1990

Database Constraint Enforcement: A Decompositional Approach.

Yuejian Sheng
Louisiana State University and Agricultural & Mechanical College

INFORMATION TO USERS

The most advanced technology has been used to photograph and reproduce this manuscript from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

U·M·I

University Microfilms International
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
313/761-4700 800/521-0600

Order Number 9104171

Database constraint enforcement: A decompositional approach

Sheng, Yuejian, Ph.D.

The Louisiana State University and Agricultural and Mechanical Col., 1990

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

DATABASE CONSTRAINT ENFORCEMENT:
A DECOMPOSITIONAL APPROACH

A Dissertation

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

The Department of Computer Science

by

Yuejian Sheng

B.S., Peking University, Beijing, China, 1982

May 1990

ACKNOWLEDGEMENTS

I would like to express my deepest and sincere gratitude to a number of individuals for their help in the research and preparation of this research. Special thanks are due to my thesis advisor, Professor S. S. Iyengar: Throughout the years, Professor Iyengar has been a constant source of advice, guidance, support and encouragement. I marvel at the breadth of his interests and his ability to locate the heart of a problem. Without his support, none of this would have been possible.

Throughout my years at LSU, Dr. S. Kundu has been a scientific inspiration and a fund of common sense. I am impressed with his critical eye and warm heart. His lectures on artificial intelligence and database systems were of great help in the preparation of this research.

I would like to thank Professor Donald H. Kraft for not only the encouragement and help I received from him, but also for his outstanding leadership under which the department has provided a valuable environment for graduate studies. I appreciate the practical experience that I gained here as an instructor and an assistant system manager.

I would also like to express my sincere gratitude to Dr. Jianhua Chen for her carefully reading of this dissertation and for giving me many valuable suggestions; to Dr. Ahmed A. El-Amawy for enhancing my knowledge on super computer architecture and parallel processing; to Dr. James P. Geaghan for serving in my examination committee; to Dr. Si-Qing Zheng for his constant encouragement and support in many ways; and to Dr. Leslie Jones for his initial advice on this research.

My special thanks go to Dr. Barry E. Jacobs of NASA for his valuable suggestions and comments throughout this research and to Dr. Ken Wanderman for his support during the early stage of this research. The initial funding for this research from NASA is greatly appreciated.

Last, but not least, I thank my wife Yan. Her love, understanding, and support have been invaluable to me.

Table of Contents

List of Figures

Abstract

This thesis presents an efficient and integrated approach to integrity constraint checking for advanced database systems. The proposed approach essentially consists of three phases: constraint decomposition, global enforcement strategy and local enforcement tuning.

The central theme of this dissertation is the development of constraint decomposition theory, which can be used to decompose each constraint formula into a set of constraint sub-formulas. The constraint sub-formulas derived from the decomposition satisfy the sufficient conditions imposed by the original constraint and are also much simpler and more efficient to check. The decomposition is done only once for each constraint, at the time of its definition.

The global enforcement strategies, with the application of decomposition theory, are studied in both sequential and parallel environments. The physical characteristics of the database state is used to determine the checking order or to make assignment of constraint sub-formulas to processing elements. It is shown that the performance of constraint enforcement, with the application of the decomposition theory, is much better than that without the application of the decomposition theory.

Local enforcement tuning methods are developed to further simplify each constraint sub-formula derived at constraint decomposition phase. Because of the simplicity of constraint sub-formulas, more efficient and simple methods can be used. Furthermore, more information, such as update types and update data, are available for the purpose of simplification.

The fundamental assumption underlying our approach is that most database updates satisfy integrity constraints. A second assumption is that transactions are localized and database updates are nonuniform in their distribution. Based on these assumptions, the proposed approach will achieve a significant performance increase over previous approaches.

Chapter I: Introduction

1.1 Problem and Motivations

An *integrity constraint* on a database is a restriction on data in the database that defines the relationships between data items. The function of an *integrity subsystem* in a database management system is to maintain the validity of the data in the database, i.e., to insure that all constraints defined upon the database are satisfied. Because database integrity constraint checking is quite an expensive operation and because no efficient checking methodology currently exists, most existing database systems only include a "bare-bones" database integrity subsystem.

With the advent of expert database systems and CAD systems, constraint enforcement is becoming even more expensive than that in traditional database systems because those systems must be able to deal efficiently with the following three problems:

(1) Keeping track of large volumes of data;

(2) Enforcing a large number of integrity constraints;

(3) Dealing efficiently with very complex constraints.

The primary objective of this research effort is to develop a realistic, comprehensive, and efficient approach for the validation of data integrity in future database systems.

In addition, we will provide a framework for future research into problems associated with constraint enforcement in future database systems.

1.2 Related Work

Much work has been done in the area of database integrity in the last fifteen years. A general treatment of the problem can be found in [Dat83, FeS81, Ull82, Ull88]. Work on integrity constraints dates back to the time when the relational data model [Cod70, Cod71] was proposed. At that time, researchers [Fos74, HaM75, Wil72] were mainly concerned with the basic functions and components of semantic-integrity subsystems. Several possible integrity-constraint assertion tools were proposed and preliminary enforcement techniques were investigated [HaD72, Sto75].

At the end of 70's, researchers were aware of the unbearable inefficiency of validating semantic constraints. Some researchers tried to simplify integrity constraints so that integrity checking would involve as little data as possible. Emphasis was placed on finding an equivalent formula for each of the integrity constraints that would be more easier to validate than the original constraint. Blaustein and Bernstein [BBC80, Bla81] propose a methodology employing the analysis of the prefix of a constraint formula to derive a simpler, equivalent formula. However, they consider only single range constraints, and additional effort was needed to recall some information about the database state. Nicolas [Nic82] uses closed well-formed formulas to express integrity constraints in relational databases. Methods are given to simplify integrity constraint checking when an update operation occurs on a current database state which satisfies all of the given integrity constraints. The operations of inserting, deleting, and modifying a tuple in a relation, as well as the transactions of such operations are considered. The theorems in [Nic82] require that incremental checking be applied

after the update is made. Unnecessary incremental checking may be required for certain modification operations since a modification operation is treated as a deletion operation followed by an insertion operation. In [Kob84], all three update operations are considered as individual operations. However, the method is incomplete in the sense that the method only handles certain forms of the tuple calculus such as $\forall$,$\exists$,$\forall\exists$ and $\exists\forall$, and not all of these forms have "if and only if" conditions for incremental checking. Henschen [HMN84] uses the knowledge about the update and the constraint to identify some specific conditions which may cause an integrity violation. His methods depend on predefined updates, and no simplification is considered for u-ranges which are governed by e-ranges. Hsu and Imielinski [HsI85] generalize Blaustein's methods for fast checking of integrity constraints to deal with multiple (multiple-tuple and multiple-range) updates. His method also needs some extra space to record information about database. The goal is to minimize the checking space.

Semantic integrity constraint modeling and enforcement have gained importance in recent years as a result of interest in CAD and Expert Databases. In [ShK86] , the general issues of constraint management are discussed and the roles of constraints in the fields of programming language, database management, and artificial intelligence are examined. In [LaS86, Mor86, ShK86] constraint formalisms and their implementations are investigated.

1.3 Our Approach

In order to achieve our objective, we present an integrated approach to the constraint enforcement problem. The proposed approach consists of three phases:

constraint decomposition, global enforcement strategies and local enforcement tuning.

The central theme of the thesis is the development of a constraint decomposition theory which can be used to decompose each constraint formula into a set of constraint sub-formulas. The constraint sub-formulas derived from the decomposition will satisfy the sufficient condition imposed by the original constraint and will also be much simpler and more efficient to check. The decomposition will be done only once for each constraint, at the time of its definition.

The second phase deals with the global enforcement strategies, using the decomposition theory. The global enforcement strategies in both sequential and parallel environment are studied. The major concerns here are to determine the checking order of constraint sub-formulas in a sequential environment and to distribute constraint sub-formulas to processing elements so that a better performance is achieved.

The third phase concerns the issues on how to efficiently validate each constraint sub-formula. At this phase, more information is available for further simplifying constraint sub-formula.

The general approach is depicted in Figure 1.1. The fundamental difference between our approach and previous approaches is that they failed to take into account the nature of the constraint enforcement problem and the properties of a database and the database update. Realizing the nature of the database constraint enforcement problem, we consider the physical characteristics of the database and the logical structure of integrity constraint formulas at the same time in order to overcome the deficiencies of previous approaches, which fail to either consider both simultaneously. Since the constraint sub-formulas derived from the constraint decomposition phase are much

simpler, and since most database updates satisfy the integrity constraints, the performance improvement over previous approaches is significant.

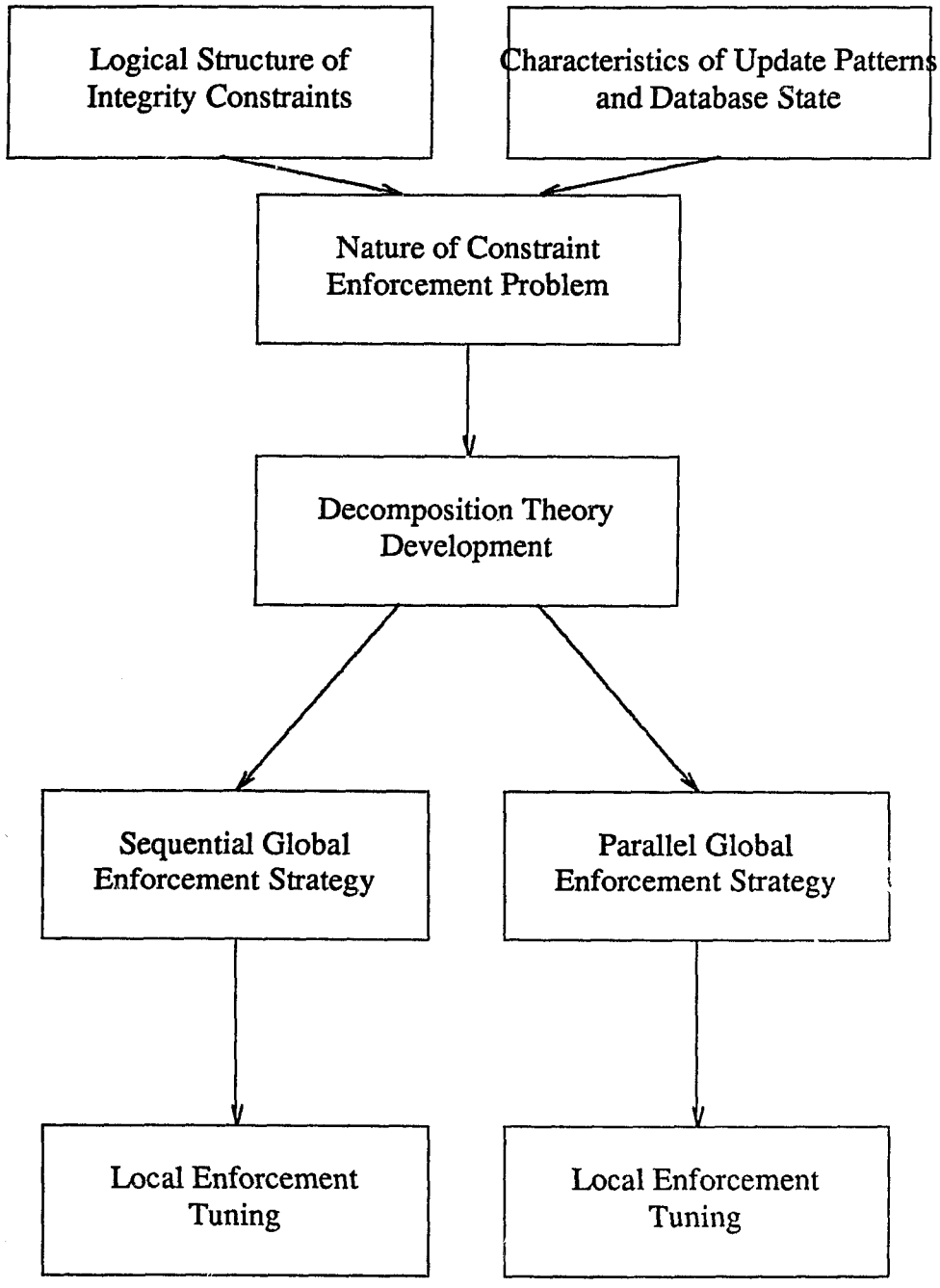

Figure 1.1: General Outline of Our Approach

The rationale behind our proposed methodology can be seen from the following three points:

(1) Nature of the Problem

Previous approaches failed to understand the nature of the constraint validation problem. Constraint validation is a facility provided by a database management system to protect the database from the updates which may violate the integrity of the data stored in a database. The concept of the *protection* should be understood fully here. It is very similar to the protection offered by an insurance company. It is assumed that the probability of constraint violation is extremely small just like insurance companies assume that the probability of an accident is very small.

Constraint validation is not a part of an application program, i.e., it does not do any productive work. Constraints are checked just in case they might be violated. In most cases, a constraint will rarely be violated because database operators or application programmers usually have sufficient knowledge of the data upon which they are working. Constraint are violated by ignorance; so, the chance of violation is quite small.

Figure 1.2 shows the differences between previous approaches and our proposed approach. Generally speaking, previous methods have failed to understand the nature of constraint validation and have made no assumptions on the probability of constraint violation. Thus, the cost of checking is independent of the probability of constraint violation. It can be seen that if the probability of constraint violation is within a certain limit, then our approach will perform much better than previous approaches.

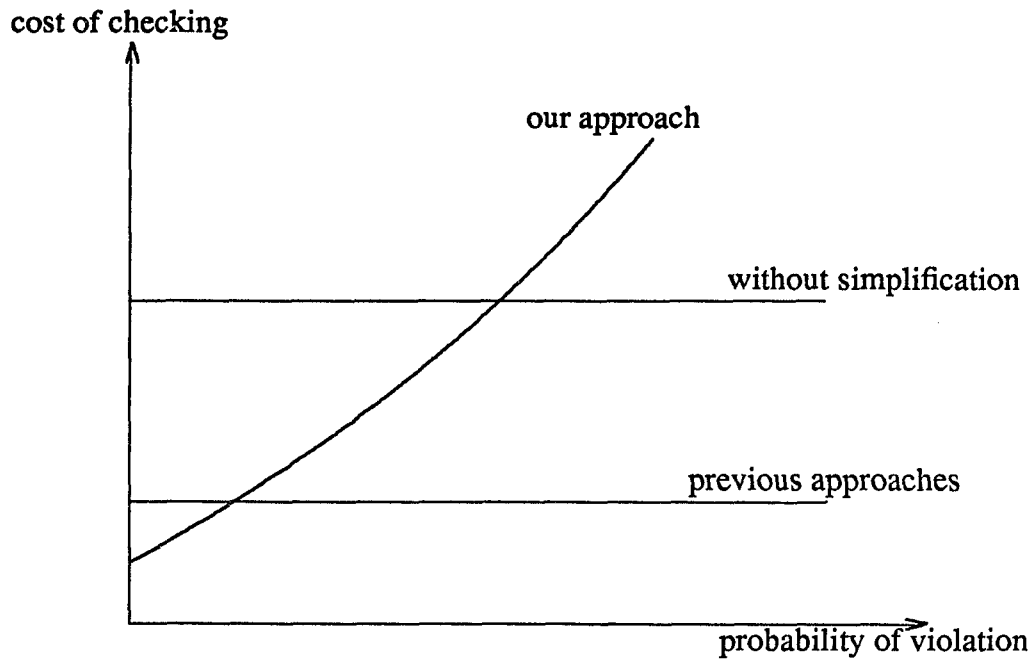

Figure 1.2 Comparisons of Different Approaches

Furthermore, it is noted that most database updates that satisfy the original integrity constraints will also satisfy some kinds of less complicated formulas that also ensure the correctness of the original constraint formulas. Because the probability of constraint violation is small, the sufficient conditions will be met most of time. Only very few updates will violate the sufficient conditions and need to be revalidated against the equivalent or original formula.

(2) Logical Properties of Integrity Constraints

The proposed approach is possible because a complex constraint formula can be decomposed into a set of constraint sub-formulas, each being a sufficient condition of the original formula. The checking of constraint sub-formulas can be done independent of one another and is less costly than that of the original constraint. Because of their independence from one another, the checking of those constraint sub-formulas

can be done in parallel.

(3) Characteristics of Database Updates

The proposed approach is efficient because database updates follow a pattern and the data involved are localized. Some constraint sub-formulas are less costly to check than others are at a given time because:

- The data involved with some constraint sub-formulas are well supported physically (with index, etc), or
- Some constraint sub-formulas are record-oriented other than set-oriented.

1.4 Organization of the Dissertation

The remainder of this dissertation is organized as follows:

Chapter 2 introduces basic concepts related to integrity constraint enforcement in database systems. An example is presented to illustrate the concepts discussed. Chapter 3 discusses the constraint decomposition theory. We first present the decomposition technique and then define the run-time parameters. Chapter 4 discusses the global enforcement strategy in conventional database systems. Chapter 5 investigates the global enforcement strategy in multiprocessor based systems. Parallel checking of constraint sub-formulas will also be discussed. Chapter 6 presents local enforcement tuning methods for simplifying constraint sub-formulas. Chapter 7 concludes this dissertation.

Chapter II: Preliminaries

2.1 Introduction

A database system is an important type of programming system, used today on the biggest and the smallest computers. As for other major forms of system software, such as compilers and operating systems, a well-understood set of principles and concepts for database systems has developed over the years. This chapter introduces the reader to basic concepts and notational conventions related to database systems that we will make use in later discussions.

2.2 Database Abstractions and Data Models

A data model is a language used to express the logical structure of a database and the logical operations that are permitted upon that structure. Thus, a data model is a vehicle not only for defining a database's data elements, relationships, and data types, but also the operations on the elements and relationships. Any data model should include the following three components:

(1) A data structure to represent a user's logical view of the database.

(2) Operations permissible on the data structure which constitute the data language base of the data model.

(3) Constraints for integrity control, i.e., a data model should be equipped with the means to preserve its integrity, to protect itself.

In the past, three particular data models have been dominant: the network model,

the hierarchical model, and the relational model. Numerous database systems and query languages have been based on these models.

In the last decade, much research has been done on semantic data models. Most of this research has been focused on the Entity-Relationship model because it offers the advantages of simplicity and uniformity [Che76]. Recently, database logic [Jac82, Jac85] was proposed as a means to provide a formal view to unify the existing data models.

For the sake of clarity, we have limited our discussions to relational data model. However, the ideas which we put forward are applicable to any data model.

2.3 Relational Model

The relational model was originally proposed by Childs [Chi68] and developed by Codd [Cod70, Cod71]. Extensive discussions of the theory and practice of relational data models can be found in [Dat86, Mai83, Ull82]. For simplicity, a relational database can be viewed as a collection of data tables. In this view, the table's columns are *attributes* and the table's rows are *tuples* corresponding to individual data records. There are no explicit connections among the tables, allowing manipulations of them to be specified simply and flexibly. One broad class of relational data manipulation languages is based on *relational algebra* [Cod70]. Relational algebra is a method of defining operators to transform tables into other tables. Basic operators include restriction, projection, and join. Another broad class of relational data manipulation languages is based on the *relational calculus* [Cod71], an applied predicate calculus.

A relation is typically defined by giving the name of the relation followed by a *relation scheme* consisting of the attribute names of the relation. For example,

SUPPLY(COMP, DEPT, ITEM)

defines a relation called SUPPLY with three attributes, COMP, DEPT and ITEM.

An *instance* of a relation scheme is the "current value" of the relation scheme, consisting of a set of tuples. A *state* of a relational database consists of all instances of relational schemes, which satisfy all of the integrity constraints defined upon the database.

2.4 Problems Affecting the Validity of the Data in a Database

There are four areas in which problems can arise that may affect the validity of the data in a database. They are:

[1] *Reliability*: Errors can be introduced by hardware or software failure. Database software usually includes facilities to restore a database to a state of consistency after such a failure.

[2] *Concurrent Consistency*: Errors can be introduced into a database if multiple users are allowed to simultaneously make changes to the same data item. Database management systems usually include a facility to provide each user with a consistent view of a database, shielding each from interfering effects due to the activities of other users, while retaining a maximum amount of parallelism for concurrent activities.

[3] *Security*: Errors may be intentionally introduced by malicious or unauthorized users. Database management systems usually include a facility to restrict the manner in which a given user may access and manipulate data in a database so as

to protect databases against operations that are actually illegal.

[4] *Semantic Integrity*: Errors may be accidentally introduced by users through typing errors, ignorance or other factors. Database software must include a facility to prevent such semantic errors.

The last area, that of semantic integrity is what we are concerned with.

2.5 Semantic Integrity Subsystem

In this section, we will introduce the concept of the integrity constraint. We will also outline the basic components of a semantic integrity subsystem and discuss constraint enforcement methods.

2.5.1 Semantic Integrity

A database is meant to serve as a model of some limited universe, values in it at any given time representing a particular configuration of that application domain. Every such world has its own internal logic: a set of rules specifying what constitutes a legitimate and plausible configuration. Those rules are called integrity constraints.

There are different kinds of constraints that exist in a real life database systems. Constraint classification has been discussed in [Dat86, IyS87]. A constraint can be *record oriented*, which deals with only one record, or *set oriented*, which needs to deal with all the records in a set [Dat86]. Constraints can be *inter-relation*, which involve the data from more than one relation, or *inner-relation*, which involves the data from only one relation [IyS87]. Constraint can be *transitional* which deals with the data from more than one database state, or *static*, which deals with the data from only the current database state [Dat86]. The classification of those constraints is

important because the cost for checking constraints in different categories is different.

2.5.2 Components of a Semantic Integrity Subsystem

There are five essential components to a semantic integrity subsystem:

[1] *An Assertion Language* : A high-level, nonprocedural language is needed for the user to specify new integrity constraints. These languages are designed to facilitate the specification of:

- The nature of the constraint.
- The time when the constraint is to be validated.
- The action to be taken if the constraint is violated.

[2] *An Optimizing Preprocessor* : An Optimizing Preprocessor is used to translate the high-level constraints into an internal form. This internal form is then used to check for constraint violations and to take appropriate action if violations are detected. The internal representation is also used by the semantic integrity subsystem to determine what constraints need to be checked after any database change.

[3] *A Constraint Enforcer* : A constraint enforcer determines which constraints need to be checked after one or more database changes occur and performs that checking.

[4] *A Violation-Action Processor* : A violation-action processor takes appropriate action whenever a constraint violation is detected by the constraint enforcer. Some possible actions include:

- Report the violation as an error, rejecting the requested database change.
- Report the violation to some appropriate authority, possibly as part of a summary report.
- Initiate automatic corrective action.

[5] *A Constraint Compatibility Checker*: A constraint compatibility checker insures that the set of constraints currently extant for a database is free from conflicts and other undesirable properties.

In this research, we will focus on the *Optimizing Preprocessor* and *Constraint Enforcer* components.

2.5.3 Constraint Validation

A universal method of validating database updates is to use a query evaluation procedure on constraint formulas. The application of a query evaluation procedure to a query yields a relation instance whereas the application of a query evaluation procedure to a constraint formula yields either *true* or *false*. A naive approach is to perform the update and then to check whether the integrity constraints are satisfied in the new database state. If the new database state is inconsistent, then the update must be reversed. This method is impractical because:

[1] The reverse of the violated update is expensive.

[2] Too many data elements must be checked.

As a result, researchers have been developing more efficient constraint validation methodologies. Basically, the following principles are used for simplifying constraints [HsI85].

[1] Limit the range of the integrity checking operation. Assume that the old database state is consistent prior to each update. Therefore, when a database is updated, only the part of the database whose integrity may be affected by the update is the target of integrity checking. For example, consider the constraint that every employee's salary must be higher than 30,000 in the context of updating the database after hiring a new employee. If the old database state is consistent, then the constraint can be simplified to: "the new employee's salary must be higher than 30,000".

[2] Make use of existing information. For example, consider the constraint that there must be at least one manager whose salary is higher than every employee's salary in the context of updating the database after hiring a new employee. If the set of managers (expressed as Vmanager) whose salaries satisfy the constraint in the old database state is known, then the constraint can be simplified to: "there exists a manager in the set Vmanager whose salary is higher than that of the new employee".

[3] Make use of aggregate data. The system keeps track of aggregate data such as *Average, Max,* and *Min*. When those values are needed in checking a constraint, they do not need to be recalculated.

[4] Make use of physical data structures such as index.

Although all of the above mentioned approaches are useful to a certain extent, we still can further improve the efficiency by employing our proposed constraint decomposition methods before applying other simplification methods.

2.6 Methods for Constraint Specification

There are two broad classes of languages for integrity constraint specification:

[1] Logic-based language.

[2] User-based language.

Logic-based languages are mainly used as a framework for theoretical research. The advantages of these languages are their expressive power, conciseness, and well-known semantics. First order language, domain relational calculus, tuple relational calculus, and their variations belong to this category.

User-based languages, as the name suggests, are designed for the user to express integrity constraints. The major concern in designing this kind of language is to provide user friendliness and expressive power. Generally, existing query languages are extended to handle integrity constraints.

2.7 Overview of MTRC

The assertion language used to define integrity constraints throughout this dissertations is MTRC, a Modified Tuple Relational Calculus. This section defines this language.

Primitive Symbols

The following are the primitive symbols of the MTRC language.

- tuple variable and constant symbols.
- function symbols.

- predicate symbols.
- relation names.
- logical connectives: $\neg, \wedge, \vee, \leftarrow, \rightarrow, \leftrightarrow$.
- quantifiers: $\forall, \exists$.
- parentheses: (,).

Using the above symbols, the objective is to specify valid constraint formulas (vcf). The rules for building vcfs require that *terms* and *atomic formulas* be defined first.

Terms

Terms are defined recursively, as follows:

(1) a constant is a *term.*

(2) a tuple variable is a *term.*

(3) If x is a tuple variable and A is an attribute name, then x.A is a *term.*

(4) if f is a function symbol of n arguments and if $t_1, t_2, \cdots, t_n$ are terms, then $f(t_1, t_2, \cdots t_n)$ is a term.

Atomic Formula

Atomic formulas are defined as follows:

- if P is a predicate of n arguments ($n \geq 0$) and if $t_1, t_2, \cdots t_n$ are terms, then $P(t_1, \ldots, t_n)$ is an atomic formula.

Valid Constraint Formula

Valid Constraint formulas are defined as follows:

(1) atomic formulas are vcfs.

(2) if C is a vcf, x is an individual tuple variable and S is a relation name, then $(\forall x \in S)C$ is a vcf.

(3) if C1 and C2 are vcfs, then $\neg(C1)$, $(C1) \vee (C2)$, $(C1) \wedge (C2)$, $(C1) \rightarrow (C2)$ are vcfs.

(4) the only vcfs are those obtainable by finitely many applications of (1), (2), and (3).

2.8 An Illustration

This section present an example to illustrate the concepts we discussed in this chapter. The example is adapted from [Nic82]. Consider a sales company database designed using the relational data model. The database consists of four relations, represented by four different relation schemes. The relational schemes and their intended meaning is shown in Figure 2.1.

A Sales Company Database		
Relation Scheme	**Intended Meaning**	
SUPPLY(COMP, DEPT, ITEM)	<C, D, I>	<=> Company C supplies department D with item I.
CLASS(ITEM, TYPE)	<I, T> ∈ CLASS	<=> I is a type T item.
SALE(DEPT, ITEM)	<D, I> ∈ SALE	<=> Department D sells item I.
SUBORD(EMP, MNG)	$<E_1,E_2>$ ∈ SUBORD	<=> Employee E_1 is a subordinate of E_2

Figure 2.1. Relation Schemes for a Sales Company Database

The seven integrity constraints on this database that are defined using MTRC, along with their English interpretations, are shown in Figure 2.2.

Integrity Constraints

(1) When a department sells an item then there is a company which supplies this item.
$\forall x \in SALE \exists y \in SUPPLY((x.DEPT=y.DEPT) \wedge (x.ITEM=y.ITEM))$

(2) No companies other than company C supplies type T_4 items.
$\forall x \in SUPPLY((\exists y \in CLASS(x.ITEM=y.ITEM) \wedge (y.TYPE=T_4)) \rightarrow (x.COMP=C))$

(3) Any company that supplies guns also supplies bullets.
$\forall x \in SUPPLY((x.ITEM=guns) \rightarrow \exists y \in SUPPLY(x.COMP=y.COMP) \wedge (y.ITEM=bullets))$

(4) Any company that supplies type T_1 items also supplies type T_2 item.
$\forall x \in SUPPLY(\exists y \in CLASS((x.ITEM=y.ITEM) \wedge (y.TYPE=T_1)) \rightarrow$
$\exists z \in SUPPLY \exists w \in CLASS(z.COMP=x.COMP) \wedge (z.ITEM=w.ITEM) \wedge (w.TYPE=T_2))$

(5) No company can supply two different departments with item I.
$\neg \exists x \in SUPPLY \exists y \in SUPPLY((x.COMP=y.COMP) \wedge (x.ITEM=y.ITEM=I) \wedge (x.DEPT \neq y.DEPT))$

(6) Whenever an employee is a subordinate of another employee which is itself a subordinate of a third one then the first one is a subordinate of the later one.
$\forall x \in SUBORD \forall y \in SUBORD((y.MNG=x.EMP) \rightarrow (\exists z \in SUBORD(z.MNG=x.MNG \wedge (Z.EMP=y.EMP))))$

(7) There is at least one type T_3 item which is supplied by every company.
$\exists x \in CLASS((x.TYPE=T_3) \wedge \forall y \in SUPPLY \exists z \in SUPPLY(z.COMP=y.COMP) \wedge (z.ITEM=x.ITEM))$

Figure 2.2 Constraints defined on Sales Company Database

Figures 2.3 through 2.6 show the database state of Sales Company Database at a given time. It can be seen that all of the integrity constraints described in figure 2.2 are satisfied in the current database state.

SUPPLY		
COMP	DEPT	ITEM
C_1	D_1	I_1
C_1	D_2	I_2
C_2	D_2	I_3
C_4	D_3	I_4
C_4	D_2	I_5
C_4	D_1	I_5
C_1	D_1	guns
C_1	D_2	bullets

Figure 2.3 An instance of the SUPPLY relation

Figure 2.3 shows an instance of relation the SUPPLY. It consists of eight tuples. Each tuple records a fact that a company supplies a item to a department.

SALE	
DEPT	ITEM
D_1	I_1
D_2	I_2
D_2	I_3
D_3	I_4
D_2	I_5
D_1	I_5
D_1	guns
D_2	bullets

Figure 2.4 An Instance of the SALE relation.

Figure 2.4 shows an instance of the relation SALE. It consists of eight tuples. Each tuple records the fact that a department sales an item.

CLASS	
ITEM	TYPE
I_1	T_1
I_4	T_4
I_5	T_4
I_2	T_2
I_3	T_3
I_6	T_4

Figure 2.5 An instance of the CLASS relation

Figure 2.5 shows an instance of the CLASS relation. It consists of six tuples. Each tuple records the fact that an item is of a certain type.

SUBORD	
EMP	MNG
E_1	E_2
E_2	E_3
E_1	E_3
E_4	E_5
E_4	E_6
E_7	E_8
E_8	E_9
E_7	E_9

Figure 2.6 An instance of the SUBORD relation.

Figure 2.6 shows an instance of the SUBORD relation. It consists of eight tuples. Each tuple records the fact that an employee is a subordinate of another employee.

2.9 Summary

In the chapter, we introduced the concepts and notations related to constraint enforcement in database systems. We reviewed basic concepts on database in general, and on relational database in particular. An example is presented to show the concepts discussed. We will make use of this example in the later chapters.

Chapter III: Constraint Decomposition

3.1 Introduction

In this chapter, we develop a constraint decomposition theory for the constraint enforcement process. The decomposition of constraint formula is a fundamental step in improving the efficiency of constraint enforcement. The result of the decomposition of constraint formulas can be used either in traditional database management systems running on an uniprocessor based system, or on a database machine with multiprocessor supports.

The organization of this chapter is as follows.

First, we give the overview of the approach. Second, we show the motivation for our approach and argue the necessity for constraint formula decomposition. Third, theories for constraint decomposition are developed and proved. A set of examples will be presented to illustrate the decomposition procedure and other concepts discussed above. Following that, we define run-time parameters to measure run-time characteristics, such as the update pattern and the nature of database state. Finally, we summarize and discuss the main concepts presented in this chapter.

3.2 Outline of Constraint Decomposition

The basic idea of constraint decomposition is to decompose a constraint formula C_i into a set of sub-formulas, $C_i^1, C_i^2, \cdots C_i^{m_i}$, such that each of the sub-formulas C_i^j (j=1 ,2 ,... m_i). is a sufficient condition for the original formula, i.e. $C_i^j \rightarrow C_i$ (j=1,2, $\cdots$,m_i). As a result, as long as any one of the sub-formulas is satisfied, the

original constraint will be satisfied. Furthermore, run-time parameters are defined to measure the run-time characteristics of database behavior, such as the update patterns and the nature of database state, so that we have an objective standard by which to choose a constraint sub-formula which is more likely to be satisfied and which costs least to check.

3.3 Motivations and Necessity

Constraint decomposition is a natural result of the searching for more efficient constraint enforcement methodologies. The motivations and necessity of constraint decomposition lies in the following facts.

[1] Characteristics of future database

- Large volume of data
- Large volume of constraints
- Complexity of constraints

Because of this, the cost of checking the validity of database update is very high. Thus, the need for any possible improvement of efficiency is apparent.

[2] The equivalent constraint formulas are much harder to find than a less strict formula that only satisfy the sufficient condition. Previous work on trying to find an equivalent and simpler formula for original formula is not effective because

(1) either it cannot find a equivalent formula for the original constraint formula, or

(2) the efficiency gained by the simplified constraint formula is limited.

[3] Most database updates satisfy integrity constraints, because database operators and application programmers have knowledge about what they are doing. The purpose of integrity constraint checking is to prevent errors introduced by ignorance of database operators or application programmers.

[4] Most of database updates that satisfy integrity constrains also satisfy some kind of less complex formulas which ensure the correctness of the original formula.

[5] Most simpler, sufficient constraint subformulas are easier to check than its complex original formula.

[6] Locality of transactions and nonuniform distribution of database updates are two of the characteristics of modern database updates. Because of those characteristics, some constraint formulas are easier to satisfy than others.

The above mentioned facts show the two points. First, the cost of constraint enforcement in future database is too big to not be considered. Second, constraint decomposition may be of help because of the characteristics of database update and database state.

3.4 Decomposition Theories

In this section, we present and discuss the underlying theory in decomposing an integrity constraint C_i into a set of sub-formulas $C_i^1, \cdots, C_i^{m_i}$, such that each of the sub-formulas is a sufficient condition for the original integrity constraint. Before presenting our theorems, the following definitions are in order.

Definition 3.1: (*Relational Range Prefix*)

The form (Qx∈ X), where Q is a universal (∀) or existential (∃) quantifier, x is a tuple variable and X is a relation name, is called a *Relational Range Prefix.*■

Definition 3.2: (*Prenex Normal Form*)

A valid constraint formula(vcf) $(Q_1 x_1 \in X_1)...(Q_n x_n \in X_n)$ C, where each Q_i is a universal or existential quantifier, x_i is a tuple variable, $x_i \neq x_j$ for $i \neq j$, X_i is a relation name and C contains no relational range prefixes, is said to be in *Prenex Normal Form.*■

Example 3.1

The following equation:

$$\forall x \in SUPPLY\, \exists y \in SUPPLY\, ((x.ITEM \neq guns) \vee (x.COMP = y.COMP) \wedge (y.ITEM = bullets)) \quad (3.1)$$

is a constraint formula. Here, ∀x∈ SUPPLY, ∃y∈ SUPPLY are relational range prefixes, and the formula is in prenex normal form because all of the relational prefixes are in the front of this constraint formula.■

Definition 3.3: (*Disjunctive Normal Form*)

A constraint formula is said to be in *Disjunctive Normal Form* if it is a disjunction consisting of one or more disjuncts, each of which is a conjunction of one or more atomic formulas or negation of atomic formulas.■

Example 3.2

Formula 3.2 is in disjunctive normal form.

$$\forall x \in SUPPLY\, \exists y \in SUPPLY\, (x.ITEM \neq guns) \vee$$
$$(\forall x \in SUPPLY\, \exists y \in SUPPLY)(x.COMP = y.COMP) \wedge (y.ITEM = bullets)) \quad (3.2)$$

With the definitions of the above concepts, we can start to develop our constraint decomposition theory.

Lemma 3.1

Let C_1 and C_2 be valid constraint formulas and R be a relation name, then the following properties hold:

$$(\forall x \in R) C_1 \vee (\forall x \in R) C_2 \rightarrow (\forall x \in R)(C_1 \vee C_2) \quad (3.3)$$
$$(\exists x \in R) C_1 \vee (\exists x \in R) C_2 \equiv (\exists x \in R)(C_1 \vee C_2) \quad (3.4)$$

Proof: The equation (3.3) and (3.4) can be obtained by applying the following properties in predicate logic.

$$(\forall y) P_1 \vee (\forall y) P_2 \rightarrow (\forall y)(P_1 \vee P_2)$$
$$(\exists y) P_1 \vee (\exists y) P_2 \equiv (\exists y)(P_1 \vee P_2)$$

■

Lemma 3.2

Let Ω be a series of relational range prefixes and C_1 and C_2 be valid constraint formulas(vcfs), then

$$\Omega C_1 \vee \Omega C_2 \rightarrow \Omega(C_1 \vee C_2) \quad (3.5)$$

Proof: We prove the above theorem using induction on the number of relational range

prefixes contained in Ω.

Basis of induction:

For n=1, the equation (3.5) is of either of the following two form:

$$(\forall x \in R)C_1 \vee (\forall x \in R)C_2 \rightarrow (\forall x \in R)(C_1 \vee C_2)$$
$$(\exists x \in R)C_1 \vee (\exists x \in R)C_2 \rightarrow (\exists x \in R)(C_1 \vee C_2)$$

The above two formulas are the direct consequences of the equation (3.3) and (3.4) respectively.

Induction step:

Suppose equation (3.5) holds for n=k, that is:

$$(Q_k Q_{k-1} \cdots Q_1)C_1 \vee (Q_k Q_{k-1} \cdots Q_1)C_2 \rightarrow (Q_k Q_{k-1} \cdots Q_1)(C_1 \vee C_2)$$

Then for n=k+1, the left-hand side is:

$$(Q_{k+1}Q_k \cdots Q_1)C_1 \vee (Q_{k+1}Q_k \cdots Q_1)C_2$$

Moving Q_{k+1} out, we have

$$(Q_{k+1})((Q_k \cdots Q_1)C_1) \vee (Q_{k+1})((Q_k \cdots Q_1))C_2$$

Applying lemma 3.1, the above formula implies the following formula.

$$(Q_{k+1})((Q_k \cdots Q_1)C_1 \vee (Q_k \cdots Q_1))C_2$$

Applying induction hypothesis, the above formula implies the following formula.

$$(Q_{k+1})(Q_k \cdots Q_1)(C_1 \vee C_2)$$

Thus, the equation holds for any integer n. ■

Theorem 3.1

Let Ω be a series of relational range prefixes and $C_1, C_2, \cdots, C_n$ be valid constraint formulas(vcfs), then

$$\Omega C_1 \vee \Omega C_2 \vee \cdots \vee \Omega C_n \rightarrow \Omega(C_1 \vee C_2 \vee \cdots \vee C_n) \tag{3.6}$$

Proof: We prove the above theorem using induction on the number of constraint formulas contained in each side.

Basis of induction:

For n=1, the equation 3.6 becomes:

$$\Omega C_1 \rightarrow \Omega(C_1) \tag{3.7}$$

Which is obviously true.

Induction Step:

Suppose equation (3.6) holds for n=k, that is:

$$\Omega C_1 \vee \Omega C_2 \vee \cdots \vee \Omega C_k \rightarrow \Omega(C_1 \vee C_2 \vee \cdots \vee C_k) \tag{3.8}$$

Then for n=k+1, the left-hand side is:

$$\Omega C_1 \vee \Omega C_2 \vee \cdots \vee \Omega C_k \vee \Omega C_{k+1}$$

By separating $k+1^{th}$ sub-formula from the other k sub-formulas, we have:

$$(\Omega C_1 \vee \Omega C_2 \vee \cdots \vee \Omega C_k) \vee \Omega C_{k+1}$$

By applying induction hypothesis, we have:

$$\Omega(C_1 \vee C_2 \vee \cdots \vee C_k) \vee \Omega C_{k+1}$$

By applying lemma 3.2, we have:

$$\Omega(C_1 \vee C_2 \vee \cdots \vee C_k \vee C_{k+1})$$

Thus the equation (3.6) holds for any integer n. ■

Theorem 3.1 provides the theoretical basis for decomposing a constraint formula in the form $\Omega(C_1 \vee C_2 \vee \cdots \vee C_k \vee C_{k+1})$ into a set of independent constraint formulas $\Omega C_1, \Omega C_2, \cdots, \Omega C_k, \Omega C_{k+1}$, each of which is a sufficient condition for the original constraint formula. Because of this, each of

the constraint formulas derived can be validated independently.

Example 3.3

the following equation is in prenex disjunctive normal form.

$$\forall x \in SUPPLY\, \forall y \in CLASS\, \exists z \in SUPPLY\, \exists w \in CLASS\, ((x.ITEM \neq y.ITEM) \vee (y.TYPE \neq T_1))$$
$$\vee (z.COMP = x.COMP) \wedge (z.ITEM = w.ITEM) \wedge (w.TYPE = T_2))$$

By applying theorem 3.1, we get the following three different constraint formulas, each of which is a sufficient condition of the original one:

$$C_1 = \forall x \in SUPPLY\, \forall y \in CLASS\, \exists z \in SUPPLY\, \exists w \in CLASS\, (x.ITEM \neq y.ITEM)$$
$$C_2 = \forall x \in SUPPLY\, \forall y \in CLASS\, \exists z \in SUPPLY\, \exists w \in CLASS\, (y.TYPE \neq T_1)$$
$$C_3 = \forall x \in SUPPLY\, \forall y \in CLASS\, \exists z \in SUPPLY\, \exists w \in CLASS\, ((z.COMP = x.COMP) \wedge$$
$$(z.ITEM = w.ITEM) \wedge (w.TYPE = T_2))$$

■

Lemma 3.3

Let C_1 and C_2 be any valid constraint formulas, then the following properties hold.

$$((\forall x \in R) C_1(x) \rightarrow C_2) \equiv (\exists y \in R)(C_1(y) \rightarrow C_2) \qquad (3.9)$$

$$((\exists x \in R) C_1(x) \rightarrow C_2) \equiv (\forall y \in R)(C_1(y) \rightarrow C_2) \qquad (3.10)$$

$$C_2 \rightarrow (\forall x \in R) C_1(x) \equiv (\forall y \in R)(C_2 \rightarrow C_1(y)) \quad (3.11)$$

$$C_2 \rightarrow (\exists x \in R) C_1(x) \equiv (\exists y \in R)(C_2 \rightarrow C_1(y)) \quad (3.12)$$

$$\neg(\forall x \in R) C_1 \equiv (\exists x \in R) \neg C_1 \quad (3.13)$$

$$\neg(\exists x \in R) C_1 \equiv (\forall x \in R) \neg C_1 \quad (3.14)$$

Proof: The above properties are the direct consequences of the following properties in first order logic.

$$(\forall x C_1(x) \rightarrow C_2) \equiv \exists y (C_1(y) \rightarrow C_2)$$

$$(\exists x C_1(x) \rightarrow C_2) \equiv \forall y (C_1(y) \rightarrow C_2)$$

$$C_2 \rightarrow \forall x C_1(x) \equiv \forall y (C_2 \rightarrow C_1(y))$$

$$C_2 \rightarrow \exists x C_1(x) \equiv \exists y (C_2 \rightarrow C_1(y))$$

$$\neg \forall x C_1 \equiv \exists x \neg C_1$$

$$\neg \exists x C_1 \equiv \forall x \neg C_1 \quad \blacksquare$$

Theorem 3.2

Let C be an arbitrary valid constraint formula, then it can be transformed into an equivalent constraint formula C' which is of the prenex normal form.

Proof: We use the induction on the number n of connectives and quantifiers in C.

Basis of induction:

For n=0, $C' \equiv$ C, i.e., C is already in prenex normal form.

Induction steps:

Suppose for n<k we have $C' \equiv$ C. Assume C has k connectives and quantifiers. We consider the following three cases.

Case 1: if C=¬B, then by the induction hypothesis, we can construct a vcf D in prenex normal form such that B≡D. Hence, ¬B≡¬D, i.e., C≡ ¬D. But, by applying equation (3.13) and (3.14) we can find a vef C' in prenex normal form such that ¬D≡C'. Hence, C≡C'.

Case 2: if C is $B_1 \rightarrow B_2$, then by the induction hypothesis, we can find vcfs B_1', B_2' in prenex normal form such that $B_1 = B_1'$ and $B_2 = B_2'$. Hence C ≡ $B_1' \rightarrow B_2'$. By applying equations (3.9)-(3.12), we can move the quantifiers in the prefixes of B_1' and B_2' to the front, obtaining a vcf C' in prenex normal form with C≡C'.

Case 3: If C=(∀x∈ R) B, then by the inductive hypothesis, there is a vcf B' in prenex normal form such that B≡B'. Hence (∀x∈R)B≡(∀x∈R)B', i.e., C≡(∀x∈R)B'. But (∀x∈R)B' is in prenex normal form. ■

The purpose of theorem 3.2 is to provide the theoretical foundation for transforming an arbitrary constraint formula into a formula in prenex normal form. When a constraint is in prenex normal form, it then can be transformed into prenex disjunctive normal form by the next theorem. A set of independent constraint formulas can then be derived, based on theorem 3.1.

Example 3.4

The following equation is a constraint formula. Let's see how it can be transformed into a formula in prenex normal form.

$$\forall x \in SUPPLY\,(\exists y \in CLASS\,((x.ITEM = y.ITEM) \wedge (y.TYPE = T_1)) \rightarrow$$
$$\exists z \in SUPPLY\,\exists w \in CLASS\,(z.COMP = x.COMP) \wedge (z.ITEM = w.ITEM) \wedge (w.TYPE = T_2))$$

After applying equation (3.9), we have

$$\forall x \in SUPPLY\,\forall y \in CLASS\,(((x.ITEM = y.ITEM) \wedge (y.TYPE = T_1)) \rightarrow$$
$$\exists z \in SUPPLY\,\exists w \in CLASS\,(z.COMP = x.COMP) \wedge (z.ITEM = w.ITEM) \wedge (w.TYPE = T_2)))$$

After applying equation (3.12), we have

$$\forall x \in SUPPLY\,\forall y \in CLASS\,\exists z \in SUPPLY\,(((x.ITEM = y.ITEM) \wedge (y.TYPE = T_1)) \rightarrow$$
$$\exists w \in CLASS\,(z.COMP = x.COMP) \wedge (z.ITEM = w.ITEM) \wedge (w.TYPE = T_2)))$$

After applying equation (3.12) again, we have the following equation, which is in prenex normal form.

$$\forall x \in SUPPLY\,\forall y \in CLASS\,\exists z \in SUPPLY\,\exists w \in CLASS\,(((x.ITEM = y.ITEM) \wedge (y.TYPE = T_1))$$
$$(z.COMP = x.COMP) \wedge (z.ITEM = w.ITEM) \wedge (w.TYPE = T_2)))$$

■

Lemma 3.3

Let C be an arbitrary valid constraint formula without quantifiers. Then it can be transformed into an equivalent formula which is in the disjunctive normal form. ■

Proof: The above lemma can be derived using the following properties from proposition logic.

$$C_1 \leftrightarrow C_2 \equiv (C_1 \rightarrow C_2) \wedge (C_2 \rightarrow C_1) \tag{3.15}$$

$$C_1 \rightarrow C_2 \equiv \neg C_1 \vee C_2 \tag{3.16}$$

$$\neg(\neg C_1) \equiv C_1 \tag{3.17}$$

$$\neg(C_1 \vee C_2) \equiv \neg C_1 \wedge \neg C_2 \tag{3.18}$$

$$\neg(C_1 \wedge C_2) \equiv \neg C_1 \vee \neg C_2 \tag{3.19}$$

$$C_1 \vee (C_2 \wedge C_3) \equiv (C_1 \vee C_2) \wedge (C_1 \vee C_3) \tag{3.20}$$

$$C_1 \wedge (C_2 \vee C_3) \equiv (C_1 \wedge C_2) \vee (C_1 \wedge C_3) \tag{3.21}$$

■

The idea of lemma 3.3 is to provide a basis for systematically decomposing constraint formulas into subformulas with certain properties.

Example 3.5

The following is a constraint formula not in disjunctive normal form.

$$\forall x \in SUPPLY\, \forall y \in CLASS\, ((x.ITEM = y.ITEM) \wedge (y.TYPE = T_4) \rightarrow (x.COMP = C))$$

After applying equation 3.16 we have:

$$\forall x \in SUPPLY\, \forall y \in CLASS\, (\neg((x.ITEM = y.ITEM) \wedge (y.TYPE = T_4)) \vee (x.COMP = C))$$

After applying equation 3.14, we have following formula, which is in disjunctive normal form.

$$\forall x \in SUPPLY\, \forall y \in CLASS\, ((x.ITEM \neq y.ITEM) \vee (y.TYPE \neq T_4) \vee (x.COMP = C))$$

■

Theorem 3.3 (*Decomposition Theorem*)

For any valid constraint formula C, there exists a set of valid constraint subformulas $C_1, C_2, \cdots, C_n$ such that

$$C \leftarrow C_1 \vee C_2 \vee \cdots \vee C_n \tag{3.22}$$

Where $C_i, (i = 1,2,...,n)$ is of the form $(Q_1^i \cdots Q_{n_i}^i) M_i$. Q_j^i is a relational range prefix and M_i is a conjunction of atomic formulas without any relational range prefix.

Proof: The proof of the decomposition theorem can be derived by applying *Theorem 3.2, Lemma 3.3, and Theorem 3.1*, in that order. ■

Theorem 3.3 establishes the theoretical foundation for our proposed global decomposition method. We can now present the decomposition algorithm based on the constraint decomposition theory.

3.5 A Decomposition Algorithm

In the last section, we have presented the theories for constraint decomposition. We are now ready to present an algorithm for constraint decomposition. The algorithm takes a valid constraint formula as the input and produces a set of constraint sub-formulas as the outputs. Each sub-formula , in the form of conjunction of atomic constraint formulas, is a sufficient condition for the original formula and is in prenex normal form.

Input: A valid constraint formula.

Output: A set of valid constraint sub-formulas.

Step-0: If the constraint is already in the form of a conjunction of atomic formulas, then exit.

Step-1: If the constraint is not in prenex normal form then move the relational range prefix to the front of constraint formula by using Theorem 3.2.

Step-2: If the matrix of the formula is not in disjunctive normal form, then derive a disjunctive normal form by using lemma 3.3.

Step-3: Derive each constraint sub-formula for each disjunct appearing in the formula by using theorem 3.1.

Step-4: Remove unnecessary relational range prefix from each constraint sub-formula.

Step-5: Discard any constraint sub-formulas which are equivalent to other sub-formula by renaming some tuple variables.

Figure 3.1. A Constraint Decomposition Algorithm

3.6 Examples

The examples to be presented in this section serve two purposes. First, they illustrate the algorithm developed above. Second, we show intuitively under what circumstances the performance of constraint enforcement can be improved. The constraints in those examples are from the sales company database introduced in chapter 2.

Example 3.6

Consider the first constraint listed in figure 2.2.

Original Constraint

When a department sells an item there is a company which supplies this item.

$$\forall x \in SALE \exists y \in SUPPLY ((x.DEPT = y.DEPT) \wedge (x.ITEM = y.ITEM))$$

Because the original constraint is already in its simpliest form, no decomposition is done in this case.

Example 3.7

Consider the second constraint listed in figure 2.2.

(1) Original Constraint

No other companies than company C supplies type T_4 items.

$$\forall x \in SUPPLY ((\exists y \in CLASS (x.ITEM = y.ITEM) \wedge (y.TYPE = T_4)) \rightarrow (x.COMP = C))$$

(2) After first step, we have the following formula in prenex normal form:

$$\forall x \in SUPPLY \forall y \in CLASS ((x.ITEM = y.ITEM) \wedge (y.TYPE = T_4) \rightarrow (x.COMP = C))$$

(3) After second step, we have the following formula in prenex disjunctive normal form:

$$\forall x \in SUPPLY \forall y \in CLASS ((x.ITEM \neq y.ITEM) \vee (y.TYPE \neq T_4) \vee (x.COMP = C))$$

(4) After third step, the following sub-formulas are generated:

$$C_1 = \forall x \in SUPPLY \forall y \in CLASS (x.ITEM \neq y.ITEM)$$

$$C_2 = \forall x \in SUPPLY \forall y \in CLASS (y.TYPE \neq T_4)$$

$C_3 = \forall x \in SUPPLY \, \forall y \in CLASS \, (x.COMP = C)$

(5) After fourth step, we have the following simplified sub-formulas by removing the unnecessary relational range prefix:

$C_1 = \forall x \in SUPPLY \, \forall y \in CLASS \, (x.ITEM \neq y.ITEM)$

$C_2 = \forall y \in CLASS \, (y.TYPE \neq T_4)$

$C_3 = \forall x \in SUPPLY \, (x.COMP = C)$

(6) The fifth step does not have any effect because there are no sub-formulas which are equivalent. So we still have the same sub-formulas.

As a result, the decomposition procedure decomposes the constraint in this example into three distinct constraint sub-formulas. The satisfaction of any one of the constraint sub-formulas will guarantee the satisfaction of the original formula. Suppose that we need to insert over 100 tuples in the SUPPLY relation, with x.COMP=C, then constraint sub-formula C_3 will always be satisfied. As a result, a lot of time will be saved.

Example 3.8

Consider the third constraint listed in Figure 2.2.

(1) Original Constraint

Any company that supplies guns also supplies bullets.

$$\forall x \in SUPPLY \, ((x.ITEM = guns) \rightarrow \exists y \in SUPPLY \, (x.COMP = y.COMP) \wedge (y.ITEM = bullets))$$

(2) After first step, we have the following formula in prenex normal form:

$$\forall x \in SUPPLY \, \exists y \in SUPPLY \, ((x.ITEM = guns) \rightarrow (x.COMP = y.COMP) \wedge (y.ITEM = bullets))$$

(3) After second step, we have the following formula in prenex disjunctive normal form:

$$\forall x \in SUPPLY \exists y \in SUPPLY ((x.ITEM \neq guns) \vee (x.COMP = y.COMP) \wedge (y.ITEM = bullets))$$

(4) After third step, we have the following constraint sub-formulas:

$$C_1 = \forall x \in SUPPLY \exists y \in SUPPLY (x.ITEM \neq guns)$$

$$C_2 = \forall x \in SUPPLY \exists y \in SUPPLY ((x.COMP = y.COMP) \wedge (y.ITEM = bullets))$$

(5) After fourth step, we have the following simplified constraint sub-formulas by removing unnecessary relational range prefixes:

$$C_1 = \forall x \in SUPPLY (x.ITEM \neq guns)$$

$$C_2 = \forall x \in SUPPLY \exists y \in SUPPLY ((x.COMP = y.COMP) \wedge (y.ITEM = bullets))$$

(6) The step 5 does not have any effect because there are no constraint sub-formulas which are equivalent. So, we still have the same sub-formulas.

For this example, if we insert a set of tuples into SUPPLY with x.ITEM≠ guns, then C_1 will always be satisfied, which in turn guarantees the satisfaction of the original constraint.

Example 3.9

This is the fifth constraint listed in figure 2.2.

(1) Original Constraint

No company must supply two different departments with item I_5.

$$\neg \exists x \in SUPPLY \exists y \in SUPPLY ((x.COMP = y.COMP) \wedge (x.ITEM = y.ITEM = I_5) \wedge (x.DEPT \neq y.DEPT))$$

(2) After first step, we have the following formula in prenex normal form.

$$\forall x \in SUPPLY\, \forall y \in SUPPLY\, ((x.COMP \neq y.COMP) \vee$$
$$(x.ITEM \neq y.ITEM) \vee (x.ITEM \neq I_5) \vee (y.ITEM \neq I_5) \vee (x.DEPT = y.DEPT))$$

(3) After second step, we have the following formula in prenex disjunctive normal form.

$$\forall x \in SUPPLY\, \forall y \in SUPPLY\, ((x.COMP \neq y.COMP) \vee$$
$$(x.ITEM \neq y.ITEM) \vee (x.ITEM \neq I_5) \vee (y.ITEM \neq I_5) \vee (x.DEPT = y.DEPT))$$

(4) After third step, we have following constraint sub-formulas.

$$C_1 = \forall x \in SUPPLY\, \forall y \in SUPPLY\, (x.COMP \neq y.COMP)$$
$$C_2 = \forall x \in SUPPLY\, \forall y \in SUPPLY\, (x.ITEM \neq y.ITEM)$$
$$C_3 = \forall x \in SUPPLY\, \forall y \in SUPPLY\, (x.ITEM \neq I_5)$$
$$C_4 = \forall x \in SUPPLY\, \forall y \in SUPPLY\, (y.ITEM \neq I_5)$$
$$C_5 = \forall x \in SUPPLY\, \forall y \in SUPPLY\, (x.DEPT = y.DEPT)$$

(5) After fourth step, we have the following simplified constraint sub-formulas.

$$C_1 = \forall x \in SUPPLY\, \forall y \in SUPPLY\, (x.COMP \neq y.COMP)$$
$$C_2 = \forall x \in SUPPLY\, \forall y \in SUPPLY\, (x.ITEM \neq y.ITEM)$$
$$C_3 = \forall x \in SUPPLY\, \forall y \in SUPPLY\, (x.ITEM \neq I_5)$$
$$C_4 = \forall x \in SUPPLY\, \forall y \in SUPPLY\, (y.ITEM \neq I_5)$$
$$C_5 = \forall x \in SUPPLY\, \forall y \in SUPPLY\, (x.DEPT = y.DEPT)$$

(6) After fifth step, the following sub-formulas are left.

$$C_1 = \forall x \in SUPPLY\, \forall y \in SUPPLY\, (x.COMP \neq y.COMP)$$
$$C_2 = \forall x \in SUPPLY\, \forall y \in SUPPLY\, (x.ITEM \neq y.ITEM)$$
$$C_3 = \forall x \in SUPPLY\, (x.ITEM \neq I_5)$$
$$C_4 = \forall x \in SUPPLY\, \forall y \in SUPPLY\, (x.DEPT = y.DEPT)$$

For this example, if we insert a set of tuples into SUPPLY relation with x.ITEM $\neq I_5$, then, C_3 will always be satisfied. This will guarantees the satisfaction of the origi-

nal constraint.

3.7 Run-time Parameters

After the constraint decomposition stage, any integrity constraint C_i has been decomposed into $C_i^1, C_i^2, \cdots C_i^{m_i}$, which satisfy the following implication.

$$C_i \leftarrow C_i^1, C_i^2, \cdots, C_i^{m_i}$$

We now define run-time parameters which will characterize the patterns of database update and the states of database.

Definition 3.4: *Sufficiency Frequency*

For an integrity constraint C_i, suppose that

$$C_i \leftarrow (C_i^1 \vee C_i^2 \vee \cdots \vee C_i^{m_i})$$

where C_i^j (j=1,2, $\cdots m_i$) are vcfs. If there exist a set of values σ_i^j (j=1,2, ... m_i) such that σ_i^j measures the relative probability that C_i^j would be satisfied after a database update and $\sum_{j=1}^{m_i} \sigma_i^j = 1$, then σ_i^j is called a *Sufficiency Frequency*.■

Definition 3.5: *Cost Coefficient*

For an integrity constraint C_i, suppose that

$$C_i \leftarrow (C_i^1 \vee C_i^2 \vee \cdots \vee C_i^{m_i})$$

where C_i^j (j=1,2, ... m_i) are vcfs. If there exist a set of values δ_i^j (j=1,2,...m_i) such that δ_i^j measures the relative cost to check the satisfiability of C_i^j at a given instance, and $\sum_{j=1}^{m_i} \delta_i^j = 1$, then δ_i^j is called a *Cost Coefficient* ■

The sufficiency frequency and the cost coefficient are very important parameters in our attempt to improve the efficiency of database integrity enforcement. The introduction of these two parameters is the natural result of the physical characteristics of database state and the update pattern. Because database updates are localized and the time needed to access different data sets are different, the costs of checking different constraint sub-formulas are different. That is why we need a cost coefficient to measure the effect of the update pattern at a given time.

When a constraint formula is decomposed into a set of constraint sub-formulas based on our method, the satisfaction of any one of the constraint sub-formulas is sufficient to guarantee the satisfaction of the original formula. Because of the locality of database updates, some constraint sub-formulas will be satisfied more frequently than others will at a given time. Thus, the sufficiency frequency is needed to measure the relative satisfiability of constraint sub-formulas of a given constraint formula.

The method to compute and update these run-time parameters will be discussed in the next chapter.

3.8 Summary

In this chapter, we have discussed how to compile an integrity constraint formula into a set of subformulas such that each of them is a sufficient condition for the original formula. The rationale behind this approach is mainly because of two reasons. First, there is no general way to find an equivalent formula that is much simpler than the original formula. Second, since database operations generally follow some pat-

terms and most of time database constraints get satisfied, some sufficient condition are more likely to be satisfied than others. With run-time dynamic maintenance of database state and update parameters, we can get global optimal strategy for checking an integrity constraint.

It should be noted that the time spent in decomposing a constraint formula into a set of constraint sub-formulas can be ignored because of the following reasons:

[1] It is done only once at the time when a constraint formula is defined.

[2] It is done in main memory. Thus, it is much faster than retrieving information from secondary memory.

The set of constraint sub-formulas derived, based on the theories developed in this chapter, can be validated in conventional database management systems running on uniprocessor based systems, or validated on database machines with multiprocessor support.

Chapter IV: Enforcement Strategy : A Sequential Realization

4.1 Introduction

This chapter investigates a possible constraint enforcement strategy with the application of the decomposition theory developed in the previous chapter. Chapter 3 presented a theoretical foundation for constraint decomposition. With a set of constraint sub-formulas derived by the constraint decomposition method, how we are going to enforce those constraint sub-formulas so as to guarantee the validity of the original constraint? What is the efficiency gained by using the constraint decomposition? Those questions will be partially answered in this chapter.

Specifically, we are going to discuss constraint enforcement strategies in conventional database management systems running on uni-processor based systems. We first discuss the strategy to maintain run-time parameters. The objective here is to accurately record database update patterns and current state of database. Then, a global checking procedure that makes use of those run-time parameters will be presented and discussed. Finally, the performance of this enforcement strategy will be analyzed both in terms of checking time and checking space.

4.2 Maintenance of Run-time Parameters

In chapter 3, we defined two run-time parameters: the sufficiency frequency (SF) and the cost coefficient(CC). They together record the characteristics of the current state and the update pattern of a database. To reflect the continuous change of the database, those parameters need to be updated each time the database is modified, as database modification changes the state of the database and the update pattern changes

over time. In this section, the methods for dynamically recomputing sufficiency frequencies and cost coefficients will be presented and discussed.

4.2.1 Computing SF and CC

From their definition, SF measures the relative probability that a constraint subformula will be satisfied upon a database update, and CC measures the relative cost of checking the satisfiability of a constraint subformula. Since a database is an evolving entity, its state is changing over time, CC cannot be a constant. Likewise, since update pattern also has its locality, SF will also be changing.

In the following, we shall present a method to recompute those parameters each time a database update occurs.

Procedure for Computing SF and CC

[1] *Initialization*: For any constraint C and its associated sufficient sub-formulas $C_1, C_2, \cdots, C_n$, let $\mu_i = \sigma_i = \delta_i = \frac{1}{n}$; μ_i is the cost of checking constraint sub-formula C_i, σ_i is the sufficient frequency for C_i and δ_i is the cost coefficient for C_i.

[2] *Update SF:* Upon an update, if the kth subformula is satisfied, update SF in the following way.

$$\sigma_i = \frac{\sigma_i}{\sum_{j=1}^{n} \sigma_j + \xi} \qquad (i \neq k) \tag{4.1}$$

$$\sigma_k = \frac{\sigma_k + \xi}{\sum_{j=1}^{n} \sigma_j + \xi} \tag{4.2}$$

where ξ is a constant.

[3] *Update CC:* Suppose after update, let the subformulas for a constraint C be renamed as $C_{t_1}, C_{t_2}, ..., C_{t_p}, C_{u_1}, C_{u_2}, \cdots, C_{u_q}$; where $p + q = n$. The subformulas with subscript t_i ,$(1 \le i \le p)$ are those which have been checked for validity and the cost for checking each of these subformulas are represented as $\mu_{t_1}, \mu_{t_2}, \cdots, \mu_{t_p}$. The subformulas with subscript u_j ,$(1 \le j \le q)$ are those which have not been checked for validity, and the cost for checking each of these formulas are previous values, which can also be represented similarly by $\mu_{u_1}, \mu_{u_2}, \cdots, \mu_{u_q}$. Since the database is changing, the cost for checking each subformula is also changing. Thus, we update the CC correspondingly by the following equation:

$$\delta_k \equiv \frac{\mu_k}{\sum_{i=1}^{p} \mu_i + \sum_{j=1}^{q} \mu_j} \qquad (1 \le k \le n) \tag{4.3}$$

Example 4.1

Suppose $C = \{C_1, C_2, C_3, C_4, C_5, C_6\}$ and $\xi = 1$. Then after initialization, we have the following:

$$\mu = \delta = \begin{bmatrix} \frac{1}{6} \\ \frac{1}{6} \\ \frac{1}{6} \\ \frac{1}{6} \\ \frac{1}{6} \\ \frac{1}{6} \end{bmatrix} \qquad \sigma = \begin{bmatrix} \frac{1}{6} \\ \frac{1}{6} \\ \frac{1}{6} \\ \frac{1}{6} \\ \frac{1}{6} \\ \frac{1}{6} \end{bmatrix}$$

Upon an update, if sub-formula 3 is satisfied, then we update sufficiency frequencies according to equation (4.1) and (4.2)

$$\sigma_i = \frac{\frac{1}{6}}{1+1} = \frac{1}{12} \quad i = 1, 2, 4, 5, 6$$

$$\sigma_3 = \frac{\frac{1}{6}+1}{1+1} = \frac{7}{12}$$

Suppose $\mu_1 = 8$, $\mu_2 = 6$ and $\mu_3 = 12$, then,

$$\delta_1 = \frac{8}{26.5} = 0.30 \qquad \delta_4 = \frac{\frac{1}{6}}{26.5} = 0.0063$$

$$\delta_2 = \frac{6}{26.5} = 0.23 \qquad \delta_5 = \frac{\frac{1}{6}}{26.5} = 0.0063$$

$$\delta_3 = \frac{12}{26.5} = 0.45 \qquad \delta_6 = \frac{\frac{1}{6}}{26.5} = 0.0063$$

In summary, after update, the matrix for μ, δ and σ are as follows:

$$\mu = \begin{bmatrix} 8 \\ 6 \\ 12 \\ 0.167 \\ 0.167 \\ 0.167 \end{bmatrix} \qquad \delta = \begin{bmatrix} 0.30 \\ 0.23 \\ 0.45 \\ 0.0063 \\ 0.0063 \\ 0.0063 \end{bmatrix} \qquad \sigma = \begin{bmatrix} \frac{1}{12} \\ \frac{1}{12} \\ \frac{7}{12} \\ \frac{1}{12} \\ \frac{1}{12} \\ \frac{1}{12} \end{bmatrix}$$

4.2.2 Discussions

In section 4.2.1, we presented a procedure to update the sufficiency frequency and the cost coefficient. The idea of the proposed update procedure is to make those run-time parameter up to date, i.e., reflecting the most current state and update pattern of the database. The following two points are made about the update procedure.

(1) The Frequency of Updating Run-time Parameters

The above proposed procedure is invoked each time a database update occurs. This will be best if the update of those run-time parameters does not take any time. However, the computation of a run-time parameter does take some time, so other considerations on when a system invokes the update procedure for run-time parameters are necessary.

In the following, we list a few possible methods.

[1] *Periodically recomputing:* This approach calls for recomputing the sufficiency frequency and cost coefficient every so often at fixed time interval. A system timer is set to that interval. Whenever the timer times out, the procedure for updating the run-time parameters will be called.

[2] *Transaction Unit*: This approach calls for recomputing the sufficiency frequency and cost coefficient whenever a transaction occurs.

[3] *Update Unit*: This approach calls for recomputing the sufficiency frequency and cost coefficient every fixed number of updates. A system counter is used to record the number of updates processed. When the counter reaches a predefined number, the procedure for updating the run-time parameters will be called.

The above are just a few methods that can be used instead of updating run-time parameters for every update. All the methods have pros and cons. Choosing which method to use depends on the attributes of a database.

(2) On Choosing the Value of ξ

The value of ξ reflects the rate of change of the sufficiency frequency. A constant value of ξ in the computation will gradually change the relative values of the sufficiency frequencies. For a database that evolves smoothly, it is better to fix the value of ξ. When a database changes sharply, we may use a bigger value of ξ to immediately reflect that change.

4.3 Global Checking Procedure

Since we have defined the database state dependent parameters, SF and CC, and provided a method to recompute them, the problem remaining is to give an overall checking process.

The following is what an integrity enforcer will do after a database update request.

[1] Choose the subformula in the decreasing order of $\frac{\sigma}{\delta}$ to check, continue until one of them is satisfied.

[2] If step [1] fails, then check against the original or an equivalent formula.

[3] If either [1] or [2] is successful, acknowledge the update, otherwise reject the update.

[4] Update SF and CC appropriately.

Example 4.2

Suppose $C = \{C_1, C_2, C_3, C_4, C_5\}$ and

$$\delta = \begin{bmatrix} C_1 & \frac{1}{8} \\ C_2 & \frac{1}{4} \\ C_3 & \frac{1}{4} \\ C_4 & \frac{3}{8} \\ C_5 & \frac{1}{8} \end{bmatrix} \qquad \sigma = \begin{bmatrix} C_1 & \frac{1}{16} \\ C_2 & \frac{1}{8} \\ C_3 & \frac{7}{16} \\ C_4 & \frac{1}{4} \\ C_5 & \frac{1}{8} \end{bmatrix}$$

Thus,

$$\frac{\sigma}{\delta} = \begin{bmatrix} C_3 & \frac{7}{4} \\ C_5 & 1 \\ C_4 & \frac{2}{3} \\ C_1 & \frac{1}{2} \\ C_2 & \frac{1}{2} \end{bmatrix}$$

So, the checking order will be C_3, C_5, C_4, C_1 and C_2

The fundamental philosophy underlying the above proposed procedure is that we choose a constraint sub-formula with maximum $\frac{\sigma}{\delta}$ to check. In this way, we may spend the least time to successfully pass the validation. As we can see, we did not

choose a constraint sub-formula with the least cost coefficient to check, nor we choose a constraint sub-formula with the maximum sufficiency frequency to check, because the one with the least cost may have minimum sufficiency frequency and the one with maximum sufficiency may have maximum cost coefficiency.

4.4 Performance Analysis

It is very difficult to quantify the cost of different integrity enforcement methods, yet this task is essential for precisely comparing them. As with any cost method, it is difficult to capture all the factors which affect the final cost and to assign a relative cost to each factor. Integrity enforcement cost depends on factors such as the structure of the integrity to be verified, the type and frequency of update, the storage structure of the database, the database state, and the values in the update.

Furthermore, there are different measurement methods to evaluate performance. The most popular one to use is checking time to measure the performance, Although some researchers use checking space to measure the performance [HsI85].

In this section, we first introduce some basic concepts and notations. Then, we analyze performance both in terms of checking time and checking space.

4.4.1 Basic Concepts, Notations, and Assumptions

This section introduces the basic concepts, notations, and assumptions needed for performance analysis.

4.4.1.1 Basic Concepts

In this sub-section, we introduce the concepts of checking time and checking space. For the performance analysis, we can either use checking time or checking space to measure the cost of constraint enforcement.

Checking Time

Checking time is a method to measure the time elapsed in checking an integrity constraint. Checking time can be measured in terms of disk read/writes, the number of data items examined, or an arbitrary time unit. The objective of various optimization techniques for integrity enforcement is to reduce the checking time. Measuring performance using checking time is the most popular method.

Checking Space

Checking space is a method to measure the complexity in checking an integrity constraint. Checking space can be measured by the number of bounded variables in the integrity constraint formula. It is generally believed that the fewer the bounded variables inside the formula, the less costly it is to check the formula. Thus, the objective of various optimization techniques for integrity enforcement is to reduce the checking space. The concept of checking space originally appeared in [HsI85]

Formally, checking space is defined by the following definition:

DEFINITION 4.1: *Checking Space*

Suppose that $C(S_1,S_2,\cdots,S_m)$ is decomposed into $C_1(S_{11},S_{12},\cdots,S_{1i_1})$, $C_2(S_{21},S_{22},\cdots,S_{2i_2})$,..., $C_n(S_{n1},S_{n2},\cdots,S_{ni_n})$, such that $C \leftarrow C_1 \vee C_2 \vee \cdots \vee C_n$,

where S_i and S_{jk} are tuple variables ($1 \le i \le m$, $1 \le j \le n$, $1 \le k \le m$), then the checking space of $C(S_1,S_2,\cdots,S_m)$ is defined as $S_1 \times S_2 \times \cdots \times S_m$. The *Checking Space* of $C_1 \vee C_2 \vee \cdots \vee C_n$ is defined as $\sum_{j=1}^{n} S_{j_1} \times S_{j_2} \times \cdots \times S_{j_{i_i}}$ ■

4.4.1.2 Notations and Assumptions

The following assumptions and notations are used throughout this chapter.

[1] A constraint formula C has been decomposed into a set of constraint subformulas $C_1 \cdots C_n$, where each one is a sufficient condition for the original constraint formula C.

[2] The cost for checking constraint C without application of constraint decomposition method is denoted by T_c. The cost for checking constraint C with the application of constraint decomposition method is denoted by T_c'.

[3] The cost for checking constraint sub-formula C_i is T_{c_i}.

[4] The sum of costs for checking each individual subformula is equal to the cost for checking the original constraint C, i.e., $T_c = \sum_{i=1}^{n} T_{c_i}$.

[5] The cost incurred mainly comes from retrieving a tuple from secondary memory and not from computation done in the CPU.

[6] σ_i is the sufficiency frequency associated with C_i, measuring the relative probability that C_i is satisfied when an update occurs.

[7] δ_i is the cost coefficient associated with C_i, measuring the relative cost to check C_i.

With these assumptions, we can now analyze the performance of constraint enforcement with the application of constraint decomposition theory.

4.4.2 Sources of Cost in Constraint Enforcement

There are three major sources of cost in enforcing a constraint, namely,

[1] Cost incurred by the compilation of a constraint.

[2] Cost incurred by the computation of data or updating run-time parameters.

[3] Cost incurred by retrieving data from secondary memory.

The cost for compiling a constraint formula C into a set of subformulas C_i $(i=1,\cdots,n)$ can be neglected because of the following:

- The compilation is done only once for each formula at the time it is defined.
- The compilation is done in memory, thus the cost of compilation is much less than that of retrieval of a tuple from secondary memory.

The updating of run-time parameters and the computation of the data are also done in CPU. Compared with the cost in retrieving data from secondary storage, this cost can also be neglected.

The cost for retrieving data from secondary storage is the major cost in validating a integrity constraint. So, the analysis below is all in terms of the retrieving cost.

4.4.3 Validation Cost with Respect to Checking Time

In this section. we present several theorems to show the improvement of the validation cost with respect to checking time when constraint decomposition technique is employed. Each theorem deals with different update pattern and state of the database,

as well as the characteristics of the given constraint. Because validation cost is the major cost for enforcement of integrity constraints, any possible improvement of performance should be exploited.

It will be seen that the theorems have considered only some typical cases, some of which are extreme, others of which are regular. However, in practice, the characteristics of a real life database lie in between these cases. Thus the results arrived here are more or less applicable to a real life database, provided that it meets the characteristics of a future database discussed in the previous chapters.

4.4.3.1 Equal Cost Coefficients

In this subsection, we consider the case with equal cost coefficients for every constraint sub-formula C_i derived from constraint formula C, i.e. $\delta_1 = \delta_2 \cdots = \delta_n$. That is, the cost for checking each constraint sub-formula is constant.

Theorem 4.1

Let $\delta_1 = \delta_2 \cdots = \delta_n$. Suppose $\sigma_k = 1$ and $\sigma_i = 0$ for $i \neq k$. Then the cost for checking C is equal to $\frac{T_c}{n}$.

Proof:

This is an extreme case that one of the constraint sub-formulas will always be satisfied. Because we assume that the cost for checking the original constraint is T_c and there are n constraint sub-formulas, the cost for checking each constraint sub-formula is $\frac{T_c}{n}$. So, the cost to check the k^{th} constraint sub-formula is also $\frac{T_c}{n}$. ■

Theorem 4.2

Let $\delta_1 = \delta_2 \cdots = \delta_n$. Assume $\sigma_i = \frac{1}{n}$, (i= 1, ..., n). Then the average cost for checking C is $\frac{T_c}{2}$

Proof:

This is the case that each constraint subformula has an equal possibility to be satisfied. Therefore, on average, half of the constraint sub-formulas need to be checked to ensure the validity of the original constraints, i.e., the cost is $\frac{T_c}{2}$.

Formally,

$$
\begin{aligned}
T_c' &\equiv \frac{T_c}{n}\frac{1}{n} + \frac{2T_c}{n}\frac{1}{n} + \cdots + \frac{nT_c}{n}\frac{1}{n} \\
&\equiv \frac{T_c}{n^2}(1+2+3+\ldots+n) \\
&\equiv \frac{T_c}{n}(1+\frac{n)n}{2} \\
&\equiv \frac{T_c}{2}
\end{aligned}
$$

■

Theorem 4.3

Let $\delta_1 = \delta_2 \cdots = \delta_n$. Assume $\sigma_1 = \frac{1}{2^1}$, $\sigma_2 = \frac{1}{2^2}$, ...,$\sigma_{n-1} = \frac{1}{2^{n-1}}$, $\sigma_n = \frac{1}{2^{n-1}}$, then the average time used to check constraint C is equal to $\frac{2T_c}{n}$

Proof:

The average cost of checking the constraint sub-formulas is (for n> 1)

$$T_c'=(\frac{1}{2})\frac{T_c}{n}+(\frac{1}{2^2})\frac{2T_c}{n}+(\frac{1}{2^3})\frac{3T_c}{n}+\cdots+(\frac{1}{2^n})\frac{nT_c}{n}$$

$$=\frac{T_c}{2n}(1+2\frac{1}{2}+3\frac{1}{2^2}+...+n\frac{1}{2^{n-1}})$$

It can be shown that when n is sufficient large, T_c' is equal to $\frac{2T_c}{n}$. ■

4.3.3.2 Exponential Cost Distribution

In this subsection, we consider the case with exponential cost distribution, i.e., $\delta_1=\frac{1}{2^1}, \delta_2=\frac{1}{2^2}, ...,\delta_{n-1}=\frac{1}{2^{n-1}}, \delta_n=\frac{1}{2^{n-1}}$.

Theorem 4.4

Assume $\delta_1=\frac{1}{2^1}, \delta_2=\frac{1}{2^2}, ...,\delta_{n-1}=\frac{1}{2^{n-1}}, \delta_n=\frac{1}{2^{n-1}}$. Suppose $\sigma_k=1$ and $\sigma_i=0$, $1\le i\le n$, $i\neq k$, then the cost for checking C is equal to $\frac{T_c}{2^k}$

Proof: Because the k^{th} constraint sub-formula is always satisfied and the cost coefficient of the k^{th} constraint sub-formula is $\frac{1}{2^k}$, therefore, $T_c'=\frac{T_c}{2^k}$ ■

Theorem 4.5

Assume $\delta_1=\frac{1}{2^1}, \delta_2=\frac{1}{2^2}, ...,\delta_{n-1}=\frac{1}{2^{n-1}}, \delta_n=\frac{1}{2^{n-1}}$. Suppose $\sigma_i=\frac{1}{n}$, (i=1,...,n). The average cost for checking constraint C is equal to $\frac{2T_c}{n}$

Proof:

$$T_c' = \left(\frac{1}{2}\right)\frac{T_c}{n} + \left(\frac{1}{2^2}\right)\frac{2T_c}{n} + \left(\frac{1}{2^3}\right)\frac{3T_c}{n} + \cdots + \left(\frac{1}{2^n}\right)\frac{nT_c}{n}$$

$$= \frac{T_c}{2n}\left(1 + 2\frac{1}{2} + 3\frac{1}{2^2} + \ldots + n\frac{1}{2^{n-1}}\right)$$

It can be shown that when n is sufficiently large, T_c' is equal to $\frac{2T_c}{n}$ ■.

4.4.3 Discussions

In this section, we have analyzed the performance of our proposed constraint enforcement strategy in terms of checking time, especially the improvement of the validation cost with constraint decomposition. The theorems presented in this section make different assumptions on the update patterns and state of database. The result is true only for the given update patterns and state of database. However, the update patterns and run-time characteristics of any database would usually lie in between our assumptions. Thus, with sufficient large number of constraint sub-formulas for a given constraint formula (n>3), the performance improvement using our global optimization technique is significant.

It is also noted that by the way the sufficiency frequency is defined, we make an assumption that constraint sub-formulas of a given constraints can only be satisfied exclusively, i.e., we do not consider the case that multiple constraint sub-formulas can be satisfied simultaneously. This makes the analysis of the performance a little bit easier. It is not hard to observe that in the case of multiple satisfaction of constraint

sub-formulas, the performance will be better.

The results presented in this section are in terms of average cases. We did not consider worst cases because in the worst cases, the cost of enforcing a constraint is simply the cost without decomposition. We did not consider best cases because in the best cases, the cost in enforcing a constraint is minimal.

4.4.4 Validation Cost with Respect to Checking Space

Measuring validation cost in terms of checking space is not as accurate and convenient as in terms of checking time. However, checking space does provide a method to characterize a constraint simplification method. To formalize our analysis and discussion, we need further assumptions and notations.

DEFINITION 4.2 *Dividing Factors and Maximum Dividing Factor*

Suppose that $C(s_1,s_2,\cdots,s_m)$ is decomposed into $C_1(s_{11},s_{12},\cdots,s_{1i_1})$, $C_2(s_{21},s_{22},\cdots,s_{2i_2})$,..., $C_n(s_{n1},s_{n2},\cdots,s_{ni_n})$, such that $C \leftarrow C_1 \vee C_2 \vee \cdots \vee C_n$, then i_j (j=1,2, ..., n) are called *dividing factors* and ω = MAX ($i_1,i_2,\cdots,i_m$) is called the *maximum dividing factor.* ■

From the formal definition of checking space, we can understand the significance of dividing factors and maximum dividing factor. Dividing factors are measurement of the complexities of each individual constraint sub-formula. The smaller the dividing factor, the simpler the given constraint sub-formula.

The maximum dividing factor is a very important parameter in measuring a decomposition method. It is conceivable that the smaller a maximum dividing factor,

the better the decomposition.

It can be seen that dividing factors are static parameters, whereas the cost coefficient and sufficiency frequency are dynamic parameters. Thus, it is conceivable that it is more precise to use cost coefficient and sufficiency frequency to determine the checking order than using static parameters like dividing factors, as a database is an evolving entity. On the other hand, dividing factors characterize a decomposition to some extent. In general, static parameters and dynamic parameters do not conflict, i.e., a decomposition with smaller checking space needs little checking time.

From the static analysis, we need a decomposition such that

$$\sum_{j=1}^{n} S_{j_1} \times S_{j_2} \times \cdots \times S_{j_{i_j}} \ll S_1 \times S_2 \times \cdots \times S_m$$

From the previous examples, we can see that our decomposition meets this requirement.

4.5 Summary

In this chapter, we discussed a constraint enforcement method with the application of decomposition theory developed in previous chapter. We have presented a run-time parameter maintenance method and a global checking procedure. We have analyzed our methods in terms of both checking time and checking space. We discussed the performance in terms of checking time in more details. Several theorems are developed based on different assumptions to show the performance improvement of the proposed method. Because of the locality of database transactions and update patterns, the cost coefficients and sufficiency frequency are more likely exponentially distributed, thus the cost in checking a constraint with decomposition is more close to

$\frac{2T_c}{n}$, where T_c is the cost in checking C without decomposition. We can see that if n>4, then the checking time will be reduced significantly.

Chapter V: Enforcement Strategy : A Parallel Perspective

5.1 Introduction

With the ever decreasing cost of computer hardware, the possibility of implementing a database management function in a multi-processor based system is quite foreseeable. However, our ability to design fast and cheap hardware far outstrips our ability to utilize this hardware effectively to solve large problems quickly [MoI87]. Although early database machine researchers have done a lot of work in exploiting parallelism in query processing, the parallelism in enforcing semantic constraints is totally ignored. They pay special attention to reducing the cost of *hard* database operations, such as *join, division*, or *cartesian product*, when they try to develop database machines. But little work has ever been done to treat integrity constraints in a database machine environment.

The objective of this chapter is to generally treat the problem of integrity constraint enforcement in a database machine environment, and, specifically, the possible enforcement strategy with the application of the constraint decomposition theory developed in chapter 3 in a database machine environment. To achieve our objective, we first discuss the parallelism that can be exploited in a database management system. Second, we will introduce an enhanced version of a database machine with the capability to handle integrity constraints in parallel. Third, we briefly discuss the classification of different kinds of constraints that call for different operation modes of a database machine. Finally, constraint enforcement strategy on a database machine will be discussed and the performance evaluated.

5.2 Parallelism in Database Management System

Database operations and processing are inherently parallel in nature. There is a lot of parallelism that exists in a database management system.

5.2.1 Parallelism in Query Processing

Query processing is the one of the most important components of any database management system. Exploiting parallelism in query processing is a major task facing database machine researchers and developers. The parallelism in query processing includes:

(1) Partitioning the data into different groups which are stored on different storage media that can be accessed independently of one another, and processing the data in different groups in parallel. The query processing procedure of this type of system is as follow:

- The query is analyzed and the request for data is sent to the site where the data involved is located.
- The data is transferred to main memory concurrently.
- The data is processed and the result is sent back to the user.

(2) Concurrent processing of more than one queries. This involves processing multiple queries in a multi-processor based system.

5.2.2 Inter Database Function Parallelism

Parallelism exists among different database management system components.

(1) Parallelism Between Database Update and Integrity Constraint Enforcement

Broadly speaking, there are three principal integrity constraint validation methods, namely, *pre-execution-time, run-time* and *post-execution time.*

With pre-execution time integrity constraint validation, a transaction is allowed to execute only after its integrity constraints are evaluated and all constraints are found true. Its advantage is that it does not require transaction rollback when integrity constraints are violated. Its disadvantage is that the validation and execution are sequential, thus the response is slow.

With run-time integrity constraint validation, integrity constraint validation of transactions are concurrent with transaction execution, where the result of a transaction is not committed. Its advantage is the concurrent execution, so no need for transaction rollback. Its disadvantage is the complexity of the enforcement.

With post-execution-time integrity constraint validation, the transaction is executed first and then the result is validated. Its advantage is its conceptual simplicity, and its disadvantage is the need for a possible transaction rollback.

Among the above mentioned methods, only run-time integrity constraint validation method can achieve the parallel execution of database update and integrity constraint enforcement.

(2) Parallelism between query processing and deferred integrity checking

Database constraints pertaining to one database operation can be validated before the next database operation, which we call immediate checking, or after the next database operation, which we call deferred checking. When multi processors are used, the

validation of one update can be done in parallel with another update or query.

5.2.3 Inter Constraint Validation Parallelism

When a database is updated, a set of constraints related to that update should be validated. The validation of different constraints can be done in parallel if a multi-processor based system is used. If we have an unlimited number of processors, each can validate one constraint. Otherwise, the load balancing of constraints to processors needs to be done to achieve good performance. To get an objective measure of the cost to evaluate each constraint, a cost parameter needs to be defined for each constraint.

5.2.4 Intra Constraint Validation Parallelism

As discussed in previous chapters, a constraint formula C is decomposed into a set of constraint sub-formulas $C_1, C_2, \cdots, C_n$, each of which is a sufficient condition of the original constraint. The validation of $C_1, C_2, \cdots, C_n$ can be done in parallel if a multi-processor based system is used and the number of processors is unlimited. If we have a limited number of processors, we need a load balancing based on the run time parameters defined in the previous chapter. We shall discuss this formally after we introduce a multiprocessor based database machine.

5.3 A Multiprocessor Based Database Machine

In this section, we propose an architecture of a multiprocessor based database machine. We shall describe the major components of the machine as well as the general operation modes of the machine.

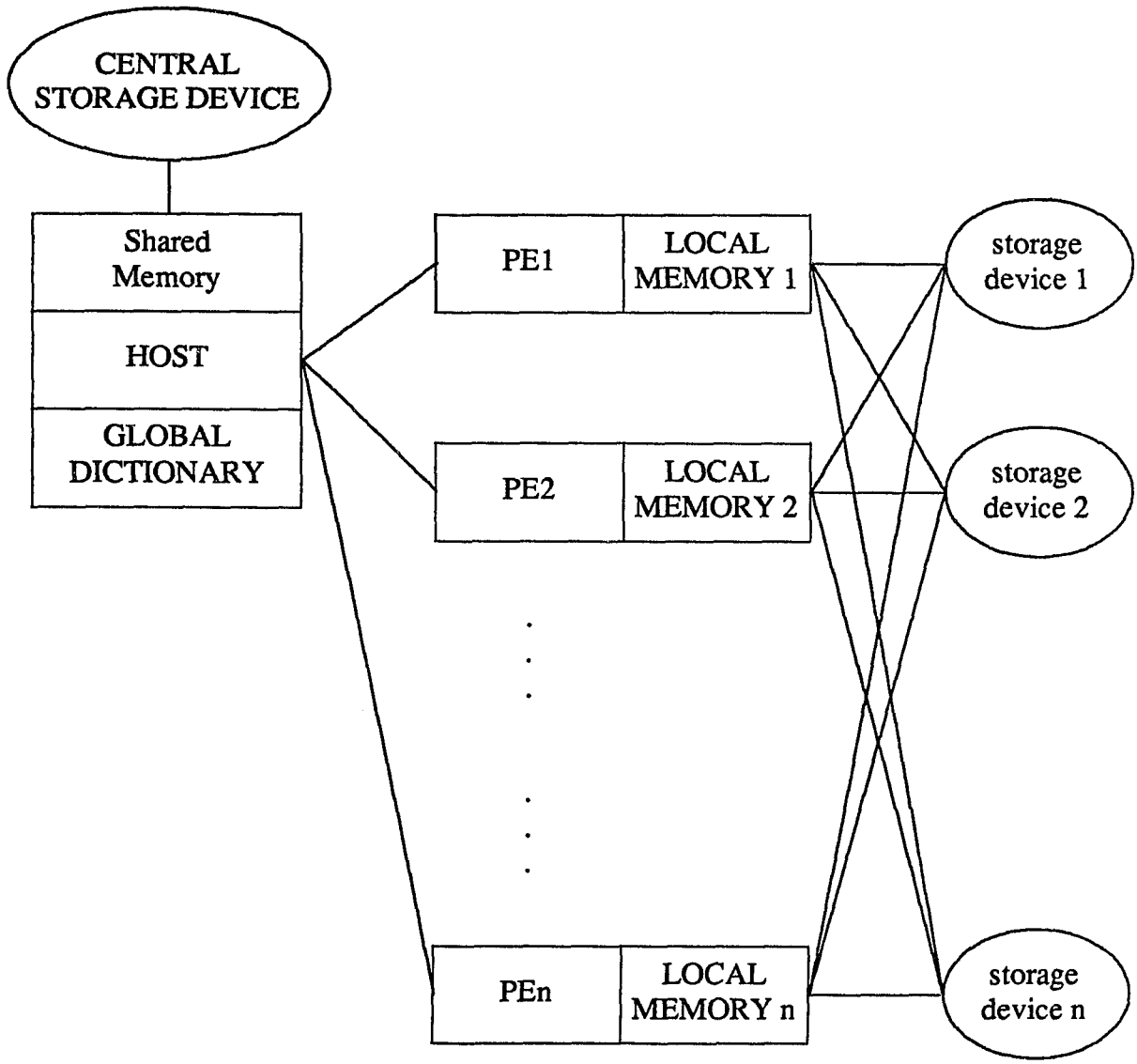

Figure 5.1. A Database Machine

5.3.1 General Description

Figure 5.1 shows a general configuration of a multiprocessor based system. The machine can either operate in a MIMD mode or a SIMD mode [IyS87], depending upon the requirement of the database operations. The discussion of each operation

mode will be presented in later sections. In the following, we shall briefly describe each component of this machine.

• **Host:** *Host* is a general purpose computer which runs the DBMS, has the ability to coordinate the activities of the processing elements (PE) and distributes the data among the storage devices. It maintains a global data dictionary which stores the following information:

(1) The information on how the data should be distributed.

(2) The value of the run-time parameters for each constraint sub-formula.

(3) Cost parameters for each constraint.

The host can be programmed to change operation mode depending on the type of operation concerned. It can request the PEi's to do the same operation on different data, or it can request the PEi's to do different operations on different data.

• **Central Storage Device:** The *Host* assigns a fixed amount of space from the main device to each relation that is initially created. The idea is that when the number of bytes of a relation does not exceed the pre_defined amount of space assigned to the given relation from the *Central Storage Device*, there is no need to distribute the data among the *Storage devices*, which saves the communication and synchronization costs.

• **Storage device 0..n-1:** When the number of bytes of a relation exceeds the pre-defined amount of space in the main storage device, the data should be distributed among the storage devices in such a way that maximum parallelism is achieved while the synchronization and communication costs are kept to a minimum. The distribution

algorithm is to be described in the next section.

• **PEi's:** *Processing Element i* is attached to storage device i to process the information stored there. It can perform basic DBMS functions, such as *Selection, Deletion* and *Update* and has integrity checking functions as well. It has the ability to communicate with the host, but not necessarily with other PEi's. The PEi's can also access any storage device to retrieve the required information stored there.

• **Shared Memory:** *Shared Memory* is the main memory space for the PEi's to communicate and synchronize with the Host, and to record the results of the operations.

• **Global Data Dictionary:** *Global Data Dictionary* is used to record the following information:

(1) Database catalog: name of relations; number of tuples in the relation, etc.

(2) Data location information: where the data are located and the number of tuples in each location.

(3) Run-time parameter value for each constraint subformula

(4) Cost parameters for each constraint formula, if any.

• **Local Data dictionary:** The *Local Data Dictionary* is used by the PEi to decide whether it needs to respond to a request from the host. It records information on the data fragment stored there.

5.3.2 Data Distribution

In this section, we describe the data distribution in the proposed database machine. Whenever data are to be inserted into the database, the algorithm must

decide how and where the data are to be stored. We first examine the data structure used by the algorithm, namely, the *Global Data Dictionary*. Then, we present a formal description of the algorithm.

Global Data Dictionary

As pointed out earlier, the global data dictionary stores the information on how and where the data are to be distributed among the storage devices.

The following data structures need to be included in the global dictionary.

```
CONST MAX_RELATION /* defines the maximum number of relations */

ENTRY=record
        LENGTH: INTEGER;
        INDEX :    0..N-1
      end

VAR GLOBAL_DATA_DICTIONARY: array[1..MAX_RELATION] of ENTRY
```

where N is the number of processing elements or storage devices.

For each relation created, an entry is set up as shown above. The variable LENGTH records the number of bytes the given relation has and INDEX indicates onto which storage device the next tuple is to be inserted.

The Distribution Algorithm

The distribution algorithm is used to decide onto which storage device the next tuple is to be inserted when a user wants to add a tuple to a relation. With the information in the Global Data Dictionary, the data distribution process is straightforward. The main criterion is to distribute the data evenly among the storage devices, to achieve the maximum parallelism. The algorithm first examines whether the length of

the relation exceeds the pre-defined amount of space in Central Storage Device after the given tuple is inserted. If not, then it inserts the given tuple onto the Central Storage Device. Otherwise, it inserts it into one of the Storage Devices based on the information in the Global Data Dictionary. The algorithm is as follows.

Algorithm 5.11: The Data Distribution Algorithm

```
Input:    T /* T is the tuple to be inserted        */
          R /* R is the relation into which         */
            /* T is to be inserted                  */

Output:   Status /* Status=TRUE if T is successfully inserted      */
              /*          into one of the fragment of R          */
              /* Status=FALSE if the insertion of T violates      */
              /*          some integrity constraints             */

Procedure

  Begin
      call constraint_checking
      If not satisfied
      Then return FALSE
      Else
         Begin
            R.LENGTH:=R.LENGTH + length(T)
            If R.LENGTH<= ζ
           Then
              insert it onto main device
           Else
             Begin
                insert T onto storage device
                indicated by R.index
                INDEX:=(INDEX+1) mod N
             End
         End
       return TRUE
  End
```

Figure 5.2 A Data Distribution Algorithm.

Note that "length" is the function which takes a tuple as the input and returns the length of the tuple as the output. ζ is a constant which determines the amount of memory space of main storage allocated to each relation.

It may be noted that the above distribution algorithm is straightforward in the sense that its goal is to distribute data evenly among the storage devices to achieve maximum parallelism. For other methods of data distribution in a distributed database environment, see [Dat86, Ull88].

5.4 Data Partition and Task Partition

For a problem to be processed effectively by a multiprocessor based system, it must meet one of the following requirements:

[1] *Task Partitionable*: This means that the task can be divided into a set of smaller and simpler sub-tasks, each being processed by a different processing element, or

[2] *Data Partitionable*: This means that the data the algorithm works on can be partitioned into a set of smaller groups, each being processed by a different processing element running the same algorithm.

The enforcement of most constraint formulas meets the above requirements. Functional dependency constraints and referential constraints are examples of data partionable constraints. The data partitionable constraints can be validated on our proposed machine in SIMD mode. The constraints we discussed in Chapter 3 are all task partitionable, because each constraint sub-formula can be enforced independently on a different processing element. The task partitionable constraints can be validated on our machine in MIMD mode. SIMD and MIMD mode operations for our proposed machine will be discussed in the following sections.

5.5 SIMD Mode Operation

In this section, we discuss two expensive integrity constraint enforcement in SIMD mode of our proposed database machine. The integrity to be enforced are functional dependency and referential dependency.

5.5.1 The Functional Dependency Enforcement

5.5.1.1 Functional Dependency

Definition 5.1 : (*Functional Dependency*)

Let $R(A_1,A_2,\cdots,A_n)$ be a relation scheme, and let X and Y be subsets of $\{A_1,A_2,\cdots,A_n\}$. We say X→Y , read "X functionally determine Y" or "Y functionally depends on X" if whatever relation r is the current value for R, it is not possible that r has two tuples that agree in the components for all attributes in the set X yet disagree in one or more components for attributes in the set Y.■

Functional dependency is a very important constraint on a database and many design theories make use of functional dependency. However, the enforcement of functional dependency is very expensive, as we can see from the following discussion.

5.5.1.2 Checking of Functional Dependency

When a tuple is to be inserted or updated, it may be possible that it violates functional dependency constraints [Ull82]. The enforcement of functional dependency is an expensive one, because the system has to examine every tuples in the relation. To make the checking efficient, we may set up an index on the determinant fields for each functional dependency constraint. This will require additional space, occasionally

greater than the space occupied by the relation itself. But in this architecture, the index is not necessary. In the following, we first describe the data structures used for this algorithm and then the algorithm itself is presented and discussed.

Data Structure

The following data structures are used in the Functional Dependency Checking Algorithm.

STATUS: ARRAY[1..N] of BOOLEAN

The array STATUS[1..N] is used to record the result of checking done by PEi.

COUNT: Integer

COUNT is used for synchronization between the Processing Elements and the Host.

The Algorithm

The following is a parallel algorithm for the checking of functional dependency constraints.

Algorithm 5.2 Functional Dependency Checking
Input: T: the tuple to be inserted
R: relation name
Rule_id: The id of the integrity constraint

Output: A status to indicate result of checking

```
Procedure Central_FD
  Begin
    If R.LENGTH+length(T)<= ξ
       Then
         Do the checking in Main device
       Else Begin
                 Count:=0;
                 Broadcast T,R to PEi's;
                 Send Checking Instruction and the Integrity Rule to PEi's;
              End
    While Count<n Do
            wait;
    Return(And STATUS[i])
  End


Procedure Local_FD
  Begin
    Do the local checking with
    the local fragment of R
    If satisfied
       Then
          statusi=true
       Else
          statusi=false
       Lock(STATUS)
         STATUS[i]=statusi
       Unlock(STATUS)
       Lock(count)
         count=count+1
       Unlock(count)
  End
```

Figure 5.3. A Parallel FD Checking Algorithm.

The Central_FD works as follows: If after the tuple has been inserted, the total amount of memory space occupied by this relation is less than or equal to the pre-defined memory space allocated to this relation in central storage device, then the integrity checking will be done in central storage device only. Otherwise, the host broadcasts T and R to the PEi's so that the PEi's can check the constraint concurrently. Synchronization is done by the following statement:

```
While COUNT<n Do
    wait
```

After all PEi's have finished their checking, Central_FD returns the result of a logical AND of each STATUS[i].

The Local_FD works as follows: it modifies STATUS[i] based on local checking. Then, it increments the global variable COUNT to signal the completion of the local checking.

5.5.2 Referential Integrity Checking

5.4.2.1 Referential Integrity

Definition 5.2:

Let R[A, X], S[A,Y] be relation schemes, where A is a primary key of relation S, and X and Y are sets of attributes. The referential integrity rule states that every value of attribute A in R must either (a) be equal to the value of the primary key A in some tuple in S or (b) be wholly null. R and S are not necessarily distinct. ■

Referential integrity is a very important constraint on a database. It reflects a

important relationship between two relations of a database.

5.5.2.2 Checking of Referential Integrity

When a tuple is to be inserted or deleted from a relation, it may be possible that a referential integrity constraint is violated. The checking of referential integrity constraint is as expensive as that of functional dependency, because the system needs to check every tuple of the involved relations.

As an example, consider the two relations

SP(S#,P#,QTY) and
S(S#,SNAME,SSTATUS,CITY)

where S is the relation scheme for supplier information and SP is a relation scheme for recording the information on which supplier(S#) supplies how much(QTY) of which part(P#). The integrity constraint "Any S# value appearing in relation SP must also appear in relation S" is the referential constraint to reflect the fact an entity that supplies parts must be a real supplier.

The data structure used in referential integrity checking is the same as those involved in functional dependency checking.

The algorithm for referential integrity constraint checking is as follows.

Algorithm 5.3 Referential Integrity Checking
INPUT: Rule_id The ID of the Integrity rule
T The tuple to be inserted
R The relation name into which R is to be inserted

OUTPUT: A status indicating the result of checking

```
Procedure Central_RI
 Begin
  If R.LENGTH + length(T)<= ζ
    Then
      Do the referential checking in main device
    Else Begin
              Count:=0;
              Broadcast T,R and Rule_id to PEi's;
              Send Checking Instruction to PEi's
            End
  While Count<n Do
        wait;
  return(OR status[i] ,i=0,n-1)
 End


Procedure Local_RI
 Begin
  Do the local referential checking
  with the local fragment of data
  If satisfied
    Then
         statusi=true
    Else
         statusi=false
  lock(STATUS);
    STATUS[i]=statusi
  unlock(STATUS);
  lock(COUNT);
    COUNT=COUNT+1
  unlock(COUNT);
End
```

Figure 5.4. A Parallel RI Checking Algorithm

It may be noted that the algorithm for referential integrity checking is identical as that for functional dependency, except that in the last step of CENTRAL_RI, it returns the result of a logical OR of each STATUS[i], instead of a logical AND.

5.5.3 Performance

There are three sources of cost associated with enforcement of an integrity constraints in a multiprocessor based system, namely,

(1) Communication and synchronization cost.

(2) Computation done on the CPU.

(3) Secondary storage access time.

Secondary storage access time is the major cost in enforcement of an integrity constraint. If the database machine is operated in SIMD mode, the synchronization and communication costs can be minimized. Also, computations on the CPU are just comparisons, therefore they do not take much time.

With the data evenly distributed over storage devices, maximum parallelism is achieved. Therefore, if the cost to check the constraint is T_c in a uniprocessor system, then the cost to check the given constraint in a multiprocessor based system with n processing elements is virtually T_c/n.

5.6 MIMD Mode Operation

In this section, we discuss integrity constraint enforcement strategy in our proposed machine in MIMD mode. In MIMD mode, each processing element executes different instruction on different data items. When operated in MIMD mode, each pro-

cessing element can validate an individual constraint in parallel with other processing elements validating other constraints, or each processing elements can validate an individual constraint sub-formula in parallel with other processing element validating other constraint sub-formula. The process of the validation will be discussed in this section.

5.6.1 Notations

In the previous chapters, we have introduced the sufficiency frequency and cost coefficient as a run-time parameters to monitor database evolution and change. Those are dynamic parameters, as their values depend on the current state of database. Moreover, those parameter are associated with constraint sub-formulas, not with the constraint itself. We also defined a static parameter, called checking space, which can measure the relative complexity of constraint formulas. Both the cost coefficient and checking space can measure the cost to check a constraint sub-formula, but they have some differences. Using checking space will save some computation time, but it fails to consider the physical characteristics of a database. Using the cost coefficient is more accurate in measuring the cost of checking a constraint formula, but it does require extra computation.

The following definition defines the cost factor of a constraint:

DEFINITION 5.3 *Cost Factor*

A cost factor of a constraint formula $C(S_1,S_2,\cdots S_m)$, denoted as $\lambda(C)$, is defined as $|S_1|\times|S_2|\times...\times|S_m|$, where $S_1,S_2,\cdots,S_m$ are tuple variables.■

Since $|S_1|\times|S_2|\times...\times|S_m| = |S_1\times S_2\times...\times S_m|$, the cost factor is actually the cardinality of the checking space, i.e., the number of rows in the checking space. Because it is a number, it is very easy for a system to compare and measure the complexity of a constraint.

5.6.2 Parallel Checking at Constraint Level

Suppose that we have n processing elements and m affected constraints after a database update. If m ≤ n, i.e., we have more processors than constraint sub-formulas, then each processing element checks one constraint. If any violation is detected, we reject the update; otherwise, we commit the update. If n<m, then we divide the constraints into n different groups, each processing elements validating constraints within a given group. The criterion to assign constraints to each group is to maximally exploit the parallelism. In the following, we treat this problem more formally.

Definition 5.4

Let $C_1, C_2, \ldots, C_m$ be m constraint formulas and $\lambda_1, \lambda_2, \ldots, \lambda_m$ be the cost factors of $C_1, C_2, \ldots, C_m$ respectively. If we divide C_i (i=1,2, ... , m) into n different group $\{C_1^{1p}, C_2^{1p}, \ldots, C_{t_{1p}}^{1p}\}$, $\{C_1^{2p}, C_2^{2p}, \ldots, C_{t_{2p}}^{2p}\}$, ... , $\{C_1^{np}, C_2^{np}, \ldots, C_{t_{np}}^{np}\}$ $(t_{1p} + t_{2p} + \ldots + t_{np} = m)$, then,

(1) $\rho_p = \{\{C_1^{1p}, C_2^{1p}, \ldots, C_{t_1}^{1p}\}, \quad \{C_1^{2p}, C_2^{2p}, \ldots, C_{t_2}^{2p}\}$, ... , $\{C_1^{np}, C_2^{np}, \ldots, C_{t_{np}}^{np}\}\}$ is called an *assignment* ρ_p.

(2) $\overline{\upsilon_p} = \sum_{i=1}^{m} \frac{\lambda_i}{n}$ is called the *average group sum of assignment* ρ_p.

(3) $\upsilon_{ip} = \sum_{j=1}^{t_{ip}} \lambda_j^{ip}$ is called the *the sum of group i of assignment* ρ_p.

(4) $\tau_p = \sum_{i=1}^{n} |\upsilon_{ip} - \bar{\upsilon}|$ is called the *sum of difference of assignment* ρ_p.

(5) $\tau_{\min} =$ MIN $\{(\sum_{k=1}^{n} |\upsilon_{kr} - \bar{\upsilon}|) \mid$ assignment ρ_r range from any possible assignment$\}$ is called *minimum sum of difference.* ■

With the above definitions, we can now introduce an assignment theorem.

Theorem 5.1 (*Assignment Theorem*)

Let $C_1, C_2, \ldots, C_m$ be m constraint formulas to be validated when a database update occurs, and $\lambda_1, \lambda_2, \ldots, \lambda_m$ be the cost factors of $C_1, C_2, \ldots, C_m$, respectively. If we have an assignment

$$\rho_q = \{ \{C_1^{1q}, C_2^{1q}, \ldots, C_{t_{1q}}^{1q}\}, \{C_1^{2q}, C_2^{2q}, \ldots, C_{t_{2q}}^{2q}\}, \cdots, \{C_1^{nq}, C_2^{nq}, \ldots, C_{t_{nq}}^{nq}\} \}$$

with $\bar{\upsilon} = \sum_{i=1}^{m} \frac{\lambda_i}{n}$, and $\upsilon_{iq} = \sum_{j=1}^{t_{iq}} \lambda_j^{ip}$, and

$$\tau_q = MIN\{(\sum_{i=1}^{n} |\upsilon_{ir} - \bar{\upsilon}|) \mid \rho_r \, range \, over \, all \, \, assignments\} \qquad (5.1)$$

then the assignment ρ_q is optimal in the sense that maximal parallelism is achieved.

Proof:

Without loss of generality, let $\upsilon_{1q} \geq \upsilon_{2q} \geq \cdots \geq \upsilon_{nq}$. To prove optimality, we only need to prove that for any other assignment

$$\rho_p = \{ \{C_1^{1p}, C_2^{1p}, \ldots, C_{t_{1p}}^{1p}\}, \{C_1^{2p}, C_2^{2p}, \ldots, C_{t_{2p}}^{2p}\}, \cdots, \{C_1^{np}, C_2^{np}, \ldots, C_{t_{np}}^{np}\} \}$$

If $\tau_p > MIN(\sum_{i=1}^{n} |\upsilon_{ir} - \bar{\upsilon}|)$ and $\upsilon_{1p} \geq \upsilon_{2p} \geq \cdots \geq \upsilon_{np}$, then $\upsilon_{1p} > \upsilon_{1q}$.

Suppose $\upsilon_{1q} > \upsilon_{1p}$, then t_1 is greater than one. move a constraint C_1^{jq} to group n such that $\upsilon_{1q} > \upsilon_{1q}^{j} + \upsilon_{nq}$. Consider the following four cases.

Case I:

Suppose that

$$\upsilon_{1q} - \upsilon_{1q}^{j} > \bar{\upsilon}$$
$$\upsilon_{nq} + \upsilon_{1q}^{j} < \bar{\upsilon}$$

After rearrangement, the sum of difference

$$\begin{aligned}\tau_q' &= |\upsilon_{1q} - \upsilon_{1q}^{j} - \bar{\upsilon}| + \sum_{i=2}^{n-1} |\upsilon_{iq} - \bar{\upsilon}| + |\upsilon_{nq} + \upsilon_{1q}^{j} - \bar{\upsilon}| \\ &= \upsilon_{1q} - \upsilon_{1q}^{j} - \bar{\upsilon} + \sum_{i=2}^{n-1} |\upsilon_{iq} - \bar{\upsilon}| - \upsilon_{nq} - \upsilon_{1q}^{j} + \bar{\upsilon} \\ &= \upsilon_{1q} - 2\upsilon_{1q}^{j} - \upsilon_{nq} + \sum_{i=2}^{n-1} |\upsilon_{iq} - \bar{\upsilon}|\end{aligned}$$

Before the rearrangement, the sum of difference is

$$\begin{aligned}\tau_q &= |\upsilon_{1q} - \bar{\upsilon}| + \sum_{i=2}^{n-1} |\upsilon_{iq} - \bar{\upsilon}| + |\upsilon_{nq} - \bar{\upsilon}| \\ &= \upsilon_{1q} - \bar{\upsilon} + \sum_{i=2}^{n-1} |\upsilon_{iq} - \bar{\upsilon}| - \upsilon_{nq} + \bar{\upsilon} \\ &= \upsilon_{1q} - \upsilon_{nq} + \sum_{i=2}^{n-1} |\upsilon_{iq} - \bar{\upsilon}|\end{aligned}$$

It is easy to note that $\tau_q' < \tau_q$, that is, the assumption that the assignment ρ_q is minimal is not true, so we have a contradiction.

Case II:

Suppose that

$$\upsilon_{1q} - \upsilon_{1q}^{j} < \bar{\upsilon}$$
$$\upsilon_{nq} + \upsilon_{1q}^{j} < \bar{\upsilon}$$

After rearrangement, the sum of difference

$$\tau_q' = |\upsilon_{1q} - \upsilon_{1q}^j - \bar{\upsilon}| + \sum_{i=2}^{n-1} |\upsilon_{iq} - \bar{\upsilon}| + |\upsilon_{nq} + \upsilon_{1q}^j - \bar{\upsilon}|$$

$$= -\upsilon_{1q} + \upsilon_{1q}^j + \bar{\upsilon} + \sum_{i=2}^{n-1} |\upsilon_{iq} - \bar{\upsilon}| - \upsilon_{nq} - \upsilon_{1q}^j + \bar{\upsilon}$$

$$= 2\bar{\upsilon} - \upsilon_{1q} - \upsilon_{nq} + \sum_{i=2}^{n-1} |\upsilon_{iq} - \bar{\upsilon}|$$

Before the rearrangement, the sum of difference is

$$\tau_q = |\upsilon_{1q} - \bar{\upsilon}| + \sum_{i=2}^{n-1} |\upsilon_{iq} - \bar{\upsilon}| + |\upsilon_{nq} - \bar{\upsilon}|$$

$$= \upsilon_{1q} - \bar{\upsilon} + \sum_{i=2}^{n-1} |\upsilon_{iq} - \bar{\upsilon}| - \upsilon_{nq} + \bar{\upsilon}$$

$$= \upsilon_{1q} - \upsilon_{nq} + \sum_{i=2}^{n-1} |\upsilon_{iq} - \bar{\upsilon}|$$

Because

$$\upsilon_{1q} > \bar{\upsilon} \tag{5.2}$$

we have

$$2\upsilon_{1q} > 2\bar{\upsilon} \tag{5.3}$$

By adding $-\upsilon_{1q} - \upsilon_{nq}$ to both sides of equation (5.3) we have

$$\upsilon_{1q} - \upsilon_{nq} > 2\bar{\upsilon} - \upsilon_{1q} - \upsilon_{nq} \tag{5.4}$$

Comparing τ_q , τ_q' and equation (5.4), it is easy to note that $\tau_q' < \tau_q$, that is, the assumption that the assignment ρ_q is minimal is not true. Again, we have a contradiction.

Case III:

Suppose that

$$\upsilon_{1q}-\upsilon_{1q}^{j}>\bar{\upsilon}$$
$$\upsilon_{nq}+\upsilon_{1q}^{j}>\bar{\upsilon}$$

After rearrangement, the sum of difference

$$\tau_q'=|\upsilon_{1q}-\upsilon_{1q}^{j}-\bar{\upsilon}|+\sum_{i=2}^{n-1}|\upsilon_{iq}-\bar{\upsilon}|+|\upsilon_{nq}+\upsilon_{1q}^{j}-\bar{\upsilon}|$$
$$=\upsilon_{1q}-\upsilon_{1q}^{j}-\bar{\upsilon}+\sum_{i=2}^{n-1}|\upsilon_{iq}-\bar{\upsilon}|+\upsilon_{nq}+\upsilon_{1q}^{j}-\bar{\upsilon}$$
$$=-2\bar{\upsilon}+\upsilon_{1q}+\upsilon_{nq}+\sum_{i=2}^{n-1}|\upsilon_{iq}-\bar{\upsilon}|$$

Before the rearrangement, the sum of difference is

$$\tau_q=|\upsilon_{1q}-\bar{\upsilon}|+\sum_{i=2}^{n-1}|\upsilon_{iq}-\bar{\upsilon}|+|\upsilon_{nq}-\bar{\upsilon}|$$
$$=\upsilon_{1q}-\bar{\upsilon}+\sum_{i=2}^{n-1}|\upsilon_{iq}-\bar{\upsilon}|-\upsilon_{nq}+\bar{\upsilon}$$
$$=\upsilon_{1q}-\upsilon_{nq}+\sum_{i=2}^{n-1}|\upsilon_{iq}-\bar{\upsilon}|$$

Because

$$\upsilon_{nq}<\bar{\upsilon} \tag{5.5}$$

we have

$$-2\upsilon_{nq}>-2\bar{\upsilon} \tag{5.6}$$

By adding $\upsilon_{1q}+\upsilon_{nq}$ to both sides of equation (5.6) we have

$$\upsilon_{1q}-\upsilon_{nq}>-2\bar{\upsilon}+\upsilon_{1q}+\upsilon_{nq} \tag{5.7}$$

Comparing τ_q , τ_q' and equation (5.7), it is easy to note that $\tau_q'<\tau_q$, that is, the assumption that the assignment ρ_q is minimal is not true. Thus, we again have a contradiction.

Case IV:

Suppose that

$$\upsilon_{1q}-\upsilon'_{q}<\bar{\upsilon}$$
$$\upsilon_{nq}+\upsilon'_{q}>\bar{\upsilon}$$

After rearrangement, the sum of differences is

$$\begin{aligned}\tau'_q&=|\upsilon_{1q}-\upsilon'_{q}-\bar{\upsilon}|+\sum_{i=2}^{n-1}|\upsilon_{iq}-\bar{\upsilon}|+|\upsilon_{nq}+\upsilon'_{q}-\bar{\upsilon}|\\&=-\upsilon_{1q}+\upsilon'_{q}+\bar{\upsilon}+\sum_{i=2}^{n-1}|\upsilon_{iq}-\bar{\upsilon}|+\upsilon_{nq}+\upsilon'_{q}-\bar{\upsilon}\\&=2\upsilon'_{q}-\upsilon_{1q}+\upsilon_{nq}+\sum_{i=2}^{n-1}|\upsilon_{iq}-\bar{\upsilon}|\end{aligned}$$

Before the rearrangement, the sum of differences is

$$\begin{aligned}\tau_q&=|\upsilon_{1q}-\bar{\upsilon}|+\sum_{i=2}^{n-1}|\upsilon_{iq}-\bar{\upsilon}|+|\upsilon_{nq}-\bar{\upsilon}|\\&=\upsilon_{1q}-\bar{\upsilon}+\sum_{i=2}^{n-1}|\upsilon_{iq}-\bar{\upsilon}|-\upsilon_{nq}+\bar{\upsilon}\\&=\upsilon_{1q}-\upsilon_{nq}+\sum_{i=2}^{n-1}|\upsilon_{iq}-\bar{\upsilon}|\end{aligned}$$

Because

$$\upsilon_{1q}>\upsilon'_{q}+\upsilon_{nq} \tag{5.8}$$

By moving υ_{nq} to the left-hand side and multiplying two sides of equation (5.8) by 2, we have

$$2\upsilon_{1q}-2\upsilon_{nq}>2\upsilon'_{q} \tag{5.9}$$

By adding $-\upsilon_{1q}+\upsilon_{nq}$ to both sides of equation (5.9) we have,

$$\upsilon_{1q}-\upsilon_{nq}>2\upsilon'_{q}-\upsilon_{1q}+\upsilon_{nq} \tag{5.10}$$

Comparing τ_q , τ'_q and equation (5.10), it is easy to note that $\tau'_q<\tau_q$, that is, the assumption that the assignment ρ_q is minimal is not true, yielding yet another contradiction.

Because the above four cases are the only possible cases, the proof is complete. ■

Theorem 5.1 gives a theoretical basis for optimally assigning constraints to processing elements to validate. We next introduce an effective procedure to assign constraints to processing elements in the hope to maximally exploit parallelism.

Algorithm 5.4 (*Assignment Algorithm, Version A.*)

Input: A set of constraint formulas
$\{C_1/\lambda_1 , C_2/\lambda_2 , \ldots ,C_m/\lambda_m \}$

Output: A set of subsets of constraint formulas
$\{T_1 , T_2 , \ldots ,T_n\}$
$T_i = \{C_{1_i}/\lambda_1^i , \ldots ,C_{t_i}^i /\lambda_{t_i}^i\}$

Step 1: Sort $\{C_1/\lambda_1 , C_2/\lambda_2 , \ldots ,C_m/\lambda_m \}$ in
descending order based on the value of λ_i.
Without loss of generality, assume $\lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_m$

Step 2: $T_i = \phi$; $\upsilon_i = 0$ (i=1, 2, ... ,n), j=0

Step 3: If j+i ≤ m then $T_i = T_i \cup \{C_{j+i}/\lambda_{j+i}\}$ (i=1,2, ..., n);
$\upsilon_i = \upsilon_i + \lambda_{j+1}$

Step 4: j=j+n

Step 5: Sort $\{T_1, T_2, \cdots ,T_n\}$ in ascending order
based on the partial group sum υ_i.
Assume $\upsilon_1 \leq \upsilon_2 \leq \cdots \leq \upsilon_n$

Step 6: Go to step 3. ■

Figure 5.5. A Constraint Assignment Algorithm (Version A)

It can be noted that algorithm 5.3 assigns constraint formulas to processing elements as evenly as possible. The following example illustrates how the algorithm works.

The time complexity of algorithm 5.4 is O(m*log*m +(m/n)n*log*n). Because the cost factor is a static parameter, sorting constraint formulas are done only once. So,the real complexity is O(m*log*n).

Example 5.1

Let C= { $C_1/3$, $C_2/4$, $C_3/7$, $C_4/16$, $C_5/2$, $C_6/17$}

After step 1 , we have

$C' = \{C_6/17 , C_4/16 , C_3/7 , C_2/4 , C_1/3 , C_5/2\}$

After step 2, we have

$T_i = 0$, $\upsilon_i = 0$, j=0, for i=1,2,...,n.

After step 3 (j=0, loop one), we have

$T_1 = \{C_6/17\}$, $T_2 = \{C_4/16\}$

$\upsilon_1 = 17$, $\upsilon_2 = 16$

After step 4 (loop one), j=2

After step 5 (loop one), we have

$\{T_2/16 , T_1/17\}$

After step 3 (j=2, loop 2), we have

$T_2 = \{C_4/16 , C_3/7\}$,

$T_1 = \{C_6/17 , C_2/4\}$

υ_2=23,υ_1=21

After step 4 (loop two), j=4

After step 5 (loop two), we have

$\{T_1/21 , T_2/23\}$

After step 3 (loop three, j=4), we have

$T_1 = \{C_6/17 , C_2/4 , C_1/3\}$,

$T_2 = \{C_4/16 , C_3/7 , C_5/2\}$

$\upsilon_1 = 24, \upsilon_2 = 25$

■

Algorithm 5.4 provides us with a procedure to assign constraints to different groups. The resulting assignment is intuitively good or near-optimal. From the theorem, we know that the computation to get an optimal assignment is exponential, because every possible assignments has to be examined. This involves the study of every combination of constraint formulas.

The trade off is the extent of the optimality of a solution and the time spent in searching that solution. The fact is that because we need to carry out this task each time a database update occurs and the solution just found is only used once, the time spent in finding the optimal solution overpowers the optimality of the solution. Generally speaking, it is not worthwhile to spend exponential time to search for a solution which will be used only once.

To make a comparision, we present another version of the assignment algorithm that takes less time to finish. But sometime the result generated by this algorithm is not as good as that of algorithm 5.4.

Algorithm 5.5 (*Assignment Algorithm, Version B.*)

Input: A set of constraint formulas
$\{C_1/\lambda_1 , C_2/\lambda_2 , \ldots ,C_m/\lambda_m \}$

Output: A set of subsets of constraint formulas
$\{T_1 , T_2 , \ldots ,T_n \}$
$T_i = \{C_{1_i}/\lambda_1^i , \ldots ,C_{t_i}^i /\lambda_{t_i}^i\}$

Step 1: Sort $\{C_1/\lambda_1 , C_2/\lambda_2 , \ldots ,C_m/\lambda_m \}$ in
descending order based on the value of λ_i.
Without loss of generality, assume $\lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_m$

Step 2: $T_i = \phi$; $\upsilon_i = 0$ (i=1, 2, ... ,n), j=0

Step 3: If j+i ≤ m then $T_i = T_i \cup \{C_{j+i}/\lambda_{j+i}\}$ (i=1,2, ..., n);
$\upsilon_i = \upsilon_i + \lambda_{j+1}$

Step 4: j=j+n

Step 5: Reverse the ordering of T_i

Step 6: Go to step 3. ■

Figure 5.6. A Constraint Assignment Algorithm (Version B)

Note that the only difference between algorithm 5.4 and 5.5 is in step 5. With the aid of a counter or a flag, reversing the ordering of T_i takes constant time, whereas sorting T_i takes n*log*n time. It is easy to observe that the time complexity of algorithm 5.5 is O(m*log*m + m/n). Similarly, because we do not need to sort constraints each

time we invoke this algorithm, the real complexity is O(m/n).

For the constraint sub-formulas presented in example 5.1, algorithm 5.5 will generate the same result as generated by algorithm 5.4. However, on average algorithm 5.4 will generate a better result for more time.

Example 5.2

Let C= { $C_1/3$, $C_2/6$, $C_3/7$, $C_4/16$, $C_5/2$, $C_6/20$}

The following are the steps taken by algorithm 5.4 to generate an assignment.

After step 1 , we have

$C' = \{C_6/20 , C_4/16 , C_3/7 , C_2/6 , C_1/3 , C_5/2\}$

After step 2, we have

$T_i = 0$, $\upsilon_i = 0$, j=0, for i=1,2,...,n.

After step 3 (j=0, loop one), we have

$T_1 = \{C_6/20\}$, $T_2 = \{C_4/16\}$

$\upsilon_1 = 20$, $\upsilon_2 = 16$

After step 4 (loop one), j=2

After step 5 (loop one), we have

$\{T_2/16 , T_1/20\}$

After step 3 (j=2, loop 2), we have

$T_2 = \{C_4/16 , C_3/7\}$,

$T_1 = \{C_6/20 , C_2/6\}$

υ_2=23,υ_1=26

After step 4 (loop two), j=4

After step 5 (loop two), we have

$\{T_2/23 , T_1/26\}$

After step 3 (loop three, j=4), we have

$T_1 = \{C_6/20 , C_2/6 , C_5/2\}$,

$T_2 = \{C_4/16 , C_3/7 , C_1/3\}$

$\upsilon_1 = 28, \upsilon_2 = 26$

The following are steps taken by algorithm 5.5 to generate an assignment.

After step 1 , we have

$C' = \{C_6/20 , C_4/16 , C_3/7 , C_2/6 , C_1/3 , C_5/2\}$

After step 2, we have

$T_i = 0 , \upsilon_i = 0 , j$=0, for i=1,2,...,n.

After step 3 (j=0, loop one), we have

$T_1 = \{C_6/20\} , T_2 = \{C_4/16\}$

$\upsilon_1 = 20, \upsilon_2 = 16$

After step 4 (loop one), j=2

After step 5 (loop one), we have

$\{T_2/16 , T_1/20\}$

After step 3 (j=2, loop 2), we have

$T_2 = \{C_4/16 , C_3/7\}$,

$T_1 = \{C_6/20 , C_2/6\}$

υ_2=23,υ_1=26

After step 4 (loop two), j=4

After step 5 (loop two), we have

$\{T_1/26 , T_2/23\}$

After step 3 (loop three, j=4), we have

$T_1 = \{C_6/20 , C_2/6 , C_1/3\}$,

$T_2 = \{C_4/16 , C_3/7 , C_5/2\}$

$\upsilon_1 = 29, \upsilon_2 = 25$

It can be seen that, for this example, algorithm 5.4 performs better than algorithm 5.5. ■.

The behaviors of the host and each processing elements follow:

Behavior of Host

(1) Run algorithm 5.3 to assign affected constraints into n different groups, namely, $T_1 , T_2 , \ldots , T_n$.

(2) Assign group T_i to processing elements P_i

(3) Wait for the message from the processing elements. If any constraint is violated, then reject the update; otherwise, commit the update.

Behavior of Processing elements

(1) Validate the constraints assigned to it by the host.

(2) If any violation is detected, inform the host about the failure.

(3) If the validation of all the constraints in the group succeeds, then inform the host about the success.

5.6.3 Parallel Checking at Constraint Sub-formula Level

In this section, we discuss the parallel checking of constraints at the sub-formula level. Suppose a constraint C_l is decomposed into $C_{l_1}, C_{l_2}, \cdots, C_{l_m}$ and there are n processing elements. $\delta_1, \delta_2, \cdots, \delta_m$ are the cost coefficients and $\sigma_1, \sigma_2, \cdots, \sigma_m$ are the sufficiency frequencies.

The behaviors of the host and the processing elements are as follows:

Behavior of Host

(1) Order C_{l_i} (i=1, 2, ... ,m) in descending order on the basis of $\frac{\sigma_i}{\delta_i}$ to form a queue Q_c.

(2) Assign one constraint sub-formula from the front of Q_c to each available processing element to validate.

(3) Wait for the messages from the processing elements. If any constraint sub-formula is satisfied, then inform all the processing elements involved to stop checking the current constraint sub-formulas. Then the host starts to check other constraint formulas.

Behavior of the processing elements

(1) Validate the constraint assigned to it by host.

(2) If the validation is a success, then inform the host about this.

(3) If the validation is a failure and there is still one subformula in the queue Q_c, pick up one from the front of the queue and validate.

5.6.4 Performance

In chapter 4, we analyzed the performance of constraint enforcement in a uniprocessor based system with the application of decomposition theory. We made some assumptions about the distribution of run-time parameters, as performance depends on the distribution of the run-time parameters. Also, in a uniprocessor based system, the order in which a constraint sub-formula is checked is important in improving the performance.

In this section, we try to analyze performance when n processing elements are used. Suppose that the number of constraint sub-formulas is m and the cost to check the original constraint is T_c. The following theorem provides the basis for our performance analysis.

Theorem 5.2

Suppose there are n processing elements available and m constraint sub-formula decomposed from an original constraint C with a checking cost T_c, then the following results hold.

(1) In the worst case, the cost to validate constraint C with decomposition is T_c.

(2) If m<n, then the average cost to validate constraint C with decomposition is $\frac{T_c}{m}$.

(3) If n<m, then the average cost to validate constraint C with decomposition is $\frac{T_c}{n}$.

Proof:

[1] The worst case occurs when the checking of all constraint sub-formulas fails. When this happens, there is no gain in utilizing multi-processors and the decomposition technique, because the original constraint formula must be validated.

[2] If m<n, i.e., the number of constraint sub-formulas is less than the number of processing elements, then each processing element can validate one constraint sub-formula. Thus, on average, the cost for a multiprocessor based systems is $\frac{T_c}{m}$

[3] If n≤m, i.e., the number of constraint sub-formulas is greater than the number of processing elements, then each processing element can validate more than one constraint sub-formulas. Thus, on average the cost in the multiprocessor based systems is $\frac{T_c}{n}$ ■

The results of this theorem are very conservative, because by the way that constraint sub-formulas is assigned to processing elements, it is very likely that a

constraint sub-formula will be satisfied in an early stage.

5.7 Summary

In this chapter, we have discussed the enforcement of constraints on a multiprocessor based system. We investigated the parallelism that exists in a database management system. We proposed a database machine architecture which support both SIMD and MIMD mode operations. With the constraint decomposition theory developed in chapter 3, we can enforce constraints in parallel.

Chapter VI: Local Enforcement Tuning

6.1 Introduction

In the previous chapters, we have developed a constraint decomposition theory and applied the theory to constraint enforcement in both sequential and parallel computational environments. It is noted that the enforcement methods discussed in chapter 4 and 5 are at a very high level. In sequential environments, they deal with the issue of how to utilize the run-time characteristics of a database state to determine the checking order to achieve global efficiency. In parallel environments, they deal with finding a way to distribute constraints or constraint sub-formulas to processing elements so that the processing power of a multi-processor based system can be maximally exploited. In neither case have we tried to answer the question of how to check each individual constraint sub-formula or formula. In this chapter, existing techniques are extended and used to further simplify each constraint sub-formula at the time of validation. By doing so, we will complete the proposed integrated approach to database constraint enforcement.

6.2 Characteristics of Enforcement Tuning

At the stage of constraint decomposition, only the properties of the constraint formula is used. When we consider the question on how we actually validate each individual constraint sub-formula, more information is available in tuning the constraint enforcement.

The information available at this stage, which is not known previously is:

[1] Types of the formula: We have a set of constraint sub-formulas, each consisting of the conjunction of atomic formulas. In general, constraint sub-formulas involved contain less quantifiers than the original formula.

[2] Types of Update. At this stage, we know whether the update is *insert*, *delete* or *modify*.

[3] The tuples(data) to be updated, which can be used to simplify the constraint sub-formula before validating.

Consequently, each individual constraint sub-formula is much simpler than the original one. Furthermore, more information can be used to enforce the simpler constraint sub-formulas. Our goal is to find methods to enforce efficiently each constraint sub-formula, utilizing the additional information available at this stage. It is also observed that local enforcement tuning is done at run-time, i.e., at the time when a database update is to be carried out.

The rest of the chapter is organized as follows. Section 6.3 introduces the preliminary concepts that we need for the later discussions. Section 6.4 presents methods in evaluating simple constraint sub-formulas which involve at most two quantifiers. Section 6.5 discusses an approach in validating a certain class of constraint subformulas which involves multiple ranges. Section 6.6 concludes this chapter.

6.3 Preliminaries

This section introduces the basic terms and facts for the sake of clear presentation of the following sections.

DEFINITION 6.1 (*Target Variables*)

Let (Q x ∈ R) M be an integrity constraint, where Q is either ∀ or ∃, R is a relation name, and M is a valid constraint formula. If an update is on relation R, then x is called a target variable. ■

It is clear that target variables largely determine the truthfulness of an integrity constraint upon a database update.

DEFINITION 6.2 (*Substitution*)

A substitution ψ is a set of pairs $\{ x_1/t_1, x_2/t_2, \ldots, x_m/t_m \}$; where the x_i's are distinct tuple variables and the t_i's are tuple constants. Applying ψ to a vcf W leads to the vcf W(ψ)(called an instance of W) where each tuple variable which appears both in W and in ψ has been replaced in W by the associated constants in ψ. ■

Example 6.1:

Consider the constraint

∀x∈ SALE∃y∈ SUPPLY((x.DEPT=y.DEPT)∧(x.ITEM=y.ITEM))

If (D_3, I_3) is to be inserted into the relation SALE, then x is called the target variable. Applying the substitution $\{ x / (D_3, I_3)\}$ to the constraint, we have:

∃y∈ SUPPLY((y.DEPT=D_3)∧(y.ITEM=I_3)) ■

Let W be a valid constraint sub-formula. If we replace every occurrence of $x_1, x_2, \cdots, x_m$ in W by $t_1, t_2, \cdots, t_m$ respectively, then we denote the resulting instance by $W(x_1/t_1, x_2/t_2, \cdots, x_m/t_m)$.

Since a constraint formula is a closed formula, every tuple variable is bound to a given relation. Thus, there is one range prefix for any variable appearing in a

constraint formula. When a tuple constant is substituted for a tuple variable, the corresponding range prefix will be removed from the resulting formula(instance).

The following lemmas are needed in the development of the simplification process for constraint sub-formula validation.

LEMMA 6.1:

The following equations hold:

$$(\forall x \in R)W \equiv \forall x(x \in R \rightarrow W) \tag{6.1}$$

$$(\exists x \in R)W \equiv \exists x(x \in R \wedge W) \tag{6.2}$$

$$\forall x(x \in \{\tau\} \rightarrow W) \equiv W(x/\tau) \tag{6.3}$$

Proof:

The proofs of the above equations follow directly from the definitions of $\forall$, $\exists$, and substitution.■

LEMMA 6.2:

Let R_{new} be the relation obtained by adding a tuple τ to a relation R_{old}, that is, $R_{new} = R_{old} \cup \{\tau\}$, then the following holds.

$$(\forall x \in R_{new})W \equiv (\forall x \in R_{old})W \wedge W(x/\tau) \tag{6.4}$$

$$(\exists x \in R_{new})W \equiv (\exists x \in R_{old})W \vee W(x/\tau) \tag{6.5}$$

Proof:

Formulas (6.4) and (6.5) are intuitively correct. The formal proofs are as follows.

The proof of (6.4)

$$\begin{aligned}(\forall x \in R_{new})W &\equiv \forall x(x \in R_{new} \rightarrow W) && \text{By (6.1).}\\ &\equiv \forall x(\neg x \in R_{new} \vee W)\end{aligned}$$

$$\equiv \forall x(\neg(x \in R_{old} \vee x \in \{\tau\}) \vee W)$$
$$\equiv \forall x((\neg x \in R_{old} \wedge \neg x \in \{\tau\}) \vee W)$$
$$\equiv \forall x((\neg x \in R_{old} \vee W) \wedge (\neg x \in \{\tau\} \vee W))$$
$$\equiv \forall x((x \in R_{old} \rightarrow W) \wedge (x \in \{\tau\} \rightarrow W))$$
$$\equiv \forall x((x \in R_{old} \rightarrow W) \wedge W(x/\tau)) \qquad By\ (6.3).$$
$$\equiv \forall x(x \in R_{old} \rightarrow W) \wedge W(x/\tau)$$
$$\equiv (\forall x \in R_{old})W \wedge W(x/\tau)$$

The proof of (6.5)

$$(\exists x \in R_{new})W \equiv \exists x(x \in R_{new} \wedge W) \qquad By\ (6.2).$$
$$\equiv \exists x((x \in R_{old} \vee x \in \{\tau\}) \wedge W)$$
$$\equiv \exists x((x \in R_{old} \wedge W) \vee (x \in \{\tau\} \wedge W))$$
$$\equiv \exists x((x \in R_{old} \wedge W) \vee W(x/\tau)) \qquad By\ (6.3)$$
$$\equiv \exists x(x \in R_{old} \wedge W) \vee W(x/\tau)$$
$$\equiv (\exists x \in R_{old}) \wedge W \vee W(x/\tau)$$

■

LEMMA 6.3:

Let R_{new} be the relation obtained by deleting a tuple τ from a relation R_{old}, that is, $R_{new} = R_{old} - \{\tau\}$, then the following holds.

$$(\forall x \in R_{old})W \equiv (\forall x \in R_{new})W \wedge W(x/\tau) \qquad (6.6)$$
$$(\exists x \in R_{old})W \equiv (\exists x \in R_{new})W \vee W(x/\tau) \qquad (6.7)$$

Proof:

The proof of (6.6)

$$(\forall x \in R_{old})W \equiv \forall x(x \in R_{old} \rightarrow W) \qquad By\ (6.1).$$
$$\equiv \forall x(\neg x \in R_{old} \vee W)$$
$$\equiv \forall x(\neg(x \in R_{new} \vee x \in \{\tau\}) \vee W)$$

$$\equiv \forall x((\neg x \in R_{new} \wedge \neg x \in \{\tau\}) \vee W)$$
$$\equiv \forall x((\neg x \in R_{new} \vee W) \wedge (\neg x \in \{\tau\} \vee W))$$
$$\equiv \forall x((x \in R_{new} \rightarrow W) \wedge (x \in \{\tau\} \rightarrow W))$$
$$\equiv \forall x((x \in R_{new} \rightarrow W) \wedge W(x/\tau)) \qquad By\ (6.3).$$
$$\equiv \forall x(x \in R_{new} \rightarrow W) \wedge W(x/\tau)$$
$$\equiv (\forall x \in R_{new})W \wedge W(x/\tau)$$

The proof of (6.7)

$$(\exists x \in R_{old})W \equiv \exists x(x \in R_{old} \wedge W) \qquad By\ (6.2).$$
$$\equiv \exists x((x \in R_{new} \vee x \in \{\tau\}) \wedge W)$$
$$\equiv \exists x((x \in R_{new} \wedge W) \vee (x \in \{\tau\} \wedge W))$$
$$\equiv \exists x((x \in R_{new} \wedge W) \vee W(x/\tau)) \qquad By\ (6.3).$$
$$\equiv \exists x(x \in R_{new} \wedge W) \vee W(x/\tau)$$
$$\equiv (\exists x \in R_{new}) \wedge W \vee W(x/\tau)$$

■

With the preliminary concepts and facts presented, we can now proceed to introduce our constraint enforcement tuning methods.

6.4 Evaluation of Simple Constraint Sub-formula

This section discusses the methods in evaluating simple constraint sub-formulas which involve at most 2 quantifiers. The methods of evaluation for insertion, deletion and modification will be studied in each sub-section.

6.4.1 Insertion Validation

In this sub-section, we first introduce a theorem which provides a theoretical foundation for simplifying the constraint sub-formula validation when a insertion update is to be carried out. Then an algorithm based on that theorem will be presented

and discussed.

6.4.1.1 Insertion Theorem

Let $R_{new} \equiv R_{old} \cup \{\tau\}$ when a tuple τ is inserted in relation R.

THEOREM 6.1: Suppose all constraint formulas are satisfied before a database update and τ is to be inserted into relation R. If x_1 and x_2 are the only target variables, then the following holds:

(1) If the constraint sub-formula C_s is of the form: $(\forall\, x_1 \in \mathrm{R})\mathrm{W}$, then C_s will be satisfied if and only if $\mathrm{W}(x_1/\tau)$ is true.

(2) If the constraint sub-formula C_s is of the form:$(\exists x_1 \in R)\mathrm{W}$, then C_s will always be satisfied.

(3) If the constraint sub-formula C_s is of the form: $(\forall x_1 \in R)(\forall x_2 \in R)W$, then C_s will be satisfied if and only if

$(\forall x_1 \in R_{old})W(x_2/\tau) \wedge (\forall x_2 \in R_{old})W(x_1/\tau) \wedge W(x_1/\tau, x_2/\tau)$ is true.

(4) If the constraint sub-formula C_s is of the form: $(\exists x_1 \in R)(\exists x_2 \in R)\mathrm{W}$, then C_s will always be satisfied.

(5) If the constraint sub-formula C_s is of the form: $(\forall x_1 \in R)(\exists x_2 \in R)\mathrm{W}$, then C_s will be satisfied if and only if $(\exists x_2 \in R)W(x_1/\tau) \vee W(x_1/\tau, x_2/\tau)$ is true.

(6) If the constraint sub-formula C_s is of the form: $(\exists x_1 \in R)(\forall x_2 \in R)\mathrm{W}$, then C_s will be true if and only if

$$(\exists x_1 \in R)((\forall x_2 \in R)W \wedge W(x_2/\tau)) \vee (\forall x_2 \in R)W(x_1/\tau) \wedge W(x_1/\tau, x_2/\tau)$$

is true.

Proof of (1):

According to lemma 6.2, $(\forall x_1 \in R_{new})W \equiv (\forall x_1 \in R_{old})W \wedge W(x_1/\tau)$. Because $(\forall x_1 \in R_{old})$W is true in the current database, $(\forall x_1 \in R_{new})$W is true if and only if W(x_1/τ) is true.

Proof of (2):

According to lemma 6.2, $(\exists x \in R_{new})W \equiv (\exists x \in R_{old}) \vee W(x/\tau)$. Because $(\exists x \in R_{old})$W is true, $(\exists x \in R_{new})W$ is true, too.

Proof of (3):

By applying lemma 6.2 repeatedly, the following equation is derived.

$$\begin{aligned}(\forall x_1 \in R_{new})(\forall x_2 \in R_{new})W &\equiv (\forall x_1 \in R_{new})((\forall x_2 \in R_{old})W \wedge W(x_2/\tau)) \\ &\equiv (\forall x_1 \in R_{new})(\forall x_2 \in R_{old})W \wedge (\forall x_1 \in R_{new})W(x_2/\tau) \\ &\equiv (\forall x_1 \in R_{old})(\forall x_2 \in R_{old})W \wedge (\forall x_2 \in R_{old})W(x_1/\tau) \\ &\wedge (\forall x_1 \in R_{old})W(x_2/\tau) \wedge W(x_1/\tau, x_2/\tau)\end{aligned}$$

Because $(\forall x_1 \in R_{old})\forall x_2 \in R_{old})W$ is true, as a result, $(\forall x_1 \in R_{new})(\forall x_2 \in R_{new})$W is true if and only if $(\forall x_2 \in R_{old})W(x_1/\tau) \wedge (\forall x_1 \in R_{old})W(x_2/\tau) \wedge W(x_1/\tau, x_2/\tau)$ is true.

Proof of (4):

By applying lemma 6.2 repeatedly, the following equation is derived.

$$\begin{aligned}(\exists x_1 \in R_{new})(\exists x_2 \in R_{new})W &\equiv (\exists x_1 \in R_{new})((\exists x_2 \in R_{old})W \vee W(x_2/\tau)) \\ &\equiv (\exists x_1 \in R_{new})(\exists x_2 \in R_{old})W \vee (\exists x_1 \in R_{new})W(x_2/\tau)) \\ &\equiv (\exists x_1 \in R_{old})(\exists x_2 \in R_{old})W \vee (\exists x_2 \in R_{old})W(x_1/\tau) \\ &\vee (\exists x_1 \in R_{old})W(x_2/\tau) \vee W(x_1/\tau, x_2/\tau)\end{aligned}$$

Because $(\exists x_1 \in R_{old})(\exists x_2 \in R_{old})W$ is true, so is $(\exists x_1 \in R_{new})(\exists x_2 \in R_{new})$W.

The proof of (5):

By applying lemma 6.2 repeatedly, the following equation is derived.

$$\begin{aligned}(\forall x_1 \in R_{new})(\exists x_2 \in R_{new})W &\equiv (\forall x_1 \in R_{new})((\exists x_2 \in R_{old})W \vee W(x_2/\tau)) \\ &\equiv (\forall x_1 \in R_{old})((\exists x_2 \in R_{old})W \vee W(x_2/\tau)) \\ &\quad \wedge ((\exists x_2 \in R_{old})W(x_1/\tau) \vee W(x_1/\tau, x_2/\tau))\end{aligned}$$

Since $(\forall x_1 \in R_{old})((\exists x_2 \in R_{old})W \vee W(x_2/\tau))$ is always true, so $(\forall x_1 \in R_{new})(\exists x_2 \in R_{new})W$ is true if and only if $(\exists x_2 \in R_{old})W(x_1/\tau) \vee W(x_1/\tau, x_2/\tau)$ is true.

The proof of (6):

By applying lemma 6.2 repeatedly, the following equation is derived.

$$\begin{aligned}(\exists x_1 \in R_{new})(\forall x_2 \in R_{new})W &\equiv (\exists x_1 \in R_{new})((\forall x_2 \in R_{old})W \wedge W(x_2/\tau)) \\ &\equiv (\exists x_1 \in R_{old})((\forall x_2 \in R_{old})W \wedge W(x_2/\tau)) \\ &\quad \vee ((\forall x_2 \in R_{old})W(x_1/\tau) \wedge W(x_1/\tau, x_2/\tau))\end{aligned}$$

Thus $(\exists x_1 \in R_{new})(\forall x_2 \in R_{new})W$ is true if and only if $(\exists x_1 \in R_{old})((\forall x_2 \in R_{old}) \wedge W(x_2/\tau)) \vee ((\forall x_2 \in R_{old})W(x_1/\tau) \wedge W(x_1/\tau, x_2/\tau))$ is true. ■

The above theorem provides a theoretical foundation to validate a constraint sub-formula which is either one of the six forms mentioned above. In the next subsection, we present an algorithm based on the theorem.

6.4.1.2 Insertion Validation Algorithm

Figure 6.1 is an insertion validation algorithm based on the theorem developed in the previous subsection.

Algorithm 6.1: (Insertion Validation)

Input: A constraint sub-formula C_s and
a tuple τ to be inserted.

Output: A Status indicating whether C_s is satisfied or not.

Step-1: If $C_s=(\forall x_1 \in R)W$, then
$C_s'=W(x_1/\tau)$

else if $C_s=(\exists x_1 \in R)W$, then
$C_s'=T$

else if $C_s=(\forall x_1 \in R)(\forall x_2 \in R)W$, then
$C_s'=(\forall x_1 \in R_{old})W(x_2/\tau) \wedge (\forall x_2 \in R_{old})W(x_1/\tau)$
$\wedge W(x_1/\tau, x_2/\tau)$

else if $C_s=(\exists x_1 \in R)(\exists x_2 \in R)$W, then
$C_s'=T$

else if $C_s=(\forall x_1 \in R)(\exists x_2 \in R)$W, then
$C_s'=(\exists x_2 \in R)W(x_1/\tau) \vee W(x_1/\tau, x_2/\tau)$

else if $C_s=(\exists x_1 \in R)(\forall x_2 \in R)$W, then
$C_s'=(\exists x_1 \in R)((\forall x_2 \in R)W \wedge W(x_2/\tau))$
$\vee(\forall x_2 \in R)W(x_1/\tau) \wedge W(x_1/\tau, x_2/\tau)$

Step-2: Replace in C_s' each pre-valued literal
by its truth value and apply, as much as possible
the following absorption rules.
$\neg T \equiv F$; $\neg F \equiv T$;
$T \vee A \equiv T$; $F \vee A \equiv A$
$T \wedge A \equiv A$; $F \wedge A \equiv F$

Step-3: Validate the resulting formula derived in step 2 in
the current database state.

Figure 6.1. A Simple Insertion Validation Algorithm

Example 6.2:

Let R(U , V , W) be a relation scheme and $C_s=(\forall x \in R)(x.V<10)$. If τ=(3 5 22) is inserted in R, then after step 1, $C_s'=(2<10)$. After step 2, $C_s'=T$. Then, step 3 will return a status indicating that C_s is satisfied. ■

6.4.2 Deletion Validation

In this sub-section, we first introduce a theorem which provides a theoretical foundation for simplifying a constraint sub-formula validation when a deletion update is to be carried out. Then an algorithm based on the theorem will be presented.

6.4.2.1 Deletion Theorem

Let x be the only one target variable in the given constraint sub-formula C_s and let $R_{new} \equiv R_{old} - \{\tau\}$ when a tuple τ is deleted from relation R.

THEOREM 6.2: Suppose all constraint formulas or sub-formulas are satisfied before a database update and τ is to be deleted from relation R. If x_1 and x_2 are the only target variables, then the following holds:

(1) If the constraint sub-formula C_s is of the form ($\forall$ x_1 $\in$ R)W , then C_s will be always satisfied.

(2) If the constraint sub-formula C_s is of the form:$(\exists x_1 \in R)$W , then C_s will be satisfied if $W(x_1/\tau)$ is false.

(3) If the constraint sub-formula C_s is of the form: $(\forall x_1 \in R)(\forall X_2 \in R)W$, then C_s will be always satisfied.

(4) If the constraint sub-formula C_s is of the form: $(\exists x_1 \in R)(\exists x_2 \in R)$W: C_s will be satisfied if $(\exists x_1 \in R_{new})W(x_2/\tau) \wedge (\exists x_2 \in R_{new})W(x_1/\tau) \wedge W(x_1/\tau, x_2/\tau)$ is false.

(5) If the constraint sub-formula C_s is of the form: $(\forall x_1 \in R)(\exists x_2 \in R)$W, then C_s will be satisfied if $(\exists x_1 \in R_{new})W(x_2/\tau)$ is false.

(6) If the constraint sub-formula C_s is of the form: $(\exists x_1 \in R)(\forall x_2 \in R)$W, then C_s will be true if $(\forall x_2 \in R_{new})W(x_1/\tau) \wedge W(x_1/\tau, x_2/\tau)$ is false.

Proof of (1):

According to lemma 6.3, $(\forall x_1 \in R_{old})W \equiv (\forall x_1 \in R_{new})W \wedge W(x_1/\tau)$. Because $(\forall x_1 \in R_{old})$W is true in the current database, $(\forall x_1 \in R_{new})W$ is true after a deletion update.

Proof of (2):

According to lemma 6.3, $(\exists x_1 \in R_{old})W \equiv (\exists x_1 \in R_{new}) \vee W(x_1/\tau)$. Because $(\exists x_1 \in R_{old})W$ is true, if $W(x_1/\tau)$ is false, then $(\exists x_1 \in R_{new})W$ must be true.

Proof of (3):

By applying lemma 6.3 repeatedly, the following equation is derived.

$$\begin{aligned}(\forall x_1 \in R_{old})(\forall x_2 \in R_{old})W &\equiv (\forall x_1 \in R_{old})((\forall x_2 \in R_{new})W \wedge W(x_2/\tau)) \\ &\equiv (\forall x_1 \in R_{old})(\forall x_2 \in R_{new})W \wedge (\forall x_1 \in R_{old})W(x_2/\tau) \\ &\equiv (\forall x_1 \in R_{new})(\forall x_2 \in R_{new})W \wedge (\forall x_2 \in R_{new})W(x_1/\tau) \\ &\wedge (\forall x_1 \in R_{new})W(x_2/\tau) \wedge W(x_1/\tau, x_2/\tau)\end{aligned}$$

Because $(\forall x_1 \in R_{old})(\forall x_2 \in R_{old})W$ is true, $(\forall x_1 \in R_{new})(\forall x_2 \in R_{new})W$ is true.

Proof of (4):

By applying lemma 6.3 repeatedly, the following equation can be derived.

$$\begin{aligned}(\exists x_1 \in R_{old})(\exists x_2 \in R_{old})W &\equiv (\exists x_1 \in R_{old})((\exists x_2 \in R_{new})W \vee W(x_2/\tau)) \\ &\equiv (\exists x_1 \in R_{old})(\exists x_2 \in R_{new})\vee(\exists x_1 \in R_{old})W(x_2/\tau)) \\ &\equiv (\exists x_1 \in R_{new})(\exists x_2 \in R_{new})W \vee (\exists x_2 \in R_{new})W(x_1/\tau) \\ &\vee (\exists x_1 \in R_{new})W(x_2/\tau) \vee W(x_1/\tau, x_2/\tau)\end{aligned}$$

Because $(\exists x_1 \in R_{old})(\exists x_2 \in R_{old})W$ is true, to guarantee the truthfulness of $(\exists x_1 \in R_{new})(\exists x_2 \in R_{new})W$, $(\exists x_2 \in R_{new})W(x_1/\tau) \vee (\exists x_1 \in R_{new})W(x_2/\tau) \vee W(x_1/\tau, x_2/\tau)$ must be false.

The proof of (5):

By applying lemma 6.3 repeatedly, the following equation is derived.

$$\begin{aligned}(\forall x_1 \in R_{old})(\exists x_2 \in R_{old})W &\equiv (\forall x_1 \in R_{old})((\exists x_2 \in R_{new})W \vee W(x_2/\tau)) \\ &\equiv (\forall x_1 \in R_{new})((\exists x_2 \in R_{new})W \vee W(x_2/\tau)) \\ &\wedge ((\exists x_2 \in R_{new})W(x_1/\tau) \vee W(x_1/\tau, x_2/\tau))\end{aligned}$$

Because $(\forall x_1 \in R_{old})(\forall x_2 \in R_{old})W$ is true, $(\forall x_1 \in R_{new})((\exists x_2 \in R_{new})W \vee W(x_2/\tau))$ must be true. Thus, if $(\forall x_1 \in R_{new})W(x_2/\tau)$ is false, then $(\forall x_1 \in R_{new})(\exists x_2 \in R_{new})W$ will be true.

The proof of (6):

By applying lemma 6.3 repeatedly, the following equation is derived.

$$\begin{aligned}(\exists x_1 \in R_{old})(\forall x_2 \in R_{old})W &\equiv (\exists x_1 \in R_{old})((\forall x_2 \in R_{new})W \wedge W(x_2/\tau)) \\ &\equiv (\exists x_1 \in R_{new})((\forall x_2 \in R_{new})W \wedge W(x_2/\tau)) \\ &\vee ((\forall x_2 \in R_{new})W(x_1/\tau) \wedge W(x_1/\tau, x_2/\tau))\end{aligned}$$

Because $(\exists x_1 \in R_{old})(\forall x_2 \in R_{old})W$ is true, $(\exists x_1 \in R_{new})(\forall x_2 \in R_{new})W$ is true if $(\forall x_2 \in R_{new})W(x_1/\tau) \wedge W(x_1/\tau, x_2/\tau)$ is false. ■

Theorem 6.2 provides a theoretical foundation for validating a constraint sub-formula when a deletion update is to be carried out. However, some conditions for ensuring the correctness of the original formula is not necessary, i.e., the failure of those condition may not guarantee the failure of the original constraint. For example, consider the second case: if τ is deleted from relation R, and $W(x_1/\tau)$ is false, then the update does not violate the constraint sub-formula. However, even if $W(x_1/\tau)$ is true, it is still possible that $(\exists x_1 \in R_{new})W$ is true. Nevertheless, if the number of tuples in R that satisfy W is small, then the possibility of violation by deleting a tuple is also small. Otherwise, if the number of tuples in R that satisfy W is very large, then the possibility of violation by deleting a tuple is still small because there are many other tuples that satisfy W. It will pay off to record the number of tuples which satisfy W, denoted by N_R. If a tuple τ is to be deleted from R, and $W(x_1/\tau)$ is false, then the update does not violate this constraint sub-formula. On the other hand, if $W(x_1/\tau)$ is true, then one is subtracted from N_R. If N_R is zero, then the constraint sub-formula is violated, otherwise, the constraint sub-formula is still satisfied.

6.4.2.2 Deletion Validation Algorithm

In this sub-section, we present an deletion simplification algorithm based on the theorem described above. The input to this algorithm includes a constraint sub-formula C_s and a tuple τ that is to be deleted. The output is a status variable indicating whether C_s is satisfied or not.

Algorithm 6.2: (deletion validation)

Input: A constraint sub-formula C_s and
a tuple τ to be deleted

Output: A Status indicating whether C_s is satisfied
or not.

Step-1: If $C_s=(\forall x_1 \in R)W$, then
$C_s'=T$.

else if $C_s=(\exists x_1 \in R)$W, then
$C_s'=\neg W(x_1/\tau)$.

else if $C_s=(\forall x_1 \in R)(\forall X_2 \in R)W$, then
$C_s'=T$.

else if $C_s=(\exists x_1 \in R)(\exists x_2 \in R)$W, then
$C_s'=\neg(\exists x_1 \in R_{new})W(x_2/\tau) \wedge (\exists x_2 \in R_{new})W(x_1/\tau)$
$\wedge W(x_1/\tau, x_2/\tau)$

else if $C_s=(\forall x_1 \in R)(\exists x_2 \in R)$W, then
$C_s'=\neg(\exists x_1 \in R_{new})W(x_2/\tau)$

else if $C_s=(\exists x_1 \in R)(\forall x_2 \in R)$W, then
$C_s'=\neg(\forall x_2 \in R_{new})W(x_1/\tau) \wedge W(x_1/\tau, x_2/\tau)$

Step-2: Replace in C_s' each pre-valued literal
by its truth value and apply, as much as possible
the following absorption rules:
$\neg T \equiv F$; $\neg F \equiv T$;
$T \vee A \equiv A$; $F \vee A \equiv A$;
$T \wedge A \equiv A$; $F \wedge A \equiv F$;

Step-3: Validate the formula derived in step 2 in the current
database state.

Figure 6.2. A Deletion Validation Algorithm

6.4.3 Modification Validation

Modification is considered to be a special transaction that consists of an deletion followed by a insertion. Thus, modification can be validated using the theorems developed in the previous two sub-sections. Suppose a tuple τ in R is to be modified to τ'. The following illustrates the methods utilizing theorems 6.1 and 6.2 to validate modification updates.

Example 6.3: Let the constraint sub-formula C_s to be of the form: $(\forall x_1 \in R)W$. Because of theorem 6.2(1), deleting τ from R does not violate C_s. Furthermore, because of theorem 6.1 (1), inserting τ' into R satisfies C_s if and only if $W(x_1/\tau')$ is true. ■

Example 6.4: Let the constraint sub-formula C_s to be of the form: $(\exists x_1 \in R)W$. Considering theorems 6.1 and 6.2 together, the modification is true if one of the following is true:

(1) $W(x_1/\tau)$ is false.

(2) $W(x_1/\tau')$ is true.

(3) $(\exists x' \in R)W$ is true. ■

6.4.4 Discussion

It is observed that the goal of constraint enforcement tuning is to further simplify constraint sub-formulas so that it takes less time to validate the resulting constraint sub-formulas. It may appear that a derived constraint sub-formula is more complex

than the original one. However, the complexity is not measured by the length of the constraint sub-formulas, but by their checking spaces or the effort in reversing a database update.

It is also observed that the simple validation procedures can deal only with certain forms of constraint sub-formulas. Some forms of constraint formulas do not have both the necessary and sufficient conditions. For those cases, some other methods needs to be investigated, or the database update needs to be validated against the original constraint. Nevertheless, the materials presented so far serve the purpose to complete the proposed methodology.

6.5 Evaluation of Multiple Range Constraints

In the previous sections, the methods for evaluating a single update on a single range constraint sub-formula were presented. This section will touch upon the issues on evaluating multiple(two) updates against multiple(two) range constraints.

Definition 6.3: (*Multiple Range Constraint*)

A constraint formula/sub-formula is called a multiple range constraint if it is of the form $Q_1 x_1 \in R_1 \cdots Q_n x_n \in R_n M$ and not all the R_i are same. Q_i is either $\forall$ or $\exists$. If all the R_i are same, then the constraint formula or sub-formula is called a single range constraint. ■

Definition 6.4: (*Multiple Update*)

If more than one tuple is updated before checking an integrity constraint, then the group of updates is called a multiple update. ■

With the above definitions, we now investigate the simplification process for checking a 2-range constraint with a 2-update.

6.5.1 Insertion Validation

Theorem 6.3: Suppose all constraint formulas are satisfied before τ^1 is inserted into R^1 and τ^2 is inserted into R^2. If x_1 and x_2 are the only target variables, then the following holds:

(1) If the constraint sub-formula C_s is of the form: $(\forall x_1 \in R^1)(\forall x_2 \in R^2)W$, then C_s will be satisfied if and only if

$(\forall x_1 \in R^1_{old})W(x_2/\tau^2) \wedge (\forall x_2 \in R^2_{old})W(x_1/\tau^1) \wedge W(x_1/\tau^1, x_2/\tau^2)$ is true.

(2) If the constraint sub-formula C_s is of the form: $(\exists x_1 \in R^1)(\exists x_2 \in R^2)$W, then C_s will always be satisfied.

(3) If the constraint sub-formula C_s is of the form: $(\forall x_1 \in R^1)(\exists x_2 \in R^2)$W, then C_s will be satisfied if and only $(\exists x_2 \in R^2_{old})W(x_1/\tau^1) \vee W(x_1/\tau^1, x_2/\tau^2)$ is true.

(4) If the constraint sub-formula C_s is of the form: $(\exists x_1 \in R^1)(\forall x_2 \in R^2)$W, then C_s will be true if and only if

$(\exists x_1 \in R^1_{old})((\forall x_2 \in R^2_{old})W \wedge W(x_2/\tau^2)) \vee (\forall x_2 \in R^2)W(x_1/\tau^1) \wedge W(x_1/\tau^1, x_2/\tau^2)$ is true.

Proof of (1):

$$
\begin{aligned}
(\forall x_1 \in R^1_{new})(\forall x_2 \in R^2_{new})W &\equiv (\forall x_1 \in R^1_{new})((\forall x_2 \in R^2_{old})W \wedge W(x_2/\tau^2)) \\
&\equiv (\forall x_1 \in R^1_{new})(\forall x_2 \in R^2_{old})W \wedge (\forall x_1 \in R^1_{new})W(x_2/\tau^2) \\
&\equiv (\forall x_1 \in R^1_{old})(\forall x_2 \in R^2_{old})W \wedge (\forall x_2 \in R^2_{old})W(x_1/\tau^1)
\end{aligned}
$$

$$\wedge(\forall x_1 \in R^1_{old})W(x_2/\tau^2) \wedge W(x_1/\tau^1, x_2/\tau^2)$$

Since $(\forall x_1 \in R^1_{old})$ is true, as a result, $(\forall x_1 \in R^1_{new})(\forall x_2 \in R^2_{new})$W is true if and only if $(\forall x_2 \in R^2_{old})W(x_1/\tau^1) \wedge (\forall x_1 \in R^1_{old})W(x_2/\tau^2) \wedge W(x_1/\tau^1, x_2/\tau^2)$ is true.

Proof of (2):

$$\begin{aligned}(\exists x_1 \in R^1_{new})(\exists x_2 \in R^2_{new})W &\equiv (\exists x_1 \in R^1_{new})((\exists x_2 \in R^2_{old})W \vee W(x_2/\tau^2)) \\ &\equiv (\exists x_1 \in R^1_{new})(\exists x_2 \in R^2_{old}) \vee (\exists x_1 \in R^1_{new})W(x_2/\tau^2)) \\ &\equiv (\exists x_1 \in R^1_{old})(\exists x_2 \in R^2_{old})W \vee (\exists x_2 \in R^2_{old})W(x_1/\tau^1) \\ &\quad \vee (x_1 \in R^1_{old})W(x_2/\tau^2) \vee W(x_1/\tau^1, x_2/\tau^2)\end{aligned}$$

Because $(\exists x_1 \in R^1_{old})(\exists x_2 \in R^2_{old})W$ is true, so is $(\exists x_1 \in R^1_{new})(\exists x_2 \in R^2_{new})$W.

The proof of (3):

$$\begin{aligned}(\forall x_1 \in R^1_{new})(\exists x_2 \in R^2_{new})W &\equiv (\forall x_1 \in R^1_{new})((\exists x_2 \in R^2_{old})W \vee W(x_2/\tau^2)) \\ &\equiv (\forall x_1 \in R^1_{old})((\exists x_2 \in R^2_{old})W \vee W(x_2/\tau^2)) \\ &\wedge ((\exists x_2 \in R^2_{old})W(x_1/\tau^1) \vee W(x_1/\tau^1, x_2/\tau^2))\end{aligned}$$

Since

$$(\forall x_1 \in R^1_{old})((\exists x_2 \in R^2_{old})W \vee W(x_2/\tau^2))$$

is always true, so $(\forall x_1 \in R^1_{new})(\exists x_2 \in R^2_{new})W$ is true if and only if

$$(\exists x_2 \in R^2_{old})W(x_1/\tau^1) \vee W(x_1/\tau^1, x_2/\tau^2)$$

is true.

The proof of (4):

$$\begin{aligned}(\exists x_1 \in R^1_{new})(\forall x_2 \in R^2_{new})W &\equiv (\exists x_1 \in R^1_{new})((\forall x_2 \in R^2_{old})W \wedge W(x_2/\tau^2)) \\ &\equiv (\exists x_1 \in R^1_{old})((\forall x_2 \in R^2_{old}) \wedge W(x_2/\tau^2)) \\ &\vee ((\forall x_2 \in R^2_{old})W(x_1/\tau^1) \wedge W(x_1/\tau^1, x_2/\tau^2))\end{aligned}$$

Thus $(\exists x_1 \in R^1_{new})(\forall x_2 \in R^2_{new})W$ is true if and only if

$(\exists x_1 \in R^1_{old})((\forall x_2 \in R^2_{old}) \wedge W(x_2/\tau^2)) \vee ((\forall x_2 \in R^2_{old}) W(x_1/\tau^1) \wedge W(x_1/\tau^1, x_2/\tau^2))$ is true.

■

6.5.2 Deletion Validation

THEOREM 6.4: Suppose all constraint formulas or sub-formulas are satisfied before τ^1 is deleted from relation R^1 and τ^2 is deleted from R^2. If x_1 and x_2 are the only target variables, then the following holds:

(1) If the constraint sub-formula C_s is of the form: $(\forall x_1 \in R^1)(\forall x_2 \in R^2)W$, then C_s will be always satisfied.

(2) If the constraint sub-formula C_s is of the form: $(\exists x_1 \in R^1)(\exists x_2 \in R^2)$W: C_s will be satisfied if $(\exists x_1 \in R^1_{new})W(x_2/\tau^2) \wedge (\exists x_2 \in R^2_{new})W(x_1/\tau^1) \wedge W(x_1/\tau^1, x_2/\tau^2)$ is false.

(3) If the constraint sub-formula C_s is of the form: $(\forall x_1 \in R^1)(\exists x_2 \in R^2)$W, then C_s will be satisfied if $(\exists x_1 \in R^1_{new})W(x_2/\tau^2)$ is false.

(4) If the constraint sub-formula C_s is of the form: $(\exists x_1 \in R^1)(\forall x_2 \in R^2)$W, then C_s will be true if $\forall x_2 \in R^2_{new})W(x_1/\tau^1) \wedge W(x_1/\tau^1, x_2/\tau^2)$ is false.

Proof of (1):

$$
\begin{aligned}
(\forall x_1 \in R^1_{old})(\forall x_2 \in R^2_{old})W &\equiv (\forall x_1 \in R^1_{old})((\forall x_2 \in R^2_{new})W \wedge W(x_2/\tau^2)) \\
&\equiv (\forall x_1 \in R^1_{old})(\forall x_2 \in R^2_{new})W \wedge (\forall x_1 \in R^1_{old})W(x_2/\tau^2) \\
&\equiv (\forall x_1 \in R^1_{new})(\forall x_2 \in R^2_{new})W \wedge (\forall x_2 \in R^2_{new})W(x_1/\tau^1) \\
&\wedge (\forall x_1 \in R^1_{new})W(x_2/\tau^2) \wedge W(x_1/\tau^1, x_2/\tau^2)
\end{aligned}
$$

and $(\forall x_1 \in R^1_{old})(\forall x_2 \in R^2_{old})W$ is true, so $(\forall x_1 \in R^1_{new})(\forall x_2 \in R^2_{new})W$ is also true.

Proof of (2):

Because of the following equation

$$\begin{aligned}(\exists x_1\in R^1_{old})(\exists x_2\in R^2_{old})W &\equiv(\exists x_1\in R^1_{old})((\exists x_2\in R^2_{new})W\vee W(x_2/\tau^2))\\ &\equiv(\exists x_1\in R^1_{old})(\exists x_2\in R^2_{new})\vee(\exists x_1\in R^1_{old})W(x_2/\tau^2))\\ &\equiv(\exists x_1\in R^1_{new})(\exists x_2\in R^2_{new})W\vee(\exists x_2\in R^2_{new})W(x_1/\tau^1)\\ &\quad\vee(x_1\in R^1_{new})W(x_2/\tau^2)\vee W(x_1/\tau^1,x_2/\tau^2)\end{aligned}$$

and $(\exists x_1\in R^1_{old})(\exists x_2\in R^2_{old})W$ is true, to guarantee the truthfulness of $\exists x_1\in R^1_{new})(\exists x_2\in R^2_{new})W$, $(\exists x_2\in R^2_{new})W(x_1/\tau^1)\vee(x_1\in R^1_{new})W(x_2/\tau^2)\vee W(x_1/\tau^1,x_2/\tau^2)$ must be false.

The proof of (3):

Because of the following equation:

$$\begin{aligned}(\forall x_1\in R^1_{old})(\exists x_2\in R^2_{old})W &\equiv(\forall x_1\in R^1_{old})((\exists x_2\in R^2_{new})W\vee W(x_2/\tau^2))\\ &\equiv(\forall x_1\in R^1_{new})((\exists x_2\in R^2_{new})W\vee W(x_2/\tau^2))\\ &\wedge((\exists x_2\in R^2_{new})W(x_1/\tau^1)\vee W(x_1/\tau^1,x_2/\tau^2))\end{aligned}$$

and $(\forall x_1\in R^1_{old})(\forall x_2\in R^2_{old})W$ is true. So $(\forall x_1\in R^1_{new})((\exists x_2\in R^2_{new})W\vee W(x_2/\tau^2))$ must be true. Thus, if $(\forall x_1\in R^1_{new})W(x_2/\tau^2)$ is false, then $(\forall x_1\in R^1_{new})(\exists x_2\in R^2_{new})W$ must be true.

The proof of (4):

Because of the following equation:

$$\begin{aligned}(\exists x_1\in R^1_{old})(\forall x_2\in R^2_{old})W &\equiv(\exists x_1\in R^1_{old})((\forall x_2\in R^2_{new})W\wedge W(x_2/\tau^2))\\ &\equiv(\exists x_1\in R^1_{new})((\forall x_2\in R^2_{new})\wedge W(x_2/\tau^2))\\ &\vee((\forall x_2\in R^2_{new})W(x_1/\tau^1)\wedge W(x_1/\tau^1,x_2/\tau^2))\end{aligned}$$

and $(\exists x_1\in R^1_{old})(\forall x_2\in R^2_{old})W$ is true. So $(\exists x_1\in R^1_{new})(\forall x_2\in R^2_{new})W$ is true if $(\forall x_2\in R^2_{new})W(x_1/\tau^1)\wedge W(x_1/\tau^1,x_2/\tau^2)$ is false. ■

6.5.3 Discussion

In this section, we have introduced enforcement methods for multiple (two) range and multiple(two) update problem. This serves mainly for theoretical interest, as it is a generalization of single range single update problem. Yet, more work is needed to further generalize those methods to handle arbitrary transactions.

6.6 Summary

In this chapter, we have discussed methods for enforcing each constraint sub-formulas. We presented theorems and algorithms for validating a single update against single range constraint sub-formulas which include at most two quantifiers. We also touched upon the issues on enforcing multiple updates against multiple range constraints.

Constraint enforcement tuning is done at run-time. There are other simplification methods that can also be used in this framework.

Chapter VII: Conclusions

7.1 Introduction

In the preceding chapters, an integrated constraint enforcement methodology with the application of constraint decomposition theory was addressed. The development of a constraint decomposition theory which provides a foundation for efficient constraint enforcement in both uniprocessor and multiprocessor based systems was a major goal of this research. In this chapter the significance of this work is summarized and several avenues of future research are identified.

7.2 Significance

This research work is a result of a long term study on constraint management in database and expert database systems. It is a natural result of the deficiency of previous methods and the perception of the author about new facets of the problem which can leads to a more efficient solution.

As modern database systems manage very large volume of data and the number of constraints are huge, the time spent in checking constraints is quite expensive. Many researchers have been working to alleviate the problems by:

(1) Limiting the range of the integrity constraints. Assume that the old database state is consistent prior to each update. When a database is updated, only the part of the database whose integrity may be affected by the update is the target of integrity checking.

(2) Making use of existing information.

(3) Making use of aggregate information.

(4) Logical simplification of constraint formulas.

(5) Making use of physical properties of a database(index, etc).

However, the above mentioned approached are limited because:

- They can be applied to only a few types of constraints,
- It is not efficient enough.

To overcome the deficiencies of previous approaches, we realized that the fundamental problem of their approaches was the failure to understand the nature of the constraint enforcement problem. The nature of constraint enforcement is the protection of database integrity from *accidental* violation. That is, the constraint is rarely violated, just as drivers are rarely involved in a car accident. The result of the misunderstanding of the problem is the isolation of logical properties of integrity constraint formulas and the physical characteristics of a database. Previous researchers either tried to simplify the constraint formula independent of the characteristics of a database state and update patterns, or they tried to design good access methods or a data structure independent of the logical properties of constraint formulas. Because the logical properties of integrity constraint formulas and the physical characteristics of database are related, we cannot achieve overall efficiency without considering them at the same time. With the assumption that the probability of a database constraint is extremely small, we can make use of the characteristics of the database state, update pattern and logical properties of database constraints to develop an efficient database

enforcement methodology.

The major contributions of this research are:

[1] For the first time, the new perspective of database constraint enforcement problem is studied. The novelty of the proposed methodology starts from the understanding of the nature of the problem. The philosophy to approach a problem in this way is in itself a contribution.

[2] For the first time, the characteristics of database update and state is considered in connection with properties of constraint formulas in enforcing constraints. Previous approaches failed to consider the dynamic nature of the database. They either ignore the physical properties of databases or properties of constraint formulas.

[3] We introduce a formal decomposition procedure to decompose a constraint formula into a set of constraint subformulas so that each constraint sub-formula can be enforced independently. Because of the locality of database updates, some constraint formulas are more efficient to check than others.

[4] Parallel checking of constraints has been discussed. Because of the independence of constraint sub-formulas, our approach provide a possible way to enforce constraints in parallel. Also, integrity constraint enforcement in a multiprocessor based database machine was discussed.

[5] The staged approach makes it possible to attack the different problems at different times. Constraint decomposition is done at compile time and is done only once, at the time a constraint is defined. Although both global enforcement strategy and local enforcement tuning are to be done at run-time, the isolation of the

global enforcement strategy and local enforcement tuning is superior to previous integrated approaches, because the problems to be solved at the two stages are different. Thus, by isolation we can concentrate on solving different problems at different times.

7.3 Further Research

The completion of the research reported here has given insight into related areas which might be fruitfully investigated.

1. Searching For Other Decomposition Methods

The question whether there exists other constraint decomposition methods which are better than the one presented in this research needs to be answered. As the decomposition theory developed has made use of only syntactic properties of constraint formulas, we can see if semantic information be used to aid the decomposition process.

2. Checking Space

The analysis of constraint enforcement in terms of checking space was only briefly presented. How to precisely judge constraint enforcement based the criterion of checking space need further study.

3. Physical Implementation Issues

The work presented in this dissertation touched only upon the theoretical aspects of constraint enforcement problems. Physical implementation of constraint enforcement has scarcely been discussed. The issues for physical representation of a constraint formula or sub-formulas in this framework needs further study.

4. Deferred Checking

Sometimes, when the probability of constraint is small, more efficiency can be achieved by deferred checking. Deferred checking means that the validation of a constraint is not done right after a database update. Suppose the cost of checking some constraint is very big, then it may be beneficial to find a way to group a set of updates such that the group size is optimal in the sense to minimize the cost.

Formally, consider the following parameters:

κ: Cost to validate a constraint.

μ: Cost to update a tuple

ρ: The probability that an update violates a constraint

γ: The probability that an update satisfies a constraint. Note that $\gamma=1-\rho$

$T_u(n)$: The total cost for n updates with undeferred checking.

$T_d(n)$: The total cost for n updates with deferred checking.

Based on the above parameters, we have the following equations for T_u and T_d

$$\begin{aligned} T_u &= n\kappa + n\mu + \rho(\mu+\kappa) + \rho^2(\mu+\kappa) + \cdots + \rho^n(\mu+\kappa) \\ &= n(\mu+\kappa) + \rho\frac{1-\rho^n}{1-\rho}(\mu+\kappa) \\ &= n(\mu+\kappa) + \frac{\rho}{\rho}(1-\rho^n)(\mu+\kappa) \end{aligned} \tag{7.1}$$

$$T_d = \kappa + n\mu + (1-\gamma^n)(n\mu+\kappa) \tag{7.2}$$

The objective is to minimize $\frac{T_d}{T_u}$ to find the optimal group size n.

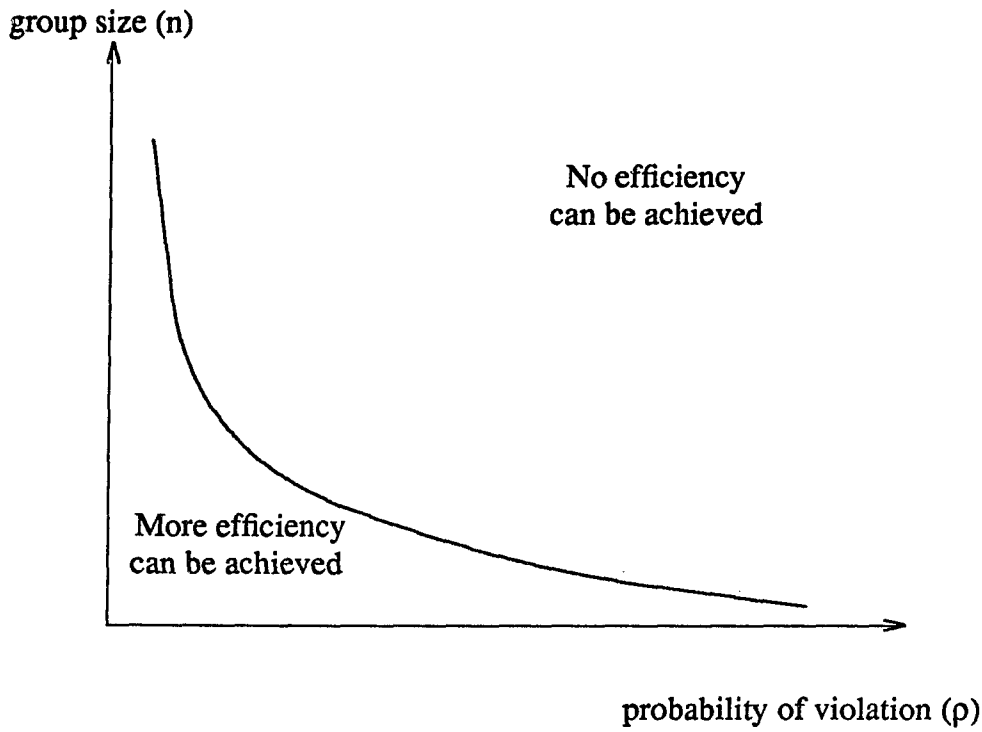

Figure 7.1 Optimal Deferred Checking

Figure 7.1 shows graphically the philosophy of the above mentioned approach. As we can see from the figure 7.1, the higher the probability of constraint violation, the smaller the group size. For a given probability of constraint violation, there is a point in the curve with a corresponding group size. The corresponding group size on the curve is the optimal with respect to the given probability. In the area below the curve, we underutilize the opportunity of big group size, thus better efficiency, whereas in the area above the curve we overutilize the opportunity of big group size, i.e., we intend to achieve more efficiency that cannot be realized because of the frequent transaction rollback.

5. Searching For Other Global Enforement Strategies

The efficiency of a global enforcement strategy depends on many factors of a database system. Generally speaking, there is no single strategy that is superior to all others. When a configuration of a database system changes, a corresponding global enforcement strategy needs to be found to suit that configuration. The independence of the global enforcement strategy and local enforcement tuning makes this task a lot easier.

6. Searching For Other Local Enforcement Tuning Methods

As pointed in the previous chapter, there are many other constraint simplification methods that can be extended or adapted for the purpose of local enforcement tuning. The success of an individual method also depends on some other factors such as, the supporting physical database structures, properties of a constraint sub-formulas and update patterns. Finding a good method for a given constraint sub-formula at a given time needs further investigation.

7.4 Concluding Remarks

The enforcement of integrity constraint in modern database or expert database systems is a difficult task. Database management software will be more successful if it has an efficient constraint enforcement mechanism. In this study we have attempted to develop a constraint decomposition theory which can be used in the constraint enforcement to improve the performance. A contribution to accomplish this has been made, but work remains.

Bibliography

[AbH87] S. Abiteboul and R. Hull, IFO: A Formal Semantic Data Model, *Proc. Third ACM Symp. on Principles of Database Systems*, 1987, 119-132.

[AdN85] M. Adiba and G. T. Nguyen, Handling Constraints and Meta-Data on a Generalized Data Management System, *Expert Database Systems*, 1985.

[AHU74] A. V. Aho, J. E. Hopcroft and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

[ABU79] A. V. Aho, C. Beeri and J. D. Ullman, The Theory of Joins in Relational Databases, *ACM Transactions on Database Systems 8*, 2 (1979), 297-314.

[BaP79] Badal,D.Z and Popek,G.J., Cost and Performance Analysis of Semantic Integrity Validation Methods, *Proceedings of 1979 ACM SIGMOD Int'l Conf. on Management of Data*, May 1979, 109-115.

[BaK86] F. Bancilhon and S. Khoshafian, A Calculus for Complex Objects, *Proceedings of Fifth ACM Symposium on Principles of Database Systems*, 1986, 53-59.

[BFH77] C. Beeri, R. Fagin and J. H. Howard, A Complete Axiomatization for Functional and Multivalued Dependencies, *ACM SIGMOD International Conference on Management of Data*, 1977, 47-61.

[BBC80] Bernstein,P.A., Blaustein,B.T. and Claerk,E.M.,, Fast Maintenance of Semantic Integrity Assertions Using Redundant Aggregate Date, *Proceedings of 6th Int'l Conference on VLDB*, Montreal, 1980.

[BeA84] E. Bertino and D. Apuzzo, Integrity Aspects in Data Base Management Systems, *IEEE Trends and Applications: Making Database Work*, 1984, 43-52.

[Bla81] B. T. Blaustein, Enforcing Database Assertion: Techniques and Applications, Tech. Rep.-21-81, Harvard University, Cambridge, Massachusetts, 1981.

[BoM79] R. S. Boyer and J. S. Moore, *A Computational Logic*, Academic Press, New York, 1979.

[Bro78] M. L. Brodie, Specification and Verification of Database Semantic Integrity, Ph.D Thesis, University of Toronto, Toronto, 1978.

[BrJ86] M. L. Brodie and M. Jarke, On Integrating Logic Programming and Databases, in *Expert Database Systems*, L. Kerschberg (editor), Benjamin/Cummings, 1986, 191-207.

[BuC79] O. P. Bumeman and E. K. Clemons, Efficient Monitoring Relational Databases, *ACM Transactions on Database Systems 4*, 3 (1979), 368-382.

[CaB80] M. A. Casanova and P. A. Bernstein, Formal System for Reasoning about Programs Accessing a Relational Database, *ACM Transactions on Programming Language 2*, 3 (July, 1980), 386-414.

[ChL73] C. Chang and R. C. Lee, *Symbolic Logic and Mechanical Theorem Proving*, Academic Press, New York, 1973.

[Che76] P. P. Chen, The Entity Relationship Model: Toward a Unified View of Data, *ACM Transactions on Database Systems 1*, 1 (March 1976), 9-37.

[Chi68] D. L. Childs, Feasibility of a set-theoretical data structure-a general structure based on a reconstituted definition of relation, *Proceedings of 1968 IFIP Congress*, Amsterdam, 1968, 162-172.

[Cod70] E. F. Codd, A Relational Model for Large Shared Data Banks, *Communication of ACM 6*, 13 (June, 1970), 377-387.

[Cod71] E. F. Codd, A Data Base Sublanguage founded on the Relational Calculus, in *Proceedings of ACM-SIGFIDET Workshop on Data Description, Access, and Control*, San Diego, California, 1971.

[CrD83] A. B. Cremers and Domanng, AIM: An Integrity Monitor For the Database System INGRES, *Proceedings of 9th International Conference on VLDB*, October, 1983.

[Dat81] C. J. Date, Referential Integrity, in *7th Int'l Conf. on Very Large Database Systems*, September, 1981.

[Dat83] C. J. Date, *An Introduction to Database Systems*, Addison-Wesley Publ. Company, 1983.

[Dat86] C. J. Date, *An Introduction to Database Systems*, Addison-Wesley, Reading Mass., 1986.

[Dec86] H. Decker, Integrity Enforcement on Deductive Database, in *Proceedings of the First International Conference on Expert Database Systems*, L. Kerschberg (editor), Charleston, South Carolina, April 1-4, 1986, 271-285.

[DCE85] A. Dogac, P. P. Chen and N. Erol, The Design and Implementation of an

Integrity Subsystem for the Relational DBMS RAP, *Proceedings of the 4th International Conference on Entity Relationship Approach*, October, 1985.

[ELG84] H. D. Ehrich, H. W. Lipeck and M. Gogolla, Specification, Semantics, and Enforcement of Dynamic Database Constraints, *Proceedings of 10th International Conference on VLDB*, Siggapore, August, 1984, 301-308.

[EsC75] K. P. Eswaran and D. D. Chamberlin, Functional Specifications of a subsystem for Data Base Integrity, in *Proc. 1st VLDB Conf.,,* Framingham, 1975.

[Esw76] K. P. Eswaran, Specification, Implementation and Interaction of a Trigger Subsystem in an Integrated Database System, *IBM Research Report RJ1820*, 1976.

[FeS81] E. B. Fernandez and R. C. Summers, *Database Security and Integrity*, Addison-Wesley, 1981.

[Fos74] B. M. Fossum, Data Base Integrity as Provided for by a Particular Data Base Management System, in *Data Base Management*, North Holland, 1974.

[FrW83] R. A. Frost and S. Whittaker, A Step Toward The Automatic Maintenance Of The Semantic Integrity Of Databases, *Computer Journal 26*, 2 (May, 1983), 124-133.

[FSC81] A. Furtado, D. Santos and J. Castilho, Dynamic Modeling of a Simple Existence Constraint, *Information Systems 6* (1981), 73-80.

[GMN78] H. Gallaire, J. Minker and J. M. Nicolas, An Overview and Introduction to Logic and Data Bases, in *Logic and Databases*, H. Gallaire and J. Minker (editor), Plenum Press, New York, 1978, 3-30.

[GaM79] G. Gardarin and M. Melkanoff, Proving the Consistency of Database Transactions, *Proceedings of the 5th International Conference on Very Large Data Bases*, Rio de Janeiro, October, 1979, 291-298.

[HaM75] M. M. Hammer and D. J. Mcleod, Semantic Integrity in a Relational Data Base System, *Proceedings of The First International Conference on VLDB System*, Framingham, 1975, 25-47.

[HaM76] M. Hammer and D. Mcleod, A Framework For Database Semantic Integrity, *Proceedings of 2nd International Conference On Software Engineering*, San Francisco, 1976.

[HaS78] M. Hammer and S. K. Sarin, Efficient Monitoring of Database Assertions,, in *Presented at the ACM SIGMOD 78 Conf.*, Austin, 1978.

[HaD72] I. T. Hawryskiewycz and J. B. Dennis, An Approach to Proving the Correctness of Data Base Operations, in *Proceedings of ACM-SIGFIDET Workshop on Data Description, Access, and Control*, 1972.

[HMN84] L. J. Henschen, W. W. McCune and S. A. Naqvi, Compiling Constraint-Checking Program from First-Order Formulas, in *Advances in Data Base Theory*, vol. 2 , H. Gallaire, J. Minker and J. M. Nicolas (editor), Plenum Press, Nw York, 1984, 145-169.

[Hoa69] C. A. R. Hoare, An Axiomatic Basis for Computer Programming,

Communication of the ACM 12, 10 (October,1969), 576-580.

[HsI85] A. Hsu and T. Imielinski, Integrity Checking For Multiple Updates, *SIGMOD Rec 14*, 4 (Dec. 1985), 152-168.

[IyS87] S. S. Iyengar and Y. Sheng, A Parallel Architecture for Integrity Constraint Checking, *Technical Report #87-021*, Baton Rouge, 1987.

[Jac82] B. E. Jacobs, On Database Logic, *Journal of the Association for Computing Machinery 29*, 2 (April 1982), 310-332.

[Jac85] B. E. Jacobs, *Applied Database Logic*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1985.

[Ker89] L. Kerschberg, *Proceedings from the Second International Conference on Expert Database Systems*, The Benjamin/Cummings Publishing Company, Inc., 1989.

[Kin84] J. J. King, *Query Optimization by Semantic Reasoning*, UMI Research Press, Ann Arbor, Michigan, 1984.

[Kob81] I. Kobayashi, Evaluation of Queries Based on the Extended Relational Calculi, *International J. Comput. Inform. Sci. 2*, 10 (1981), 63-103.

[Kob84] I. Kobayashi, Validating Database Updates, *Information Systems 9*, 1 (1984), 1-17, Pergamon Press.

[Laf79] G. Lafue, An Approach to Automatic Maintenance of Semantic Integrity in Large Design Databases, *National Computer Conference, AFIPS*, 1979.

[Laf82] G. M. E. Lafue, Semantic Integrity Dependencies and Delayed Integrity Checking, *Eighth International Conference on Very Large Data Bases*, Mexico City, 1982.

[LaS86] G. M. E. Lafue and R. G. Smith, Implementation of Semantic Integrity Manager with a Knowledge Representation system, in *Expert Database Systems*, L. Kerschberg,1986 (editor), The Benjaming/Cummings Publishing Company, Inc., 1986, 330-350.

[Mai83] D. Maier, *The Theory of Relational Database*, Computer Science Press, Rockville, MD., 1983.

[Men79] E. Mendelson, *Introduction to Mathematical Logic*, D. Van Nostrand Company, New York, 1979.

[MoI87] A. Moitra and S. S. Iyengar, Parallel Algorithms For Some Computational Problems, *Advances in Computers 26* (1987).

[Mor86] M. Morgenstern, The Role of Constraints in Databases, Expert Systems, and Knowledge Representation, in *Expert Database Systems*, L. Kerschberg (editor), The Benjaming/Cummings Publishing Company, Inc., 1986, 351-368.

[Nak83] R. Nakano, Integrity Checking in a Logic-Oriented ER Model, in *Entity-Relationship Approach to Software Engineering*, C. G. Davis, S. Jajodia, P. A. Ng and R. T. Yeh (editor), North-Holland, 1983, 551-565.

[NiY78] J. M. Nicolas and K. Yazdanian, Integrity Checking in Deductive Data Bases, in *Logic and Databases*, H. Gallaire and J. Minker (editor),

Plenum Press, New York, 1978.

[NiG78] J. M. Nicolas and H. Gallaire, Data Base: theory vs. interpretation, in *Logic and Databases*, H. Gallaire and J. Minker (editor), Plenum Press, New York, 1978.

[Nic82] J. Nicolas, Logic for Improving Integrity Checking in Relational Data Bases, *Acta Informatica 18* (1982), 227-253, Springer-Verlag.

[Opp78] D. C. Oppen, Reasoning about Recursively Defined Data Structures, *Proceedings of Fifth Symposium on Principles of Programming Languages*, Tucson, Ariz., January, 1978, 151-157.

[Ozk86] E. Ozkaranhan, *Database Machines and Database Management*, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1986.

[Phi84] N. C. K. Phillips, Safe Data Type Specifications, *IEEE Transactions on Software Engineering 10*, 3 (May, 1984), 285-289.

[QiS87] X. Qian and D. R. Smith, Constraint Reformulation for Efficient Validation, *Proceedings of 13th International Conference on Very Large Databases*, Sept. 1987, 417-425.

[Qia88] X. Qian, An Effective Method for Integrity Constraint Simplification, *Proceedings of 4th International Conference on Data Engineering*, Feb, 1988.

[ShS85] T. Sheard and D. Stemple, Coping with Complexity in Automated Reasoning about Database Systems, *Proceedings of the 11th International Conference on Very Large Data Bases*, Stockholm, August,

1985, 426-435.

[ShS89] T. Sheard and D. Stemple, Automatic Verification of Database Transaction Safety, *ACM Transactions on Database Systems 14*, 2 (September, 1989), 322-368.

[ShK86] A. Shepherd and L. Kerschberg, Constraint Management in Expert Database Systems, in *Expert Database Systems*, L. Kerschberg (editor), The Benjamin/Cummings Publishing Company Inc., 1986, 309-330.

[SiV86] E. Simon and P. Valduriez, Integrity Control in Distributed Database Systems, *Proceedings of Hawaii International Conference on System Sciences*, Jan 1986.

[SiV87] E. Simon and P. Valduriez, Design and Analysis of a Relational Integrity Subsystem, Tech. Report DB-015-87, 1987.

[Smi86] J. M. Smith, Expert Database Systems: A Database Perspective, in *Expert Database Systems*, L. Kerschberg (editor), Benjamin/Cummings, 1986, 3-15.

[StS84] D. Stemple and T. Sheard, Specification and Verification of Abstract Database Types, *Proceedings of the Third Symposium on Principles of Database Systems*, Waterloo, April, 1984, 248-257.

[StS85] D. Stemple and T. Sheard, Database Theory for Supporting Specification-Based Database System Development, *Proceedings of Eighth International Software Engineering Conference*, London, August, 1985, 43-39.

[SSB86] D. Stemple, T. Sheard and R. Bunker, Abstract Data Types in Database: Specification, Manipulation and Access, *Proceedings of the IEEE Second International Conference on Data Engineering*, Los Angeles, February, 1986, 590-597.

[SMS87] D. Stemple, S. Mazumdar and T. Sheard, On the Modes and Meaning of Feedback to Transaction Designers., *Proceedings of ACM SIGMOD Conference*, San Francisco, May, 1987, 374-386.

[Sto75] M. Stonebraker, Implementation of Integrity Constraints and Views by Query Modification, in *Proceedings of ACM-SIGMOD International Conference on the Management of Data*, San Jose, California, May, 1975.

[TsL82] D. C. Tsichritzis and F. H. Lochovsky, *Data Models*, Prentice-Hall, Inc., 1982.

[Ull82] J. D. Ullman, *Principles of Database Systems*, Computer Science Press, Rockville, Maryland, 1982.

[Ull88] J. D. Ullman, *Principles of Database and Knowledge-Base Systems*, Computer Science Press, 1988.

[Var86] M. Y. Vardi, On The Integrity of Databases With Incomplete Information, *Proceedings of Fifth ACM Symposium on Principles of Database Systems*, 1986, 252-266.

[Var88] M. Y. Vardi, Fundamentals of Dependency Theory, *Trends in Theoretical Computer Science*, Rockville, MD, 1988, 171-224.

[WaS81] A. Walker and S. C. Salveter, Automatic Modification of Transactions to

Preserve Data Base Integrity Without Undoing Updates, *Tech. Report 81/026, State University of New York at Stony Brook*, June, 1981.

[WSK83] W. Weber, W. Stucky and J. Karszt, Integrity Checking In Database Systems, *Information Systems 8*, 2 (1983), 125-136.

[Wil72] M. V. Wilkes, On Preserving the Integrity of Data Bases, *The Computer Journal 3*, 15 (1972), 191-194.

VITAE

Yuejian Sheng was born to Anlan Sheng and Xiansen Yao on October 5, 1960, in Shenyang, The People's Republic of China. He graduated from Shenyang No. 1 High School in July 1978. He entered Peking University in China in September 1978 and received a Bachelor of Science degree in computer science in July 1982.

In August 1984, he enrolled in graduate school at Louisiana State University in Baton Rouge. He is now a candidate for a Doctor of Philosophy degree in Computer Science.

DOCTORAL EXAMINATION AND DISSERTATION REPORT

Candidate: Yuejian Sheng

Major Field: Computer Science

Title of Dissertation: Database Constraint Enforcement: A Decompositional Approach

Approved:

Major Professor and Chairman

Dean of the Graduate School

EXAMINING COMMITTEE:

Date of Examination:

2/16/90