u, x, y, z))$
(*IC*-6) 'The *dno* of every tuple in the *proj* relation exists in the *dept* relation'
$(\forall u \forall v \forall w \exists x \exists y \exists z)(proj(u, v, w) \rightarrow dept(v, x, y, z))$
(*IC*-7) 'The *mgrno* of every tuple in the *dept* relation exists in the *emp* relation'
$(\forall t \forall u \forall v \forall w \exists x \exists y \exists z)(dept(t, u, v, w) \rightarrow emp(v, x, y, z))$
(*IC*-8) 'The manager salary, *mgrsal*, in the *dept* relation exists in *emp* relation, *esal*'
$(\forall u \forall v \forall w \forall x \exists y \exists z)(dept(u, v, w, x) \rightarrow emp(w, y, z, x))$
General Semantic Integrity Constraints
(*IC*-9) 'Every manager in department *D*1 earns > 4000'
$(\forall w \forall x \forall y \forall z)(dept(w, x, y, z) \wedge (w = \text{'}D1\text{'}) \rightarrow (z > 4000))$
(*IC*-10) 'Every employee must earn ≤ to the manager in the same department'
$(\forall t \forall u \forall v \forall w \forall x \forall y \forall z)(emp(t, u, v, w) \wedge dept(u, x, y, z) \rightarrow (w \leq z))$
(*IC*-11) 'All managers who are working on project *P*3 must earn more than 1000'
$(\forall v \forall w \forall x \forall y \forall z)(dept(v, w, x, y) \wedge proj(x, z, P3) \rightarrow (y > 1000))$
(*IC*-12) 'Any department that is working on a project *P*1 is also working on project *P*2'
$(\forall x \forall y \exists z)(proj(x, y, P1) \rightarrow proj(z, y, P2))$

Fig. 1. The *Company* Static Integrity Constraint

In the database literature, many types and variations of integrity tests have been described (McCune and Henschen, 1989; McCarroll, 1995). The classifications of integrity tests are based on some of their characteristics, as explained below.

a. Based on when the integrity test is evaluated: (i) post-tests - allow an update operation to be executed on a database state, which changes it to a new state, and when an inconsistent result is detected undo this update. The method that applies these integrity tests is called the detection method. (ii) pre-tests - allow an update to be executed only if it changes the database state to a consistent state. The method that applies these integrity tests is called the prevention method.
b. Based on region: (i) local tests – verify the consistency of a database within the local region, i.e. by accessing the information at the local site. The method that adopts these integrity tests is called the local method. (ii) global tests - verify the consistency of a database outside the local region, i.e. by accessing the information at the remote site(s). The method that adopts these integrity tests is called the global method.
c. Based on its properties: (i) sufficient tests - when the test is satisfied, this implies that the associated constraint is satisfied and thus the update operation is safe with respect to the constraint. (ii) necessary tests - when the test is not satisfied, this implies that the associated constraint is violated and thus the update operation is unsafe with respect to the constraint. (iii) complete tests - has both the sufficiency and the necessity properties.

Integrity test based on input	**Integrity test based on region**	**Integrity test based on detection/prevention methods**	**Integrity test based on its properties**
Non-support test	Global test – spans remote site(s)	Post-test – evaluated after an update is performed	Sufficient test
			Necessary test
			Complete test
		Pre-test – evaluated before an update is performed	Sufficient test
			Necessary test
			Complete test
	Local test – spans local site	Post-test – evaluated after an update is performed	Sufficient test
			Necessary test
			Complete test
		Pre-test – evaluated before an update is performed	Sufficient test
			Necessary test
			Complete test
Support test	Global test – spans remote site(s)	Post-test – evaluated after an update is performed	Sufficient test
			Necessary test
			Complete test
		Pre-test – evaluated before an update is performed	Sufficient test
			Necessary test
			Complete test
	Local test – spans local site	Post-test – evaluated after an update is performed	Sufficient test
			Necessary test
			Complete test
		Pre-test – evaluated before an update is performed	Sufficient test
			Necessary test
			Complete test

Table 2. Types of Integrity Tests in Distributed Database

d. Based on the input used to generate the test: (i) non-support tests - these integrity tests are generated based on the update operation and the integrity constraint to be checked, called target integrity constraint, and (ii) support tests - any tests that are derived using other integrity constraints as the support to generate the tests. These types of integrity tests are summarized in Table 2.

4. The proposed framework

Figure 2 illustrates the proposed framework of integrity constraint checking for distributed database systems. This framework is divided into two modules: COMPILE-TIME MODULE and RUN-TIME MODULE which are elaborated in the subsections 4.1 and 4.2, respectively. The proposed framework has been successfully implemented using Visual Basic 6.0 programming language. Each module has been developed and tested with respect to the example database that is considered in this chapter. The major tasks of the framework are to generate the integrity tests for a given update operation, and ranked the selected integrity tests. We do not attempt to discuss in detail the implementation of the components that underpin the framework, but rather present brief results of the implementation of the various components embodied in this framework.

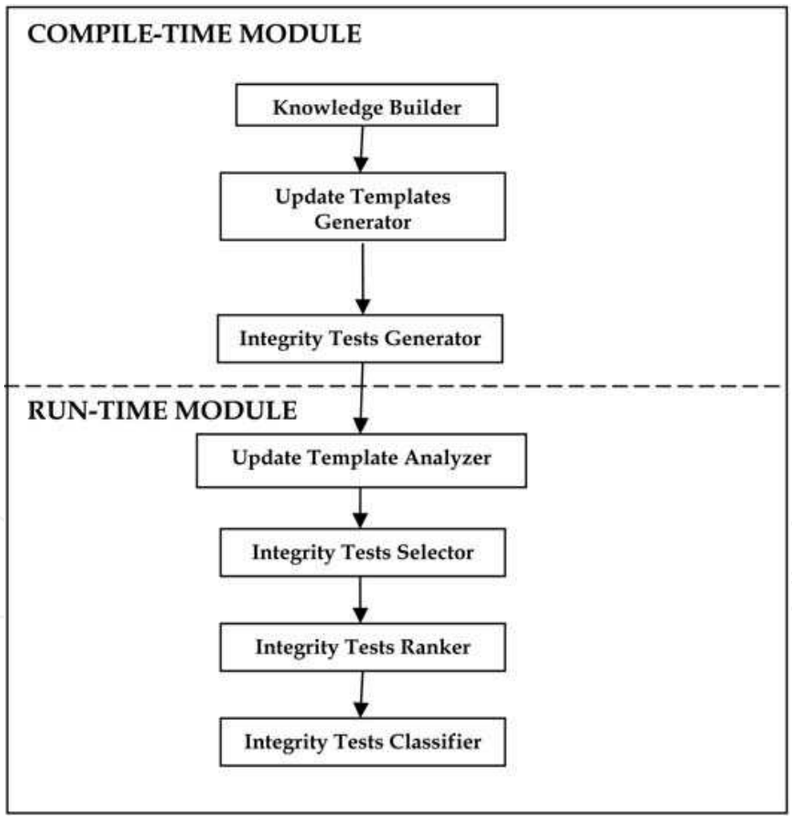

Fig. 2. The Proposed Framework of Constraint Checking

4.1 Components of compile-time module

This module encompasses three components, namely: Knowledge Builder, Update Templates Generator, and Integrity Tests Generator as explained below.

Knowledge Builder: This component analyzes the database schema and integrity constraints for a particular database application. It checks the syntactic correctness of a database schema and extracts some facts that include the names of relations in the database system, names and number of attributes in each relation. In addition, it checks if the input constraints specified by the user are valid and correct with respect to the syntactic formula given in the constraint specification.

Update Templates Generator: The aim of this component is to derive all possible update operations (templates) that might violate a given constraint. The update templates are generated for a particular database by applying the well-known update theorems. These theorems with their proofs can be found in (Nicolas, 1982) and are therefore omitted here.

Integrity Tests Generator: The most important and the essential component in the Compile-Time Module is integrity tests generator. The main operation is to construct the integrity tests by simplifying the integrity constraint which is specified in prenex conjunctive normal form. This component addresses the issue of checking the constraints locally regardless the location of the submitted update operation as elaborated in Section 2. Three types of integrity test are generated by using three different algorithms, namely: complete tests are derived using *Algorithm*-1 proposed by Nicolas (1982); complete/sufficient tests are generated using *Algorithm*-2 proposed by Ibrahim (1998); while support tests are produced using *Algorithm*-3 which is proposed by us. These algorithms adopt the substitution techniques and absorption rules to generate integrity tests (Nicolas, 1982; Ibrahim, 1998). The difference between our algorithm and *Algorithm*-1 and *Algorithm*-2 is that our proposed simplification technique uses other integrity constraints to generate integrity tests while both the *Algorithm*-1 and *Algorithm*-2 used the target integrity constraint (integrity constraint to be checked) as the input. The details of these algorithms are omitted here. Interested readers may refer to Alwan *et al* (2007). Table 3 summarizes the integrity tests generated for the integrity constraints listed in Figure 1 using these algorithms.

In Figure 3 the interface for generating update templates is illustrated for the *Company* database.

Figure 4 presents the interface for the Integrity Tests Generator component that has been implemented for the *Company* database.

4.2 Components of run-time module

The Run-Time Module encompasses four components namely: Update Template Analyzer, Integrity Tests Selector, Integrity Tests Ranker, and Integrity Tests Classifier as elaborated below.

Update Template Analyzer: This component analyzes the syntax of an update operation submitted by a user. It checks that the name of relation and the number of attributes/columns which are specified in the update operation are the same as the name of relation and the number of attributes/columns that appear in the database schema.

Integrity Tests Selector: The main function of this component is to identify the integrity constraints that might be violated given an update operation and select the integrity tests associated to those constraints. This phase is achieved by comparing the real update operation with the update templates that have been generated. This comparison includes checking the name of relation and type of update operation. If both the actual update operation and update template have the same relation name and type of update operation, then the integrity tests of the update template are selected. This is to ensure that only those constraints and their associated integrity tests that might be violated for the given update operation are considered for evaluation.

IC-i	**Update template**	**Integrity test**	**Type of integrity test**
IC-1	insert *emp*(*a*, *b*, *c*, *d*)	1. $d > 0$	Complete Test
IC-2	insert *emp*(*a*, *b*, *c*, *d*)	2. $(\forall x2 \forall y2 \forall z2)(\neg emp(a, x2, y2, z2) \vee [(b = x2) \wedge (c = y2) \wedge (d = z2)])$	Complete Test
		3. $(\forall x1 \forall y1 \forall z1)(\neg emp(a, x1, y1, z1))$	Complete Test
		4. $(\exists v \exists w)(proj(a, v, w))$	Support Test
		5. $(\exists t \exists u \exists w)(dept(t, u, a, w))$	Support Test
IC-3	insert *dept*(*a*, *b*, *c*, *d*)	6. $(\forall x2 \forall y2 \forall z2)(\neg dept(a, x2, y2, z2) \vee [(b = x2) \wedge (c = y2) \wedge (d = z2)])$	Complete Test
		7. $(\forall x1 \forall y1 \forall z1)(\neg dept(a, x1, y1, z1))$	Complete Test
		8. $(\exists t \exists v \exists w)(emp(t, a, v, w))$	Support Test
		9. $(\exists u \exists w)(proj(u, a, w))$	Support Test
IC-4	insert *emp*(*a*, *b*, *c*, *d*)	10. $(\exists x \exists y \exists z)(dept(b, x, y, z))$	Complete Test
		11. $(\exists t \exists v \exists w)(emp(t, b, v, w))$	Sufficient Test
		12. $(\exists u \exists w)(proj(u, b, w))$	Support Test
	delete *dept*(*a*, *b*, *c*, *d*)	13. $(\forall t \forall v \forall w)(\neg emp(t, a, v, w))$	Complete Test
		14. $(\forall u \forall w)(\neg proj(u, a, w))$	Support Test
IC-5	insert *proj*(*a*, *b*, *c*)	15. $(\exists x \exists y \exists z)(emp(a, x, y, z))$	Complete Test
		16. $(\exists v \exists w)(proj(a, v, w))$	Sufficient Test
		17. $(\exists t \exists u \exists w)(dept(t, u, a, w))$	Support Test
	delete *emp*(*a*, *b*, *c*, *d*)	18. $(\forall v \forall w)(\neg proj(a, v, w))$	Complete Test
		19. $(\forall t \forall u \forall w)(\neg dept(t, u, a, w))$	Support Test
IC-6	insert *proj*(*a*, *b*, *c*)	20. $(\exists x \exists y \exists z)(dept(b, x, y, z))$	Complete Test
		21. $(\exists u \exists w)(proj(u, b, w))$	Sufficient Test
		22. $(\exists t \exists v \exists w)(emp(t, b, v, w))$	Support Test
	delete *dept*(*a*, *b*, *c*, *d*)	23. $(\forall u \forall w)(\neg proj(u, a, w))$	Complete Test
		24. $(\forall t \forall v \forall w)(\neg emp(t, a, v, w))$	Support Test
IC-7	insert *dept*(*a*, *b*, *c*, *d*)	25. $(\exists x \exists y \exists z)(emp(c, x, y, z))$	Complete Test
		26. $(\exists v \exists w)(proj(c, v, w))$	Support Test
	delete *emp*(*a*, *b*, *c*, *d*)	27. $(\forall t \forall u \forall w)(\neg dept(t, u, a, w))$	Complete Test
		28. $(\forall v \forall w)(\neg proj(a, v, w))$	Support Test
IC-8	insert *dept*(*a*, *b*, *c*, *d*)	29. $(\exists y \exists z)(emp(c, y, z, d))$	Complete Test
	delete *emp*(*a*, *b*, *c*, *d*)	30. $(\forall u \forall v)(\neg dept(u, v, a, d))$	Complete Test
IC-9	insert *dept*(*a*, *b*, *c*, *d*)	31. $(a <> \text{'}D1\text{'}) \vee (d > 4000)$	Complete Test
IC-10	insert *emp*(*a*, *b*, *c*, *d*)	32. $(\forall x \forall y \forall z)(\neg dept(b, x, y, z) \vee (d \leq z))$	Complete Test
		33. $(\exists t \exists v \exists w)(emp(t, b, v, w) \wedge (w \geq d))$	Sufficient Test
	insert *dept*(*a*, *b*, *c*, *d*)	34. $(\forall t \forall v \forall w)(\neg emp(t, a, v, w) \vee (w \leq d))$	Complete Test
IC-11	insert *dept*(*a*, *b*, *c*, *d*)	35. $(\forall z)(\neg proj(c, z, P3) \vee (d > 1000))$	Complete Test
		36. $(\exists x \exists y \exists z)(emp(c, x, y, z) \wedge (d > 1000))$	Support Test
	insert *proj*(*a*, *b*, *P*3)	37. $(\forall v \forall w \forall y)(\neg dept(v, w, a, y) \vee (y > 1000))$	Complete Test
		38. $(\exists z)(proj(a, z, P3))$	Sufficient Test
		39. $(\exists x \exists y \exists z)(emp(a, x, y, z) \wedge (z > 1000))$	Support Test
IC-12	insert *proj*(*a*, *b*, *P*1)	40. $(\exists z)(proj(z, b, P2))$	Complete Test
		41. $(\exists x)(proj(x, b, P1))$	Sufficient Test
	delete *proj*(*a*, *b*, *P*2)	42. $(\forall x)(\neg proj(x, b, P1))$	Complete Test
		43. $(\exists z)(proj(z, b, P2) \wedge (z <> a))$	Sufficient Test

Table 3. Integrity Tests of the Integrity Constraints of the Example Database

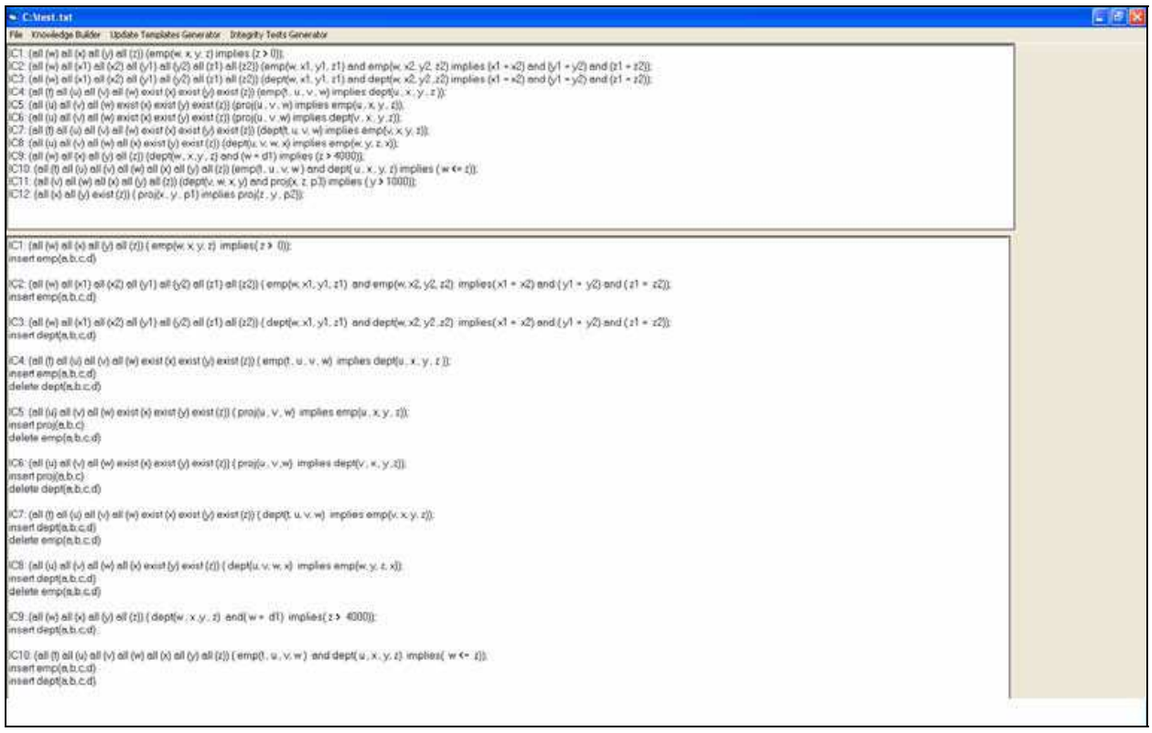

Fig. 3. The List of Update Templates Generated for the *Company* Database

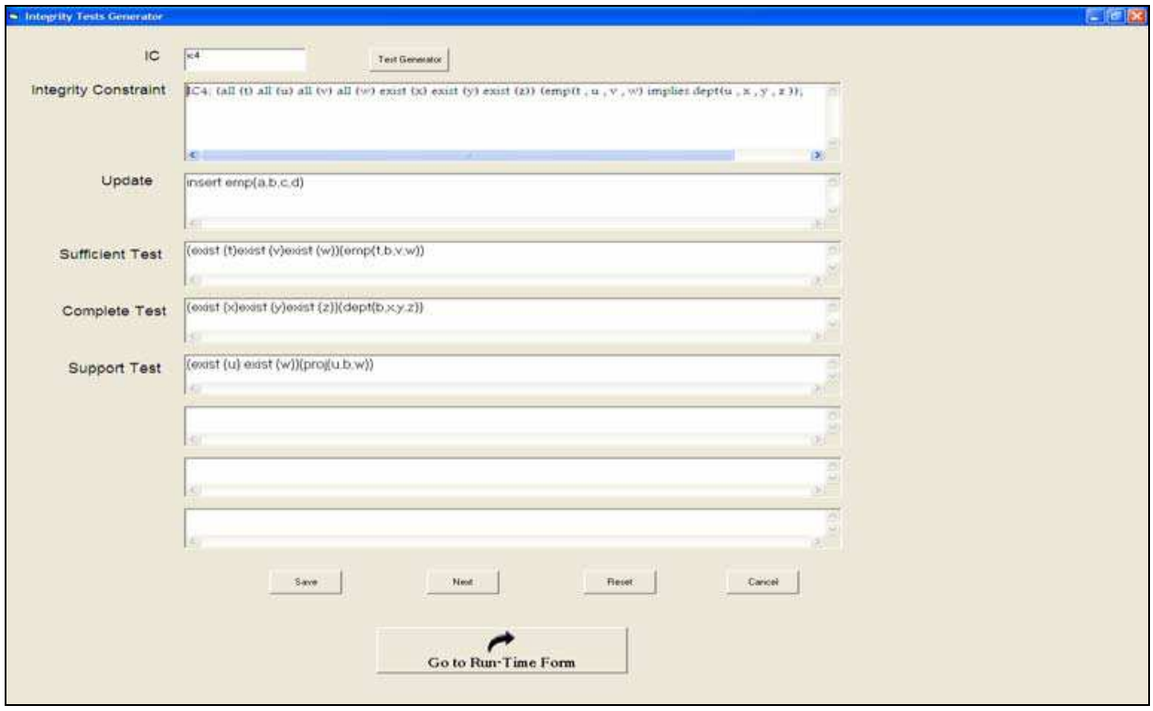

Fig. 4. The Integrity Tests of the Referential Integrity Constraint *IC*-4 for the *Company* Database

Integrity Tests Ranker: The main aim of this component is to rank the selected integrity tests. It attempts to answer the following questions; which test should be selected if there are several alternatives that can be chosen from? What are the criteria that should be measured in order to identify the suitable test?

Most of the works in integrity constraints checking focused on techniques to simplify integrity constraints with the assumption that the simplified forms of the constraints are cheaper than the initial constraints. Thus, the simplified form is evaluated (instead of the initial constraint) to verify the consistency of the database. Moreover, most of the efficiency measurements consider a single cost component and are more applicable for measuring the cost of evaluating integrity constraints in a centralized environment rather than a distributed environment. In addition, most of the previous works consider limited type of integrity tests (complete and sufficient) and depend strictly on the assumption that the update operation is always submitted at the site where the relation specified in the update is located. Thus, their approaches in selecting suitable integrity tests are not general enough. These approaches do not consider support tests and there is no ranking between the types of tests, i.e. all tests are considered the same.

We argue that tests should be ranked as they have different probability of being true or false in a given database state. Thus, we suggest complete test should have the highest priority, followed by sufficient test, and lastly by support test. This is because complete test has both the sufficiency and the necessity properties, sufficient test can only verify for valid database state, and most of the support test is either sufficient test or necessary test. As mentioned in the literature, the amount of data transferred across the network is the most critical factor; therefore we suggest the amount of data transferred to have the highest priority in the ranking model, followed by the number of sites involved, and the amount of data accessed. Based on these arguments, we have proposed a ranking model as shown in Figure 5. Each value in the box, i.e. 1, 2, 3, …, *P*, is the *rank value* where *P* is the maximum rank value. The *rank value* of a test, $Test_i$, with respect to T is denoted by $Rank_T$. Similar notation is used for indicating the rank value of a test with respect to σ and $\mathscr{A}$. Thus, we can calculate the total rank value for a given test by simply adding the rank values for each of the parameter, i.e. $Rank\text{-}Test_i = Rank_T + Rank_\sigma + Rank_{\mathscr{A}}$, where T provides an estimate of the amount of data transferred across the network, $\mathscr{A}$ provides an estimate of the amount of data accessed, and σ gives a rough measurement of the amount of nonlocal access necessary to evaluate a constraint or integrity test. The test with the smallest rank value is said to be the suitable test. A test with the lowest total rank value is said to be the most appropriate test. The ranking model is designed as follows:

1. If the amount of data transferred of a given test is 0 (i.e. the test is a local test), then depending on its property, a value of 1, 2, and 3 is assigned to the $Rank_T$ if the test is a complete, sufficient, and support, respectively. Otherwise for each nonlocal test ($T \neq 0$), the tests are ordered according to the value of T and the test with the lowest T, a value of 4 is assigned to its $Rank_T$. The next lowest, a value of 5 is assigned to its $Rank_T$ and so on.
2. If the number of sites involved in checking a given test is 1, then depending on its property, a value of 4, 5, and 6 is assigned to the $Rank_\sigma$ if the test is a complete, sufficient, and support, respectively. The rank value begins with 4 (and not 1, 2, or 3) to show that the number of sites has lower priority than the amount of data transferred. Otherwise, for each test with $\sigma \neq 1$, the tests are ordered according to the number of sites

involved and the test with the lowest σ, a value of 7 is assigned to its Rank_σ. The next lowest, a value of 8 is assigned to its Rank_σ and so on. Also, note that a test with Rank_σ = 4 (5 and 6, respectively) will definitely not be assigned a Rank_T = 4 (5 and 6, respectively) since Rank_T = 4 (5 and 6, respectively) indicates that the test is a nonlocal test while Rank_σ = 4 (5 and 6, respectively) denotes that the test is a local test. Although they have the same rank value, i.e. 4, but after adding the rank value for both T and σ, the local test will definitely have lower total rank value compared to the nonlocal test.

3. If the amount of data accessed for each of the test is the same, then depending on its property, a value of 7, 8, and 9 is assigned to the $\text{Rank}_{\mathscr{A}}$ if the test is a complete, sufficient, and support, respectively. The rank value begins with 7 (and not 1, 2, …, 6) to show that the amount of data accessed has the lowest priority compared to the amount of data transferred and the number of sites involved. Otherwise, for each test with different amount of data accessed, these tests are ordered according to the amount of data accessed and the test with the lowest $\mathscr{A}$, a value of 10 is assigned to its $\text{Rank}_{\mathscr{A}}$. The next lowest, a value of 11 is assigned to its $\text{Rank}_{\mathscr{A}}$ and so on. Also, note that a test with $\text{Rank}_{\mathscr{A}}$ = 7 (8 and 9, respectively) can be assigned a Rank_σ = 7 (8, 9, …, P_σ) which indicate that the test is a nonlocal complete test (nonlocal sufficient test and nonlocal support test, respectively) and the amount of data accessed is the same for all the alternative tests.

Parameter/ Type of Test	Complete, *C*	Sufficient, *S*	Support, *Sup*	Remarks
$T = 0$	1	2	3	If $T \neq 0$, the tests are rank accordingly based on the amount of data transferred. Rank value begins with 4, 5, 6, …, P_T
$\sigma = 1$	4	5	6	If $\sigma \neq 1$, the tests are rank accordingly based on the number of sites involved. Rank value begins with 7, 8, 9, …, P_σ
$\mathscr{A}_C = \mathscr{A}_S = \mathscr{A}_{Sup}$	7	8	9	If $\mathscr{A}_C \neq \mathscr{A}_S \neq \mathscr{A}_{Sup}$, the tests are rank accordingly based on the amount of data accessed. Rank value begins with 10, 11, 12, …, $P_{\mathscr{A}}$.

Note: P_T (P_σ and $P_{\mathscr{A}}$, respectively) is the maximum rank value assigned to a test based on T (σ and $\mathscr{A}$, respectively).

Fig. 5. The Proposed Ranking Model

To illustrate the ranking model for integrity tests, three scenarios are considered:

i. Centralized database (all relations are located at the same site).
ii. Average case (two relations are located at the same site while the other is located at a different site).
iii. Worst case (each relation is located at different sites).

We assume that *emp* relation contains 500 employees (500 tuples), *dept* relation contains 10 departments (10 tuples), and *proj* relation contains 100 projects (100 tuples).

IC-4 and the insert operation into *emp* relation are used to demonstrate the model, i.e. tests 10 (complete), 11 (sufficient), and 12 (support) are compared.
Based on the result shown in Table 4, complete test, *C*, is selected, as it is the most suitable test for centralized database. Since all tests have similar characteristics with regards to T (= 0) and σ (= 1), the only different are the properties of the tests and $\mathcal{A}$. Mazumdar (1993) scatter metric alone is not able to select the suitable test as these tests have the same scatter metric, σ = 1, while Ibrahim *et al* (2001) will select the test with the lowest $\mathcal{A}$. If the tests have the same amount of data accessed, then no solution is given in Ibrahim *et al* (2001).

Update is submitted at site:	**Location of Relations**		**Rank-$Test_i$**	**Test Selected**
*S*1	*S*1	*emp, dept, proj*	*C* = 1 + 4 + 10 =15 *S* = 2 + 5 + 12 = 19 *Sup* = 3 + 6 + 11 = 20	Test *C*

Table 4. Case (i) Centralized Database

Table 5 presents an average case with several different scenarios. Here, we assume that two of the relations are located at the same site while the other relation is located at a different site, and update operation is submitted at any of these sites. From the results, we observed that local test is always selected regardless the type of the tests. In cases where more than one local test is available, then the tests are rank according to the type and the amount of data accessed (this scenario is similar to the case (i) centralized database discussed earlier).

Update is submitted at site:	**Location of Relations**		**Rank-$Test_i$**	**Test Selected**
*S*1	*S*1	*emp, dept*	*C* = 1 + 4 + 10 = 15 *S* = 2 + 5 + 12 = 19 *Sup* = 4 + 7 + 11 = 22	Test *C*
	*S*2	*proj*		
*S*2	*S*1	*emp, dept*	*C* = 4 + 7 + 10 = 21 *S* = 5 + 7 + 12 = 24 *Sup* = 3 + 6 + 11 = 20	Test *Sup*
	*S*2	*proj*		
*S*1	*S*1	*emp, proj*	*C* = 4 + 7 + 10 = 21 *S* = 2 + 5 + 12 = 19 *Sup* = 3 + 6 + 11 = 20	Test *S*
	*S*2	*dept*		
*S*2	*S*1	*emp, proj*	*C* = 1 + 4 + 10 = 15 *S* = 5 + 7 + 12 = 24 *Sup* = 4 + 7 + 11 = 22	Test *C*
	*S*2	*dept*		
*S*1	*S*1	*dept, proj*	*C* = 1 + 4 + 10 = 15 *S* = 4 + 7 + 12 = 23 *Sup* = 3 + 6 + 11 = 20	Test *C*
	*S*2	*emp*		
*S*2	*S*1	*dept, proj*	*C* = 4 + 7 + 10 = 21 *S* = 2 + 5 + 12 = 19 *Sup* = 5 + 7 + 11 = 23	Test *S*
	*S*2	*emp*		

Table 5. Case (ii) Average Case

Table 6 presents the worst case scenario. In this case each relation is located at different sites. Here, we assume *emp* relation is located at site *S*1, *dept* relation is located at site *S*2, and *proj* relation is located at site *S*3. The test that is selected is the local test (T = 0 and σ = 1). As mentioned earlier, most of the previous works assumed that the update operation is submitted at the site where the relation specifies in the update is located, and thus the sufficient test is always selected. In the ranking model, the suitable test (with the lowest total rank value) is selected and this test can be complete, sufficient or support.

Update is submitted at site:	**Location of Relations**		**Rank-$Test_i$**	**Test Selected**
*S*1	*S*1	*emp*	*C* = 4 + 7 + 10 = 21	Test *S*
	*S*2	*dept*	*S* = 2 + 5 + 12 = 19	
	*S*3	*proj*	*Sup* = 5 + 7 + 11 = 23	
*S*2	*S*1	*emp*	*C* = 1 + 4 + 10 = 15	Test *C*
	*S*2	*dept*	*S* = 5 + 7 + 12 = 24	
	*S*3	*proj*	*Sup* = 4 + 7 + 11 = 22	
*S*3	*S*1	*emp*	*C* = 4 + 7 + 10 = 21	Test *Sup*
	*S*2	*dept*	*S* = 5 + 7 + 12 = 24	
	*S*3	*proj*	*Sup* = 3 + 6 + 11 = 20	

Table 6. Case (iii) Worst Case

Obviously, in some cases, support tests can benefit the distributed database, where local constraint can be achieved. Integrating these various types of integrity tests during constraint checking and not concentrating on certain types of integrity tests (as suggested by previous works) can enhance the performance of the constraint mechanism. Thus, developing an approach that can increase the performance and minimize the cost during the process of constraint checking in the distributed database is important. We have evaluated the model with several cases and several different types of integrity constraints, and in all cases the model is able to select the suitable test as expected.

Integrity Tests Classifier: This component focuses on classifying the integrity tests based on region i.e., into local test or global test. Each test regardless the type can be classified as either local or global depending on where the real update operation is submitted and the location of the relation(s) specified in the integrity tests is located. If the test can be performed locally, then the test is being local. In contrary, when the test needs to transfer data across the network from another site(s) the test is being global. Note that the Integrity Tests Ranker component ranks the integrity tests without selecting any of the tests for evaluation. Only after classification that the test with the lowest total rank value (normally local test) is selected. The *Test Selected* column in tables 4, 5 and 6 is to demonstrate the whole idea of selecting the suitable integrity test to be evaluated from a list of alternative tests. Figure 6 illustrates the interface of the Run-Time Module.

5. Conclusion

In this chapter, we have proposed an approach that performs constraint checking at the target site by utilizing as much as possible the local information to avoid the possibility of transferring data across the network. The novelty of this approach is that local checking can be performed regardless the location of the submitted update operation. This is achieved by having several types of integrity tests and not focusing on certain type of integrity tests as

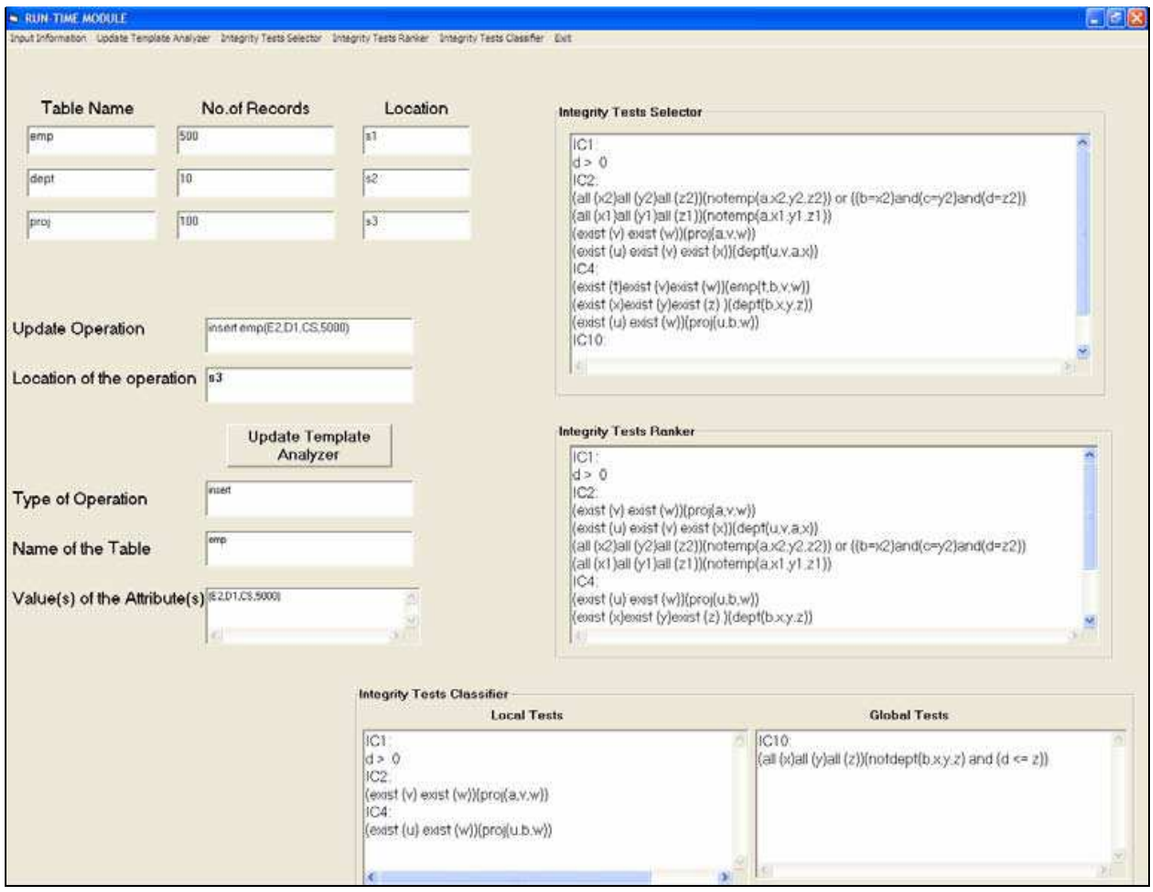

Fig. 6. The Interface of the Run-Time Module

suggested by previous researchers in this area. Also, we have proposed an approach for ranking and selecting suitable integrity tests that reduce the amount of data transferred across the network, the amount of data accessed, and the number of sites involved. Most importantly, we have proved that in most cases, support tests can benefit the distributed database, where local constraint checking can be achieved. Thus, the efficiency of checking constraint process is increased. Both of these strategies are embedded in our proposed framework as presented in this chapter. For future works further enhancement to the proposed approach can be done by considering strategies to maintain the distributed database state when violation occurs. Considering multiple operations or transaction is another area that can be explored where strategies that can minimize the cost of checking the constraints are needed.

6. References

Alwan, A. A.; Ibrahim, H. & Udzir, N. I., (2007). Local Integrity Checking using Local Information in a Distributed Database. *Proceedings of the 1st Aalborg University IEEE.Student Paper Contest 2007 (AISPC'07)*, Aalborg, January 2007, Denmark.

Christiansen H. & Martinenghi D. (2006). On using Simplification and Correction Tables for Integrity Maintenance in Integrated Databases, *Proceedings of the 17th International*

Conference on Database and Expert Systems Applications (DEXA'06), pp. 569 – 576, Krakow, September 2006, Poland.

Feras, A. H. H., (2006). *Integrity Constraints Maintenance for Parallel Databases*. Ph.D. Thesis, Universiti Putra Malaysia, Malaysia.

Gupta A., (1994). *Partial Information based Integrity Constraint Checking*. Ph.D. Thesis, Stanford University, USA.

Ibrahim H. (1998). *Semantic Integrity Constraints Enforcement for a Distributed Database*, Ph.D. Thesis, University of Wales College of Cardiff, Cardiff (UK).

Ibrahim H.; Gray W.A., & Fiddian N.J. (2001). Optimizing Fragment Constraints – A Performance Evaluation, *International Journal of Intelligent Systems – Verification and Validation Issues in Databases, Knowledge-Based Systems, and Ontologies*, Edited by: Ronald, R., John Wiley & Sons Inc.,Vol. 16, No. 3, , 2001, pp. 285 – 306, ISSN 0884-8173.

Ibrahim H. (2002). A Strategy for Semantic Integrity Checking in Distributed Databases, *Proceedings of the Ninth International Conference on Parallel and Distributed Systems*, pp. 139- 144, Republic of China, IEEE Computer Society, December 2002, China.

Ibrahim, H., (2006). Checking Integrity Constraints – How it Differs in Centralized, Distributed and Parallel Databases. *Proceedings of the Second International Workshop on Logical Aspects and Applications of Integrity Constraints*, pp. 563 – 568, Krakow, September 2006, Poland.

Madiraju P.; Sunderraman R., and Haibin W. (2006). A Framework for Global Constraint Checking Involving Aggregates in Multidatabases Using Granular Computing, *Proceedings of IEEE International Conference on Granular Computing (IEEE-GrC'06),* pp. 506 – 509, Atlanta, May 2006, USA.

Martinenghi, D., (2005). *Advanced Techniques for Efficient Data Integrity Checking*. Ph.D. Thesis, Roskilde University, Denmark.

Mazumdar, S., (1993). Optimizing Distributed Integrity Constraints. *Proceedings of the 3rd International Symposium on Database Systems for Advanced Applications*, pp. 327 – 334, Vol. 4, Taejon, April 1993, Korea.

McCarroll N.F. (1995). *Semantic Integrity Enforcement in Parallel Database Machines*, PhD Thesis, Department of Computer Science, University of Sheffield, Sheffield, UK.

McCune, W. W. & Henschen, L. J., (1989). Maintaining State Constraints in Relational Databases: a Proof Theoretic Basis. *Journal of the Association for Computing Machinery*, Vol. 36 No.1, January 1989, pp. 46 – 68, ISSN:0004-5411.

Nicolas, J. M., (1982). Logic for Improving Integrity Checking in Relational Databases. *Acta Informatica*, Vol. 18, No.3, 1982, pp. 227 – 253, ISSN:0001-5903.

Qian, X., (1989). Distribution Design of Integrity Constraints. *Proceedings of the 2nd International Conference on Expert Database Systems*, pp. 205 – 226, Vienna, Virginia, April 1989, USA.

Simon, E. & Valduriez, P., (1986). Integrity Control in Distributed Database Systems. *Proceedings of the 19th Hawaii International Conference on System Sciences*, pp. 622 – 632, Honolulu, Hawaii, January 1986, USA.

Soumya B.; Madiraju, & Ibrahim H. (2008). Constraint Optimization for a System of Relation Databases, *Proceedings of the IEEE 8th International Conference on Computer and Information Technology (CiT 2008)*, pp. 155 - 160, Sydney, July 2008, Australia.

Convergence and Hybrid Information Technologies

Edited by Marius Crisan

ISBN 978-953-307-068-1
Hard cover, 426 pages
Publisher InTech
Published online 01, March, 2010
Published in print edition March, 2010

Starting a journey on the new path of converging information technologies is the aim of the present book. Extended on 27 chapters, the book provides the reader with some leading-edge research results regarding algorithms and information models, software frameworks, multimedia, information security, communication networks, and applications. Information technologies are only at the dawn of a massive transformation and adaptation to the complex demands of the new upcoming information society. It is not possible to achieve a thorough view of the field in one book. Nonetheless, the editor hopes that the book can at least offer the first step into the convergence domain of information technologies, and the reader will find it instructive and stimulating.

InTech Europe
University Campus STeP Ri
Slavka Krautzeka 83/A
51000 Rijeka, Croatia
Phone: +385 (51) 770 447
Fax: +385 (51) 686 166
www.intechopen.com

InTech China
Unit 405, Office Block, Hotel Equatorial Shanghai
No.65, Yan An Road (West), Shanghai, 200040, China
中国上海市延安西路65号上海国际贵都大饭店办公楼405单元
Phone: +86-21-62489820
Fax: +86-21-62489821

IntechOpen