uOttawa

L'Université canadienne
Canada's university

FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTDOCTORALES

uOttawa

L'Université canadienne
Canada's university

FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES

Arif Emre Caglar

AUTEUR DE LA THÈSE / AUTHOR OF THESIS

M.C.S.

GRADE / DEGREE

School of Information and Technology Engineering

FACULTÉ, ÉCOLE, DÉPARTEMENT / FACULTY, SCHOOL, DEPARTMENT

Query-aware Generation of Database Instances for Testing Databases Applications

TITRE DE LA THÈSE / TITLE OF THESIS

H. Ural

DIRECTEUR (DIRECTRICE) DE LA THÈSE / THESIS SUPERVISOR

CO-DIRECTEUR (CO-DIRECTRICE) DE LA THÈSE / THESIS CO-SUPERVISOR

EXAMINATEURS (EXAMINATRICES) DE LA THÈSE / THESIS EXAMINERS

I. Kiringa

S. Ajila

Gary W. Slater

Le Doyen de la Faculté des études supérieures et postdoctorales / Dean of the Faculty of Graduate and Postdoctoral Studies

Query-aware Generation of Database Instances for Testing Database Applications

Arif Emre Caglar

Thesis submitted to the
Faculty of Graduate and Post Doctoral Studies
in partial fulfillment of the requirements
for the Masters in Computer Science Degree*

School of Information Technology and Engineering
Faculty of Engineering
University of Ottawa

---

* The Masters program in Computer Science is a joint program with Carleton University, administered by the Ottawa-Carleton Institute for Computer Science

Your file Votre référence
ISBN: 978-0-494-61285-9
Our file Notre référence
ISBN: 978-0-494-61285-9

Abstract

Test data preparation phase for testing database applications deals with instantiating program variables, as well as generating database instance(s) to run application queries on these instances. We first propose a method that takes a database schema, a test case representing a sequence of simple queries on a path within a transaction in a given database application , and corresponding properties associated with these queries; forms a Constraint Satisfaction Problem (CSP) by considering the interactions among the queries and the integrity constraints given by the schema; and generates, by solving this CSP, an initial database instance which satisfies the property associated with each query and is consistent with respect to the integrity constraints given in the schema. Then, we extend this method by taking a sequence of test cases containing nested queries, and generate one or more consistent database instances that satisfy the properties of queries in each test case in the given sequence of test cases. We also discuss how to integrate our method in an approach given in the literature which instantiates the input host variables of queries automatically.

Acknowledgements

I would like to thank my thesis supervisor Dr. Hasan Ural for introducing me to Database Applications Testing field, for his guidance during my Masters and for his hospitality during my first year in Canada. I also would like to thank Dr. Herna Viktor for giving me the opportunity to become her Teaching Assistant, and Dr. Iluju Kiringa for being very helpful in times of need. I would like to acknowledge the generous financial support from Natural Science and Engineering Research Council and Ontario Centres of Excellence.

I consider myself the luckiest person for being the child of Adil and Guler Caglar, their endless support, patience and love are appreciated beyond words. I also wish to thank all my closest friends for their good fellowship, support and great times we had.

Table of Contents

List of Figures

List of Tables

Chapter One

Introduction

1.1 Background

A *database* (DB) is a structured collection of data that is stored in a computer system. *Database Management System* (DBMS) is a software, which provides organizations with efficient data access, data independence and integrity. The most commonly used model to represent a database is the *relational model* [RAM03]. Since relations represent tables in mathematics, the *relational model* structures the data in tabular format.

Database Applications (DBAs) retrieve or modify the information in a relational database by using a query language, such as *Structured Query Language* (SQL) [DAT97]. SQL consists of a *Data Definition Language* (DDL) and a *Data Manipulation Language* (DML). DDL is the standard language for creating, deleting and manipulating tables in SQL, whereas DML is the standard language for creating, modifying, deleting and retrieving the data via INSERT, UPDATE, DELETE and SELECT queries [RAM03]. The *logical schema*, which describes the stored data in terms of the data model of the DBMS [RAM03], can be expressed by DDL. DDL is also used to express *integrity constraints* (ICs), which are used to ensure the accuracy and consistency of data in relational databases.

Although lots of research have been devoted to increase the performance and reliability of DBMSs, relatively little attention has been given to testing of database applications (DBAs) that are using them. Testing a DBA ensures its consistency with its specification, and can be partitioned in three subproblems: *Test Cases Generation* (TCG), *Test Data Preparation* (TDP) and *Test Execution and Output Verification* (TEOV) [CHA99].

A DBA can be viewed as a relation from an input space *I* to an output space *O*. Thus, a DBA can be tested by applying selected values from *I*, and checking whether resulting values from *O* agrees with the specification of DBA [CHA04].

Given an instance in time, a *database state* refers to the complete database environment; including DBMS, table structures and data, stored procedures, constraints and other functionality at that time and a *database instance* refers to data that is being stored in terms of rows of tables. Database instance is a part of both selected values from *I* and resulting values from *O*. Therefore, the TDP subproblem deals with generating a database instance, as well as generating application inputs, in particular, generating values for the *input host variables*, which are the program variables used to pass values from a DBA to SQL queries.

When generating a database instance for testing a DBA, one needs to consider the schema of the DB that the DBA uses, queries embedded in the DBA and ICs that are possibly formed in the requirement analysis phase of the DB design. Query-awareness when creating a new database instance is important because one would want the queries in the DBA under test to return meaningful results when executed on the resulting database instance. Ideally, when testing a DBA, one would like an *initial database instance* such that every test case can run on this instance. Such an initial database instance must also be a *consistent* database instance i.e., one that satisfies all the constraints associated with the given logical schema.

1.2 Motivation and Objectives of the Thesis

A widely used programming convention in the development of DBAs is that programmers separate the DBA into several transactions. For instance, in an online registration application developed for the use of students in a university, enrolling in a course, dropping a course, retrieving transcript information and many more functions

would be grouped into separate transactions to avoid data inconsistencies in the database and to boost the performance of this DBA. Therefore, each transaction can be thought of as a logical unit which accomplishes certain functionality within a DBA. In this context, one would want to test each transaction to assure their correctness with respect to their functionality. In a realistic application, each transaction will contain multiple queries to fulfill its task. Therefore, we will consider a test case to be a path in a transaction containing a sequence of queries embedded in the host language code in a DBA.

In the case of testing DBAs, TDP problem can be broadly partitioned into initializing input host variables of each test case and generating consistent initial database instances so that each test case can be executed on it. We will first assume that the tester provides the values of the input host variables. Therefore, TDP is reduced to the generation of a consistent database instance. Later, we will relax this assumption by integrating our method to an approach given in [EMM07] which is, to the best of our knowledge, the only TDP approach for testing DBAs that does not assume that the input host variables are given.

Most of the methods generating a database instance in literature are not query-aware, and these methods are not usable in testing, because the database instance is generated without considering the queries in the DBA under test, which often leads to empty query results. The ones that are query-aware have very restrictive assumptions, such as considering a single SELECT query, and are far from practical usage. There are other methods that consider a test case to be a path in a transaction, but these methods can only handle simple queries and the interactions between queries are ignored.

A query-aware database instance generation method should have the following characteristics to provide a complete testing environment in terms of the database instance:

1. Given a test case containing multiple queries with a property assigned to each one of them, executing each query should satisfy the property assigned to it.

2. Generated database instance should be consistent, satisfying all the integrity constraints given by the logical schema.

3. Generated database instance could be used for as many test cases as possible, thus, saving time by minimizing the cost of generating and resetting different databases for each test case.

A test case in the context of database instance generation is represented in this thesis as a sequence of INSERT, DELETE, UPDATE or SELECT queries. When a test case is run, all the queries in this test case will be executed on the database instance that is to be generated. Thus, each execution of INSERT, DELETE and UPDATE query will lead to a modified database instance, and will be the part of the input domain for the following query to be executed, which we call *interaction* between queries. As stated earlier, there is no method in the literature that considers the interactions between queries in a test case. Therefore, generating tuples for each query in isolation by ignoring the interactions between queries, then merging the results is not a desired way to deal with the execution of multiple queries in an order, because execution of one query might modify the tuples generated for another query, which can result in empty result sets returned by the queries in the test case, or failure to update the tuples in the database.

In our work, we will address the problem of generating consistent database instances by considering the interactions among queries in a given sequence of queries.

1.3 Contributions of the Thesis

Although the majority of the applications in practice are database applications, and relational database theory has matured over several decades, there is not enough research

and automation in testing database applications. In this thesis, we propose a method that significantly automates the generation of database instances so that testers will be freed from using non query-aware methods to generate database instances and worrying about empty result sets and interactions between queries.

Specifically, we propose a method for solving the following problem: Given a test suite as a sequence of test cases, the database schema and properties assigned to each query of each test case of the test suite, we devise a method to generate consistent database instances with respect to integrity constraints such that queries in each test case satisfy their assigned properties, and the number of database instances generated are minimized with respect to the order of the test cases in the given test suite to reduce the number of resets to the database during testing. We use properties (i.e., exists or not exists) associated with queries to specify whether a tuple should be generated for a query.

For ease of presentation, we first consider a special case of this problem where the given test suite consists of only one test case which is given as a sequence of simple queries. We show how executing one query on a database instance can affect the execution of the other, which we name as *conflict*. We categorize the conflicts and provide methods to generate tuples such that executing one query on these tuples will not affect the execution of the other, which we call *resolving* the conflict.

We then consider the integrity constraints given with the logical schema, and provide conditions for a generated database instance to be consistent with respect to these integrity constraints. Using the proposed mechanisms for conflict detection and resolution as well as those of handling integrity constraints, we propose a method for solving the following problem: given a DBA, a logical schema of the database that the DBA uses, a test case as a sequence of simple queries along with their associated properties, and the values for the input host variables, generate a consistent initial

database instance where each query in the given test case can run without violating properties associated with the queries.

In practice however, it is common to encounter nested queries in DBAs. Therefore, we extend our method to handle a sequence of nested queries. Using this extension, we then present our method for the solution of the problem stated above where the given test suite consists of a sequence of test cases where each test case is a sequence of queries.

1.4 Organization of the Thesis

Chapter 2 gives the terminology that we need for our discussion, which is related to the relational model, SQL, integrity constraints and ternary logic, and the constraint satisfaction problem.

Chapter 3 reviews the related work done on the three subproblems of testing database applications: Test Cases Generation (TCG), Test Data Preparation (TDP) and Test Execution and Output Verification (TEOV).

Chapter 4 presents a method for solving the following problem: given a DBA, a logical schema, a test case as a sequence of simple queries along with their associated properties, and the values for the input host variables, generate an initial database instance where each query in the given test case can run without creating a conflict and violating the integrity constraints given by the schema.

Chapter 5 considers more complex queries, and further extends the method to handle nested queries using IN, NOT IN, EXISTS, NOT EXISTS, ANY and ALL keywords. We discuss the modifications that have to be done to the method proposed in Chapter 4 to handle nested queries. Also, our method for generating minimal number of database instances to run a test suite consisting of a sequence of test cases is discussed in this chapter.

Chapter 6 concludes the thesis, summarizes the strengths and weaknesses of the method, and gives directions for future research. This chapter also elaborates on the practical use of the proposed method. We first review an approach that instantiates input host variables and generates a database instance given a DBA and coverage criterion. We discuss the weaknesses of this approach in terms of database instance generation, and how to integrate our method to this approach to obtain test suites with instantiated host variables as input to our method for database instance generation. Finally, we discuss how to relax some of the restrictive assumptions we made in Chapters 4 and 5.

Chapter Two

Preliminaries

2.1 Relational Model and DBA

We adopt the following terms and definitions in our discussion. R_k is a *relation* defined over a set $X_k = \{x_{k1}, x_{k2}, \ldots, x_{ky}\}$ of attributes, $y \geq 1$. A *tuple* T is an element of a relation R_k; $T \in R_k$. A database *schema* is a set of relations, $S = \{R_1, R_2, \ldots, R_m\}$, where $m \geq 1$ [TSA90]. We define a *database instance* D to be an m-tuple $<D_{R1}, D_{R2}, \ldots, D_{Rm}>$, where D_{Rk} is the partial database instance corresponding to R_k, $1 \leq k \leq m$. In other words, $D_{Rk} = \{T_{k1}, T_{k2}, \ldots, T_{kz}\}$, where $z \geq 1$ and every tuple in D_{Rk} is an element of R_k.

Given a DBA, a test case Q is a sequence of queries $Q = q_1\ q_2\ \ldots\ q_n$, representing a path in a transaction in the DBA and a test suite is a sequence of test cases $\Omega = Q_1\ Q_2 \ldots Q_w$. Each query is one of the four types provided by the SQL: type(q_i) $\in$ {SELECT, INSERT, UPDATE, DELETE}. Moreover, D_i denotes the resulting database instance D after executing q_i, $1 \leq i \leq n$. D_0 is the initial database instance which we would like to generate.

In addition, as discussed in [ZHA01], when generating a database instance, the tester might want to assign a property to each query to control generation of tuples for the query, especially when executing a path in the DBA depends on whether the result set of a query is empty or not. The method discussed in [ZHA01] generates a database instance such that result of executing the query satisfies the property. However, we use the notion of property in our method in a slightly different way. Suppose that $P = p_1\ p_2 \ldots\ p_n$ is a sequence of properties, where $\forall i$, $p_i \in$ {EXISTS, NOT EXISTS} is the property assigned to q_i, $1 \leq i \leq n$. We say p_i is automatically satisfied by D_{i-1} if q_i = INSERT. Moreover, if there is some T such that T causes the WHERE clause of q_i to evaluate to *true* and if p_i =

EXISTS, or if there is no T such that T causes the WHERE clause of q_i to evaluate to *true* and if p_i = NOT EXISTS, then we say p_i is satisfied by D_{i-1}. In all other cases, p_i is not satisfied by D_{i-1}.

Similar to [ZHA01], we assume a constraint solver tool *CST*, which takes a set of unknowns and a set of constraints as an input, and either outputs the solution of each unknown satisfying all the constraints, or reports a contradiction.

2.2 SQL Queries

Given relations R_i and R_j defined over sets of attributes X_i and X_j, respectively, a *predicate* μ is defined to be $x_{is}\ \theta$ *Value* or $x_{is}\ \theta\ x_{jt}$ where $\theta \in \{=, <, \leq, >, \geq, \neq\}$ is a relational operator, *Value* $\in$ domain(x_{ks}), $1 \leq i, j \leq m$, $1 \leq s,t \leq y$.

A query q is a *simple query* if the WHERE clause of q consists of only conjunction or disjunction of predicates. A *subquery* is a SELECT query that is nested inside a SELECT, DELETE, UPDATE or INSERT query. Similarly, a *nested query* is a SELECT, DELETE, UPDATE or INSERT query which contains one or more SELECT subqueries nested in itself, which we also refer to as the *inner query*. Inner query of a nested query q is referred to as q'. In SQL, there is no theoretical limit on the number of subqueries that can be used inside a nested query.

2.3 Integrity Constraints

Integrity constraints (ICs) are used by database designers to ensure data accuracy and consistency. In SQL92 [SQL92] specification and onward, seven types of integrity constraints are defined, namely *primary key constraints*, *uniqueness constraints, foreign key constraints, domain constraints, not null constraints, table constraints,* and *assertions.*

A *primary key constraint* is a statement that a certain minimal subset of the fields of a

relation is a unique identifier for a tuple. A set of fields that uniquely identifies a tuple according to a key constraint is called a *candidate key* for the relation. Among candidate keys, a database designer identifies a primary key [RAM03]. A *uniqueness constraint* on an attribute also enforces all the values of the attribute to be unique as primary key constraint does, however, a null value is accepted for an attribute with a uniqueness constraint, whereas it is not accepted for an attribute with a primary key constraint.

Any tuple in a relation, in which there is a primary key constraint, can be referred to in another relation by using the primary key identifying the tuple. A *foreign key constraint* ensures that the referenced tuple exists in the referenced table.

A *domain constraint* is used to restrict values of a single attribute. As its name suggests, *not-null constraints* prevent an attribute to take a null value. Therefore, if we give the attribute as an unknown to a constraint solver tool, which will be explained in more detail in the next section, and use its domain constraint in the logical formula, a value in that unknown's domain will be found for that unknown, and the not-null constraint can be dealt with in this manner.

Table constraints are the constraints enforced over a single table. A table constraint involves multiple attributes of a single tuple of a single relation, and it is checked each time a new tuple is inserted or an existing tuple is modified. Since a table constraint can be enforced over multiple attributes of a tuple instead of one, table constraints subsume domain constraints.

In ANSI SQL 92 and 99 [SQL92] [SQL99], an *assertion* is a named integrity constraint that may relate to content of individual rows of the table, to the entire content of the table, or to a state required to exist among a number of tables.

In Chapter 4, we will talk about integrity constraints in more detail and explain how we

incorporate them in the methods we propose.

2.4 Ternary Logic

Ternary Logic is a logic system in which there are three truth values indicating *true*, *false*, and *unknown*. SQL implements ternary logic to represent the unknown data (missing or inapplicable data) with a NULL reserved keyword. Unknown values do not represent the non-existence of a value, SQL assumes that there exists a value, but it is not recorded in the database. Any arithmetic comparison using an unknown value results in an unknown value. However, a logic expression using an unknown value does not need to result in an unknown value. Below is a truth table for operations in Ternary Logic.

Table 2.1. Truth Table for Ternary Logic

A	***B***	***A* OR *B***	***A* AND *B***	**NOT *A***
True	True	True	True	False
True	Unknown	True	Unknown	False
True	False	True	False	False
Unknown	True	True	Unknown	Unknown
Unknown	Unknown	Unknown	Unknown	Unknown
Unknown	False	Unknown	False	Unknown
False	True	True	False	True
False	Unknown	Unknown	False	True
False	False	False	False	True

Ternary Logic is encountered in various components of a DBMS, such as Aggregate Functions, Joins, etc. However, in this thesis, we are particularly interested in the Ternary logic used in DML and table constraints, which will be explained in Chapter 4.

2.5 Constraint Satisfaction Problem

Constraint Satisfaction Problem consists of a set of unknowns and constraints, where the set of constraints govern the binding of these unknowns to a value in some domain [TSA93] [JEA97] [FRA07]. Formally, we can define a constraint satisfaction problem as a triple $<A, B, C>$, where A is set of unknowns, B is a domain of values, and C is a set of constraints. Every constraint in C is a pair $< V, R>$, where V is a tuple of variables and R is a set of tuples of values. All tuples have same number of elements and R is a relation. Evaluation of the unknowns in A is a function from unknowns to domains, f: $A \rightarrow B$. Such an evaluation satisfies a constraint $<(x_{k1}, x_{k2}, \ldots, x_{ky}), R_k>$ if $(f(x_{k1}),\ldots, f(x_{ky})) \in R_k$ [TSA93]. Solution of a CSP is obtained by finding values for each unknown in A such that all the constraints in C are satisfied. CSP problem is NP-complete, however, there are tools efficiently solving more complex problems than ours. Among others, [FRA07] provides details about constraint solving approaches and the tool we use (HySat).

We are interested in a subset of CSPs, namely, satisfiability of constraints in the form of Boolean expressions consisting of arithmetic expressions. In our work, unknowns represent the attributes or columns specific to a tuple having integer or real domain, and constraints are the logical formulas derived from WHERE clauses of queries and integrity constraints given in the logical schema. Solution is the set of tuples representing the database instance we want to generate.

2.6 Logical Formulas Derived from Queries

A *logical formula* $L(q)$ of a query q represents the conditions that must be satisfied for the tuples to be operated on by q. Since $L(q)$ of a query q will be used for generating values for the attributes refereed to in q, we need to augment the conjuction and/or disjunction of

predicates in the WHERE clause of q with the domain constraints of the attributes used in the predicates of the WHERE clause, as well as those of the attributes listed in SELECT clause if type(q) = SELECT, or the attributes listed in SET clause if type(q) = UPDATE. Hence, we construct $L(q)$ of a query q as follows:

We first build an initial version of $L(q)$ by using a function called **getWHEREclause(q)** which returns the conjuction and/or disjunction of predicates in the WHERE clause of q. Then, negation operators in L(q) are eliminated by using a procedure called **NegationFree(L(q))** given below.

Algorithm 2.1 NegationFree

```
procedure NegationFree(L(q))
   Let U, V, and W be predicates or subformulas in L(q).
   for each W in L(q) do
      If there is a join predicate in W then
         take it out (along with the domain constraints of W) of the negation
      endif
   endfor
   for each subformula of the form ¬ (U ∨ V) in L(q) do
      replace ¬ (U ∨ V) with ¬U ∧ ¬V
   endfor
   for each subformula of the form ¬ (U ∧ V) in L(q) do
      replace ¬ (U ∧ V) with ¬U ∨ ¬V
   endfor
   //After these two steps, since negations only apply to predicates//
   for each predicate ¬μ in L(q) do
      eliminate the negation by changing the relational operator θ in μ
```

```
      eliminate each double negation by changing ¬¬μ to μ.
   endfor
endprocedure
```

The $L(q)$ returned by NegationFree($L(q)$) is then converted into disjunctive normal form by algebraic manipulation. A logical formula is in *disjunctive normal form* (DNF) if it is a disjunction of *implicants*, which are conjunctions of predicates. If type(q) = SELECT, then before converting $L(q)$ into DNF, $L(q)$ is augmented with the conjunction of the domain constraints of the attributes listed in SELECT clause, by using a function called **getDomainConstraint(x)** which returns the domain constraint of the attribute x. We will refer to each implicant of $L(q)$ in DNF with a superscript, i.e., the first implicant of $L(q)$ as $L^1(q)$, and so on. Also, the number of implicants in $L(q)$ will be denoted by λ.

Finally, each implicant of $L(q)$ in DNF is augmented with the conjunction of the domain constraints of the attributes used in the implicant by using the function getDomainConstraint.

The procedure ObtainLogicalFormula represents the process of obtaining the logical formula for a query:

Algorithm 2.2 ObtainLogicalFormula

```
procedure ObtainLogicalFormula(q, L(q))
   L(q) ← getWHEREclause(q)
   NegationFree(L(q))
   if(type(q) = SELECT and p = ∃) then
      for each attribute x used in <attribute_list> of q do
         L(q) ← L(q) ∧ getDomainConstraint(x)
      endfor
   endif
```

```
    if(type(q) = UPDATE) then
       for each attribute x used in SET clause of q do
           L(q) ← L(q) ∧ getDomainConstraint(x)
       endfor
    endif
    convert L(q)into DNF by algebraic manipulation
    for each implicant L^k(q) in L(q)do
       for each attribute x used in L^k(q)do
          L^k(q) ← L^k(q) ∧ getDomainConstraint(x)
       endfor
    endfor
endprocedure
```

In a logical formula, there can be some predicate that violates the domain constraints of the attributes referred to in the predicate, or contradicts another predicate, or that is subsumed by other predicates. The procedure Simplify given below eliminates implicants containing predicates that contradict another predicate or violate domain constraints, and eliminates subsumed predicates from $L(q)$.

Let μ be a simple predicate in a query q of the form $(x\ \theta\ c)$, where x is an attribute and c is a constant, and let I_μ be the range of values from the domain of x restricted by the predicate μ, where substitution of any value from I_μ into x in μ will return the result *true*. Notice that if q does not have a WHERE clause, then we denote $I = (-\infty, +\infty)$. Two predicates μ and Ψ intersect if:

- μ is of the form $(x_1\ \theta\ x_2)$ and Ψ contains x_1 or x_2, or
- $I_\mu \cap I_\Psi \neq \varnothing$

Let v_1 and v_2 be two values of a domain, $\mu_1 = x_1\ \theta_1\ v_1$ and $\mu_2 = x_2\ \theta_2\ v_2$ and x_1 and x_2 be two attributes having the same name and belonging to the same relation. We say that

μ_1 *subsumes* μ_2 if the following conditions hold:

- $x_1 \in R$ and $x_2 \in R$, and
- x_1 has the same name with x_2, and
- $\mu_1 = \mu_2$, or
- If $\theta_1 \in \{\leq,<\}$ and $\theta_2 = \{<,\leq\}$, then $v_2 \geq v_1$, or
- If $\theta_1 \in \{\geq, >\}$ and $\theta_2 = \{\geq,>\}$, then $v_2 \leq v_1$

Algorithm 2.3 Simplify

```
procedure Simplify(L(q), S)
   for each implicant L^k(q) in L(q) do
      //Eliminate implicants containing predicates violating domain
constraints by checking the intersection of each predicate with the
corresponding domain constraints//
      for each predicate μ in L^k(q) do
         B^k = conjunction of domain constraints of attributes of μ in S
         if (μ does not intersect with any of the conjuncts of B^k)
            delete L^k(q) from L(q)
            //Break from the loop traversing each predicate of L^k(q)
            break
         endif
      endfor
      for each predicate μ_i = x_i θ_i v_i in L^k(q)
         for each predicate μ_j = x_j θ_j v_j in L^k(q), i ≠ j
            //Delete implicants containing contradicting predicates
            if (x_i = x_j and μ_i does not intersect with μ_j)
              delete L^k(q) from L(q)
            endif
```

```
            //Eliminate subsumed predicates
            if (x_i = x_j) and
                ((θ_i ∈ {≤,<} and θ_j = {<,≤}, and v_j ≥ v_i) or
                 (θ_i ∈ {≥, >} and θ_j = {≥,>}, and v_j ≤ v_i) or
                 (θ_i = θ_j and v_i = v_j)) then
                      remove μ_j from L^k(q)
            endif
         endfor
      endfor
   endfor
if (L(q) is empty)
   //If L(q) is empty at this point, input host variable instantiation should
   //be redone by the tester
   generate an error
endif
endprocedure
```

Example 2.1. In this example, we will illustrate how to form a logical formula from a SELECT query. Note that integrity constraints will not be considered in the examples until Section 4.6. Consider the query q_i shown below:

```
SELECT E.age, W.months
FROM Emp E, Works W
WHERE E.eid = W.eid AND E.salary < 5000 AND E.age < 40;
```

We form the logical formula $L(q_i)$ by using the procedures ObtainLogicalFormula and Simplify given above. The initial form of $L(q_i)$ is obtained from the WHERE clause of q_i by getWhereclause(q_i) which yields:

$L(q_i)$ = (E.eid = W.eid) $\wedge$ (E.salary < 5000) $\wedge$ (E.age < 40)

Since there is no negation in $L(q_i)$, NegationFree($L(q_i)$) returns $L(q_i)$ unchanged. Since type(q) = SELECT, $L(q_i)$ is augmented by the conjunction of domain constraints of *age* attribute of *Emp* table and *month* attribute of *Works* table appearing as the attributes listed in the SELECT clause of q_i by using getDomainConstraint(x) to yield:

$L(q_i)$ = (E.eid = W.eid) $\wedge$ (E.salary < 5000) $\wedge$ (E.age < 40) $\wedge$ (E.age $\geq$ 20 $\wedge$ E.age $\leq$ 80) $\wedge$ (W.months $\geq$ 0 $\wedge$ W.months $\leq$ 480)

Converting $L(q_i)$ into DNF yields

$L(q_i)$ = (E.eid = W.eid $\wedge$ E.salary < 5000 $\wedge$ E.age < 40 $\wedge$ E.age $\geq$ 20 $\wedge$ E.age $\leq$ 80 $\wedge$ W.months $\geq$ 0 $\wedge$ W.months $\leq$ 480)

Since $L(q_i)$ consists of a single implicant, this implicant is augmented by the conjunction of domain constraints of *eid, salary* and *age* attributes of *Emp* table and *eid* and *months* attributes of *Works* table by using getDomainConstraint(x) to yield:

$L(q_i)$ = (E.eid = W.eid $\wedge$ E.salary < 5000 $\wedge$ E.age < 40 $\wedge$ E.age $\geq$ 20 $\wedge$ E.age $\leq$ 80 $\wedge$ W.months $\geq$ 0 $\wedge$ W.months $\leq$ 480 $\wedge$ E.eid $\geq$ 10000 $\wedge$ E.eid $\leq$ 99999 $\wedge$ W.eid $\geq$ 10000 $\wedge$ W.eid $\leq$ 99999 $\wedge$ E.salary $\geq$ 1000 $\wedge$ E.salary $\leq$ 12000)

We simplify the logical formula above by removing the subsumed predicates, e.g., (E.salary < 5000 $\wedge$ E.salary $\leq$ 12000) = E.salary < 5000, by using the procedure Simplify($L(q_i)$,S) which returns the simplified $L(q_i)$ in DNF as the following:

$L(q_i)$ = (E.eid = W.eid $\wedge$ E.salary $\geq$ 1000 $\wedge$ E.salary < 5000 $\wedge$ E.age $\geq$ 20 $\wedge$ E.age < 40 $\wedge$ W.months $\geq$ 0 $\wedge$ W.months $\leq$ 480 $\wedge$ E.eid $\geq$ 10000 $\wedge$ E.eid $\leq$ 99999 $\wedge$ W.eid $\geq$ 10000 $\wedge$ W.eid $\leq$ 99999)

As one can observe, this formula consists of only one implicant, which is referred as $L^1(q_i)$. The predicates forming the implicants are referred to with another superscript. For instance, the predicate (E.eid = W.eid) in $L^1(q_i)$ can be referred as $L^{11}(q_i)$, (E.salary < 5000) can be referred as $L^{12}(q_i)$, and so on.

2.7 The Problem Definition

Given a test suite $\Omega = Q_1\ Q_2\ \ldots\ Q_w$, the database schema S along with the integrity constraints, and a sequence of properties P containing the properties corresponding to the queries in Ω, we partition Ω and obtain $\Omega' = \{\Omega_1\ \Omega_2 \ldots\ \Omega_h\}$ where $\Omega_k = Q_i\ Q_{i+1}\ \ldots\ Q_j$ $1 \leq k \leq h$, $1 \leq i < j \leq w$, and generate set of database instances $D' = D^1\ D^2 \ldots\ D^h$ such that each database instance is consistent with the integrity constraints given in S, and running each query q of each test case in Ω_k on D_k satisfies the property for q, $1 \leq k \leq h$.

In Chapter 4, we consider a test suite Ω consisting of a single test case $Q = q_1\ q_2\ \ldots\ q_n$ where q_i is a simple query. We generate a single consistent database instance D, such that such that executing every q_i in the order given by Q on D satisfies the corresponding property p_i, where $1 \leq i \leq n$.

In Chapter 5, we consider a test suite $\Omega = Q_1\ Q_2\ \ldots\ Q_w$, $w \geq 1$, instead of a single test case, and each query in the test cases is simple or nested. We explain our approach to partition the test suite Ω into $\Omega' = \{\Omega_1\ \Omega_2 \ldots\ \Omega_h\}$ such that if running Q_t and Q_r one after the other results in a contradiction, Q_t and Q_r are placed into different test suites Ω_i and Ω_j, $1 \leq t < r \leq w$ and $1 \leq i < j \leq h$.

Chapter Three

Previous Work

As stated in Chapter 1, testing database applications can be partitioned into three subproblems: *Test Cases Generation* (TCG), *Test Data Preparation* (TDP) and *Test Execution and Output Verification* (TEOV). In this chapter, we will discuss how these subproblems were addressed in the literature. Since our main concern is the TDP subproblem, we will have a broader discussion about this subproblem compared to the others.

3.1 Test Cases Generation

In the literature, a test case for a DBA can have different meanings. Most of the related work described below considers a test case to be a single query or a function containing a single embedded SQL query and implemented with a general purpose programming language, whereas others consider a test case to be multiple queries embedded in a function like us.

An important work done for testing DBAs is the AGENDA tool [CHA04] which is composed of Agenda Parser, Input Generator, State Generator, State Validator and Output Validator. It assumes a DBA consisting of a single query. A tester provides suggested values partitioned into data groups, called *sample-values files*. The authors define *test templates*, which are abstract test cases containing uninstantiated input host variables. Instantiating host variables of these templates according to the heuristics given by a tester results in the generation of test cases, which will be explained in Section 3.3. Templates also have preconditions and postconditions for aiding test outcome verification, and this will be explained in Section 3.2.

Chan et al. [CHA99] [CHE99] state that in traditional white box testing, the

semantics of SQL statements are rarely considered, thus, a DBA is considered as a black box in testing process. They proposed a method to transform SQL queries into procedures in host programming language, so that additional test cases using white box testing methods can be generated. Due to the variation of SQL commands supported by different DBMSs, Relational Algebraic Expressions are used as a common interface language between different DML command implementations. In this paper, there is an implicit assumption that queries are static in the host language. However, in most cases, queries make use of input host variables in the program. Therefore, they are dynamically created.

Kapfhammer and Soffa [KAP03] define a family of dataflow based test adequacy criteria for DBAs, namely, *all-database-DUs*, *all-relation-DUs*, *all-attribute-DUs*, *all-record-DUs*, and *all-attribute-value-DUs* as companions to the traditional *all-DUs* dataflow adequacy criterion. In their work, a DELETE or UPDATE query is either defining or defining-using, an INSERT query is defining, and a SELECT query is using. They also provide an algorithm to construct *database interaction control flow graph* (DICFG) for a DBA to calculate the family of test adequacy metrics. Therefore, given a DICFG, one can produce test cases using dataflow coverage metrics by DICFG. In their work, they assume that one simple query is embedded in the functionality being tested.

Suarez-Cabal and Tuya [SUA04] proposes a method for measuring the coverage of a simple SELECT query based on the condition coverage concept. They create a tree structure, called *coverage tree*, in which each level represents a predicate of the query, and evaluation of this coverage tree is a measure whether a given database does a good job of exercising the query.

Emmi et al. [EMM07] describes an algorithm for automatically generating input data for a DBA, as well as database instance generation to systematically explore all paths of the program, including the ones whose execution are depending on the result of a query.

Therefore, they cover TCG, TEOV, and TDP subproblems in this paper. We believe that this approach is complementary to what is presented in this thesis, and we will explain it in more detail in Chapter 6.

3.2 Test Execution and Output Verification

To the best of our knowledge, there are not many studies in the literature targeted towards Test Execution and Output Verification subproblem. Recall that for a DBA, output space O consists of the actual output of the functionality under test, plus the modified database state. Output of the functionality under test can be verified by the traditional approach; comparing the actual output with the expected output. However, unless there is a formal specification of the intended behavior of the application, it is not possible to fully automate the verification of modified database state [CHA04]. Below, we will explain the work regarding TEOV subproblem that we have encountered.

In [CHA04], two components are responsible for validation. The Output Validator is responsible for validating the result set of a SELECT query, and State Validator is responsible for validating the modified database state if the executed query is INSERT, UPDATE or DELETE. State validator saves the old and new values of the attributes of the tuples modified by the query to a log table using active rules in PostgreSQL that acts like triggers. Each test template has a precondition and post condition. Preconditions and postconditions are used to generate constraints that check:

- If tables that should not have changed did not change, and to check if tables that should have changed did change, and
- If tables changed in a correct way, according to constraints generated, and
- If the new state satisfies the relevant constraints specified by the tester and defined by the schema and application.

Output Validator submits a query, which is generated from the application query, to application tables to capture affected rows, because triggers/rules do not act upon the execution of a SELECT query. The result set of this query is recorded to log tables, and from this point on, Output Validator acts the same as State Validator. This approach is enhanced in [DEN05], to handle assertions (although no DBMS support it) and to check if a transaction brings the database in a consistent state.

In [CHR06], authors propose an approach to decrease the test suite execution time and reduce the number of lines of test code. The existing unit-test frameworks make the central assumption that all test methods must be independent. However, authors state that dependencies between test methods can be advantageous especially for database applications. They exploit these dependencies by ordering test cases to make building and maintaining test cases less labor intensive and decrease the time to execute test cases and test suites. In this approach, test cases and their dependencies are represented by a directed graph. If this graph is acyclic, then a topological sort defines an ordering in which the test cases can be executed.

Another approach for test case execution is presented in [KOS05], where a regression framework for DBAs is defined. The author's goal is to control the database state during testing and ordering the test cases in such a way that the number of resets to the database state is as few as possible, because resetting is an expensive operation. This approach is extended in [HAF05] to allow test cases to be executed in parallel for shared-database and shared-nothing architectures. As described in [HAF05], in shared-nothing architecture, there are N separate and independent installations of the application under test and its underlying database. Since applications do not share state, they do not interfere with each other. In shared-database architecture, there is one installation of application under test and the underlying database. Concurrent test runs interfere in this architecture because they may update the same database.

3.3 Test Data Preparation

The TDP subproblem for testing DBAs consists of assigning values for program variables as well as generating database instance(s) such that a test case can be executed on them. In Section 3.3.1, we will explain the previous work done on assigning values for program variables, and in Section 3.3.2, we will review the existing methods that generates database instances for testing.

3.3.1 TDP in terms of Program Value Instantiation

In Section 3.1, we briefly discussed the test case generation method of AGENDA [CHA04]. We now explain the Input Generator component, which is responsible for instantiating the input host variables of test cases. Input generation is loosely based on the category partition method [OST88], where a tester provides suggested values partitioned into different categories, which is referred to as *sample-values files*, and these values are selected to instantiate the input host variables by considering different heuristics which the tester provides. In [CHA04], the heuristics defined are *boundary values*, *duplicates from Application DB state*, *nulls*, *all groups* and *all templates*. The tester might select more than one heuristic, and depending on the heuristics that the tester selects, Input Generator chooses values from sample-values files and assigns them to input host variables. A complete description of the heuristics they utilize can be found in [CHA02].

Authors of [CHA04] realize that values assigned to input host variables by the method described in their paper often return empty result sets. In their following paper [DEN05], a test case is now a function written in a general purpose programming language, which has simple embedded queries in it. They aim to solve the empty result set problem by only selecting test cases (with their input host variables instantiated) that can be executed on the current database generated. They define *type A* and *type B* test cases, where a *type A* test case causes the WHERE clause to be true for some tuple for

each query executed by the test case and causes the transaction related to the test case to commit. Other test cases are designated by *type B*. It is impossible to statically determine whether a test case is *type A* or not, but they use a heuristic to try to select *type A* test cases. According to this heuristic, values for some input host variables are selected independently, and others are found by generating and executing queries involving the independently selected values.

In [CHA08], the approach to find *type A* test cases are replaced by a new approach. In this approach, authors generate a query, which retrieves data from the database (containing the information from sample-values files) by cross joining all possible values for the corresponding attributes of input host variables used in a (simple) query of the test case, and selecting values which resulted in the successful execution (returning non-empty result set) of the query. However, computing Cartesian products is expensive, especially if the number of attributes used in the test case is large and although they use this method to handle test cases with multiple queries, interactions and dependencies between them are not considered.

[TSA90] considers all or part of the DBA's requirement is expressed as a single relational algebra query, and generates test cases for black box testing automatically from this relational algebra query. Their approach is based on domain testing theory [WHI78] [WHI80] [CLA82], in which the idea is faults in the code are more likely to be found by test values chosen near appropriately defined program input and output domain boundaries. First, the query is converted to a linear predicate (which we refer as a logical formula $L(q)$ for a query q). Then, this predicate is converted to systems of linear inequalities (SSLI), solutions to which are exactly the points satisfying the predicate and test cases are generated by finding values inside and outside the boundary of the geometrical form defined by SSLI.

3.3.2 Generating Database Instances for Testing

One approach to obtain a database instance for testing DBAs is to use real data (data actually stored in the database of the application). First drawback of this approach is there might not be any data in the application database yet, because the application is not ready to use. The second drawback, as described by [CHA03], is that the real data might not reflect variety of situations that could occur. Therefore, testing is limited to the situations that could occur in the current DB state. Even if the data has variety of interesting situations for the tester, it is difficult to find them, and difficult to determine the appropriate user inputs and application outputs. Also, privacy and security issues might arise if the data is of critical importance. Therefore, we believe that using generated data should be preferred for testing.

There are also tools and methods [IBM09] [DTM09] [BRU05] [HOU06] [STE04], generating a database instance given the schema based on statistical distribution of the values of attributes or value ranges, however, these methods and tools are not query-aware. Thus, although they are suitable for testing performance of a DBMS component, they do not provide a suitable database instance to run the test cases.

In the AGENDA tool [CHA02] [CHA04] [DEN05] [CHA08], the database instance generation method is loosely based on category partition method presented in [OST88]. The database instance generation is done by State Generator component. The sample-values files (also used by Input Generator) and the database schema are used by State Generator to populate the database. Similar to input generation, state generation is also guided by heuristics from the tester. The available heuristics for state generation are *boundary values*, *duplicates*, *nulls* and *all groups*. Details for these heuristics can be found in [CHA02]. Recall from Section 3.1 that query-awareness is tried to be established in these methods by selecting the tests which can be executed on the

generated database. However, we believe that test cases should be selected independently from database instance, and the database instance should be generated so that tester can run the test case on the generated database instance.

In [BIN07], authors develop a new technique called Reverse Query Processing (RQP) to generate database instances. Given a SELECT query q, a database schema S, and a relation R which is the result set of q, they find a database instance D such that $R = q(D)$, where $q(D)$ denotes the execution of q on D. Note that D is not unique, and the aim is finding just one D that satisfies $R = q(D)$. In [BIN08], they further extend this approach to handle multiple SELECT queries. More formally, given a set of arbitrary SELECT queries $Q = \{q_1, q_2, ..., q_n\}$, and a set of expected results $R = \{R_1, R_2, ..., R_n\}$ of these queries, as well as the schema S, they find a database instance D such that $R_i = q_i(D)$ is valid for all $1 \leq i \leq n$, and D is consistent with respect to the ICs defined by S. The ICs are limited to primary key and uniqueness constraints. This approach is called Multi-RQP (MRQP). Since RQP is not decidable for arbitrary SQL queries, MRQP is not decidable either. Moreover, they assume that queries in Q are disjoint (non-conflicting in our case). Therefore, they generate a D_i for each q_i and take the union of them to obtain D. In this thesis, we stress on the cases where queries in Q are not disjoint.

Given a single SQL statement and a single property, [ZHA01] creates a database instance such that the result of executing the query on the database satisfies the property. They develop a tool, which takes the schema of the database, including the desired number of rows for each table, the query and the property as inputs, and returns a set of constraints to be given to a CST. They refrain from giving a formal language to specify properties, and they use two properties in their examples.

However, in a realistic setting, DBA will consist of multiple queries, and there might be several integrity constraints enforced by the schema. Our method extends the method

described in [ZHA01] in following ways:

1. We consider multiple queries and the interactions among them.
2. In [ZHA01], only SELECT queries are considered. We consider all four queries in a DBA under test.
3. In addition to primary and foreign key constraints, we consider all the integrity constraints except assertions in our solution to create a consistent database instance.
4. We consider more complex queries, such as nested queries using [NOT] IN, [NOT] EXISTS, ANY and ALL keywords.

In addition, as discussed in [ZHA01], when generating a database instance, tester might want to assign a property to each query, which we already discussed in Section 2.1. The method discussed in [ZHA01] generates a database instance such that the result of executing the query satisfies the property. We extend the use of the notion of property in our method as follows: Suppose that $P = \{p_1, p_2, \ldots, p_n\}$ is the set of properties, where $\forall i$, $p_i \in$ {EXISTS, NOT EXISTS} is the property assigned to q_i. We say p_i is automatically satisfied by D_{i-1} if q_i = INSERT. Moreover, if there is some T such that T causes the WHERE clause of q_i to evaluate to *true* and if p_i = EXISTS, or if there is no T such that T causes the WHERE clause of q_i to evaluate to *true* and if p_i = NOT EXISTS, then we say p_i is satisfied by D_{i-1}. In all other cases, p_i is not satisfied by D_{i-1}. The motivation behind a NOT EXISTS property is to allow the tester to force the result set of a query to be empty. This is particularly useful if a path in a test case can only be entered if a query returns an empty result set.

None of the related works explained above consider the interactions between multiple queries nor do they generate query-aware database instances for nested queries. In this thesis, we will take both of these aspects into account when we generate database instances.

Chapter Four

Generating Tuples for a Sequence of Simple Queries

In this chapter, we aim to present a method for solving the database instance generation problem where the given test suite is composed of a single test case consisting a sequence of simple SQL queries. First, we will give a grammar to formally state what we mean by simple queries. Then, we enumerate the conflicts that can occur between a pair of queries, and provide algorithms to detect and resolve them. Finally, we incorporate integrity constraints to our proposed method, and illustrate the method with an example. In the next chapter, we will extend this method by considering a test suite composed of a sequence of test cases where a test case is a sequence of simple or nested queries using keywords IN, NOT IN, EXISTS, NOT EXISTS, ANY and ALL.

4.1 Grammar for the Simple Queries

Figure 4.1 illustrates the grammar for the simple queries. Note that we omit in particular the following from SQL-92 to construct the grammar shown in Figure 4.1:

- GROUP BY and HAVING clauses
- CHARACTER, CHARACTER VARYING, CHARACTER LARGE OBJECT, BINARY LARGE OBJECT, DATE, TIME, TIMESTAMP and INTERVAL data types
- All ANSI functions, except the aggregate functions SUM, COUNT, MIN, MAX, AVG if they are used in <attribute_list>

```
<sel_simple> := SELECT <attribute_list> FROM <table_list>
                WHERE <where_simple>

<del_query>* := DELETE FROM <table_name>
              WHERE <where_simple>

<upd_query>* := UPDATE <table_name>
              SET <upd_cond_list>
              WHERE <where_simple>

<ins_query> := INSERT INTO <table_name> VALUES (<value_list>)

<table_name> := <alphanumeric>

<table_alias> := <alphanumeric>

<where_simple> := <predicate> | <predicate> <lop> <where_simple> | ε

<upd_cond_list> := <upd_condition> | <upd_condition> , <upd_cond_list>

<upd_condition> := <attribute> = <value>

<value_list> := <value> | <value> , <value_list>

<predicate> := <attribute> <rop> <value> |
             <attribute> <rop> <attribute>

<attribute> := <attribute_name> | <table_alias>.<attribute_name>

<lop> := AND | OR

<rop> := < | > | <= | >= | = | <>

<value> :=  <alphanumeric> | <numeric>

<attribute_name> := { ∑⁺ | ∑ = {A, B,…, Z, a, b,…, z, 0, 1,…, 9}}

<alphanumeric> := {∑⁺ | ∑ = {A, B,…, Z, a, b,…, z, 0, 1,…, 9}}

<numeric> = { ∑⁺ | ∑ = {0, 1,…, 9}}

* In UPDATE and DELETE queries, table aliases are not used
```

Figure 4.1 Grammar for Simple Queries

As one can observe, <where_simple> consists of conjunction and/or disjunction of *predicates*, where a predicate consists of an attribute, a relational operator and another

attribute or a constant. Note that in database literature, simple queries in this thesis are often referred to as Select-Project-Join (SPJ) queries.

We will use the following DB schema in our examples.

```
CREATE TABLE Emp (
  eid INTEGER CHECK(eid >= 10000 AND eid <= 99999),
  name VARCHAR(20),
  age INTEGER CHECK(age >= 20 AND age <= 80),
  salary INTEGER CHECK(salary >= 1000 AND salary <= 12000),
  PRIMARY KEY(eid));

CREATE TABLE Dept (
  did INTEGER CHECK (did >= 100 AND did <= 999),
  dname VARCHAR(30),
  budget INTEGER CHECK(budget >= 0 AND budget <= 2000000),
  PRIMARY KEY(did));

CREATE TABLE Works (
  eid INTEGER CHECK (eid >= 10000 AND eid <= 99999),
  did INTEGER CHECK (did >= 100 AND did <= 999),
  months INTEGER CHECK (months >= 0 AND months <= 480),
  PRIMARY KEY(eid, did),
  FOREIGN KEY(eid) REFERENCES Emp (eid)
            ON UPDATE CASCADE ON DELETE CASCADE,
  FOREIGN KEY(did) REFERENCES Dept(did)
            ON UPDATE CASCADE ON DELETE CASCADE);
```

Figure 4.2. Example Database Schema Definition in SQL

Consider the simple query *q* given below, which retrieves data from *Emp* and *Works* tables, where *Emp* table stores the information about employees, and *Works* table captures the employment relationship between employees and departments.

```
SELECT E.eid, W.months

FROM Emp E, Works W

WHERE E.age > 50 AND E.eid = W.eid;
```

WHERE clause of *q* consists of two predicates in conjunction with each other. First predicate, *E.age* > 50 compares an attribute and a value, whereas the second predicate compares the attribute *E.eid* with the attribute *W.eid*. The latter predicate is also called a *join predicate* because it is used to join the relations *Emp* and *Works*. Table aliases *E* and *W* are used to clarify which attribute belongs to which table. However, in UPDATE, DELETE and INSERT queries, table aliases are not used because they affect a single table. As an example, consider the following UPDATE query:

```
UPDATE Emp
SET salary = salary * 1.03,
    age = age + 1;
```

The UPDATE query above raises the salary of all employees by three percent and increments their age by 1. This query affects all the tuples in the *Emp* table because it does not have a WHERE clause, whose absence is denoted by ε in Figure 4.1. The following DELETE query deletes the departments in *Dept* relation whose budget is less than $100,000.

```
DELETE FROM Dept
WHERE budget < 100000
```

Lastly, the following INSERT query inserts a new employee to the database with name John, id 20000, age 35 and unknown salary.

```
INSERT INTO Emp(eid, name, age)
VALUES(20000, 'John', 35)
```

4.2 A Simplified Problem Statement

Given a test case $Q = q_1\ q_2\ ...\ q_n$, a database schema S, and a property sequence $P = p_1, p_2, ..., p_n$, we consider the problem of generating a database instance D such that executing every q_i in the order given by Q on D satisfies the corresponding property p_i, where $1 \leq i \leq n$, and D is consistent with the constraints given in S.

Our proposed method solves this problem by generating a constraint $L(Q)$, which is a logical formula generated for the test case Q such that solution of the unknowns in $L(Q)$ forms the D for Q. Our proposed method then gives $L(Q)$ to a constraint solver tool (*CST*) as an input, with the attributes as unknowns. Solutions of these unknowns governed by $L(Q)$ forms the required D.

Assumption 4.1. Domains of attributes used in queries are numeric. In Chapter 6, we will describe how to handle the attributes with string domain, but for simplicity of presentation, we will use attributes with numeric domains in our examples, because *CST*s support numeric data types only. In [EMM07], a *CST* with limited string constraint solving capabilities are discussed, which we explain in more detail in Chapter 6.

Assumption 4.2. Every numeric attribute should have a *domain constraint* associated with it. As noted earlier, domain constraints are used to restrict the values of an attribute. We make this assumption for two reasons. Firstly, the constraint solver tool that we use (HySat) requires the domain to be specified for each unknown. Secondly, we want to include the attributes used in <attribute_list> of the SELECT queries and the attributes used in the SET clause of an UPDATE query to the logical formulas that we generate. We explain the importance of this in Section 4.4.3. As we discuss in Chapter 6, if the domain constraint for an attribute is not needed or not known, the tester can create a dummy domain constraint by using the minimum and maximum values of the corresponding attribute's data type as lower and upper bounds.

Assumption 4.3. Input host variables used in queries are instantiated by the tester, and values of these variables conform to the domain of the associated attributes. In Chapter 6, instantiating host variables will be discussed in detail.

Assumption 4.4. Domain and table constraints are disjunction or conjunction of simple predicates. As discussed in Chapter 2, we will consider primary and foreign key constraints, uniqueness constraints, domain constraints, not null constraints and table constraints in this thesis.

Assumption 4.5. The assignment of values to input host variables or assignment of properties to queries by the tester will not be contradictory. Further, the values assigned to input host variables should not violate any integrity constraints.

4.3 Conflict Among Simple Queries

As stated earlier, even when a database instance is sufficient to run two or more queries in isolation, it may be insufficient when the queries in the test case are run in the given order due to the interaction between queries. To see this, consider the DB schema consisting of three tables in Figure 4.2. Note that as stated in Assumption 4.2, every attribute with numeric domain has an associated domain constraint.

Suppose the tester selected the queries with uninstantiated host variables (which are in the form :variable_name) as shown in Figure 4.3(a) where $\exists$ stands for EXISTS property and $\nexists$ stands for NOT EXISTS property. Further, suppose that the tester chose the values shown in 4.3(b) to instantiate the input host variables. Without loss of generality, we can describe the problem caused by interactions by only considering the interaction between q_1 and q_4.

Example 4.1. For now, suppose that $Q = q_1\ q_4$, $p_1 = \exists$ and $p_4 = \exists$. Also, suppose that we disregard the interaction between these two queries, and enhance the method

described by [ZHA01] to handle DELETE queries. As a result, a tuple for each query is obtained, values are assigned manually for each attribute having a string domain, and merged into *Emp* table as shown in Table 4.1.

```
q1 (∃): DELETE FROM Emp WHERE salary > :salary1
q2 (∃): DELETE FROM Emp WHERE age > :ageLimit1
q3 (∃): UPDATE Works SET months = months + 1
q4 (∃): SELECT E.eid FROM Emp E WHERE E.salary >   :salary2 AND E.age
          < :ageLimit2
q5 (∄): SELECT W.eid FROM Works W, Emp E WHERE E.eid=W.eid AND W.months
          < :months AND E.age > :ageLimit3
q6 (∄): SELECT * FROM Employee E WHERE E.salary <:salary3
```

Figure 4.3(a). Queries extracted from source code with uninstantiated host variables

```
q1 (∃): DELETE FROM Emp WHERE salary > 6000
q2 (∃): DELETE FROM Emp WHERE age > 65
q3 (∃): UPDATE Works SET months = months + 1
q4 (∃): SELECT E.eid FROM Emp E
        WHERE E.salary > 5500 AND E.age < 65
q5 (∄): SELECT W.eid FROM Works W, Emp E
       WHERE E.eid=W.eid AND E.salary < 5000 AND E.age > 60
q6 (∄): SELECT * FROM Emp E WHERE E.salary < 5700
```

Figure 4.3(b).Test case Q with instantiated host variables and the properties given by tester

Table 4.1. *Emp* table (Interaction is ignored)

Eid	*Name*	*Age*	*Salary*
9101	Susan	32	7000
5933	John	45	6500

The first tuple satisfies the predicate of q_1, and the second tuple satisfies the predicate of q_4. However, the second tuple also satisfies the predicate of q_1, and executing q_1

will delete both of these tuples in the *Emp* table. Therefore, *Emp* table cannot satisfy the property of q_4 anymore, which is $\exists$.

As illustrated in this example, interactions between queries and the intermediate database instances that result by executing each query have to be considered when generating a database instance. The example above illustrated only one type of conflict, which is between a DELETE and a SELECT query. However, a careful observation will reveal that there are other types of conflicts between queries as well. Below, we enumerate all the possible combinations between a pair of queries q_i and q_j in $Q = q_1 q_2 ... q_n$, $1 \leq i < j \leq n$, and determine if a conflict between them is possible. Notice that any possible combination of query types, their relative orders, and properties are covered in this enumeration:

1) Suppose type(q_i) = SELECT and $p_i = p_j = \exists$. There cannot be any conflicts, regardless of type(q_j), because SELECT queries do not modify any tuple.
2) Suppose type(q_i) = DELETE and $p_i = p_j = \exists$. If type(q_j) $\in$ {SELECT, UPDATE, DELETE}, a ***DELETE conflict*** might occur, which will be explained in more detail in Section 4.3.1. If type(q_j) = INSERT, we cannot have an DELETE conflict, because the tuple inserted by q_j does not exist in database yet, and we are not generating any tuples for INSERT queries, therefore q_i cannot affect it.
3) Suppose type(q_i) = UPDATE and $p_i = p_j = \exists$. If type(q_j) $\in$ {SELECT, UPDATE, DELETE}, an ***UPDATE conflict*** might occur, which will be explained in more detail in Section 4.3.3. If type(q_j) = INSERT, we cannot have an UPDATE conflict, because the tuple inserted by q_j does not exist in database yet, and we are not generating any tuples for INSERT queries, therefore q_i cannot affect it..
4) Suppose type(q_i) = INSERT and $p_i = p_j = \exists$. As stated above, we are not generating any tuples for INSERT queries, and tuple inserted by q_i cannot violate $p_j = \exists$. Therefore, no conflict can occur in this case.

5) Suppose $p_i = \nexists$ and $p_j = \exists$. In Section 4.3.2, we will state our assumption that only SELECT queries can have $\nexists$ property, therefore, type(q_i) = SELECT. If type(q_j) $\in$ {SELECT, UPDATE, DELETE}, then a ***NOT EXISTS conflict*** might occur, which will be explained in more detail in Section 4.3.2. If type(q_j) = INSERT, the inserted tuple might violate p_i, however, tester should deal with this case, as Assumption 4.5 states.
6) Suppose $p_i = \exists$ and $p_j = \nexists$. As described above, type(q_j) = SELECT. If type(q_i) = DELETE, we will not have any conflicts, because the tuples generated for q_i will be deleted by q_i, and cannot violate p_j. If type(q_i) = INSERT, p_j might be violated, however, avoiding this case is tester's responsibility as stated in Assumption 4.5. If type(q_i) $\in$ {SELECT, UPDATE} a ***NOT EXISTS conflict*** might occur, which will be explained in more detail in Section 4.3.2.

In the rest of this section, we will define the detection and resolution of DELETE, UPDATE and NOT EXISTS conflicts that might occur between queries in a test case.

4.3.1 DELETE Conflicts

The first type of conflict that we will consider is the DELETE conflicts as mentioned above. Suppose we have a DELETE query q_i and a SELECT query q_j, where $Q = q_1 q_2 \ldots q_n$, and $1 \leq i \leq n$, $1 \leq j \leq n$, $i \neq j$. A DELETE conflict occurs when we generate different tuples for q_i and q_j, however when we run the queries consecutively, q_i will delete the tuples that are generated for q_j.

Suppose we are given two queries, q_i and q_j, in $Q = q_1 q_2 \ldots q_n$ and suppose that $L(q_i)$ and $L^l(q_j)$ are obtained by the procedures ObtainLogicalFormula and Simplify. We say that q_i is in DELETE conflict with q_j when q_i might delete the tuples generated for q_j. Formally, q_i is in DELETE conflict with q_j denoted by $q_i \rightarrow_d q_j$, if all of the conditions below hold:

(1). type(q_i) = DELETE and type(q_j) $\in$ {SELECT, UPDATE, DELETE}.

(2). q_i is executed **before** q_j in the order given by Q

(3). $p_i = p_j = \exists$.

(4). $\exists L^r(q_i)$, $\exists L^s(q_j)$ such that each predicate in $L^r(q_i)$, using an attribute of a relation R that is also used in $L^s(q_j)$, intersects with a predicate in $L^s(q_j)$

Below, we give a pseudocode for the algorithm of the procedure DetectDELConflict which is described above. DetectDELConflict takes a pair of queries and their logical formulas as an input, determines if there is a DELETE conflict between them, and saves conflicting implicants to the conflict set *CS*. Every element in *CS* consists of two implicants conflicting with each other and the type of the conflict.

Algorithm 4.1 DetectDELConflict

```
procedure DetectDELConflict(L(q_i), L(q_j), CS)
   if (type(q_i)=DELETE and type(q_j)∈{SELECT, UPDATE, DELETE} and
      i<j and p_i= p_j=∃)then
      for each L^r(q_i) in L(q_i)do
         for each L^s(q_j) in L(q_j) do
            if(each predicate in L^r(q_i), using an attribute of a relation
             R that is also used in L^s(q_j), intersects with a predicate in
             L^s(q_j))
               CS = CS ∪ {L^r(q_i) →_d L^s(q_j)}
            endif
         endfor
      endfor
   endif
endprocedure
```

After a DELETE conflict is detected, the following algorithm is used to resolve the DELETE conflict between two implicants: The procedure ResolveConflict takes two conflicting implicants, along with the conflict type and the logical formula of the affected query. In a conflict $L^r(q_i) \rightarrow L^s(q_j)$, the *affected query* is q_j. When we discuss the other two conflict types in the following subsections, we will expand this algorithm to handle all three types of conflicts discussed in this chapter.

Algorithm 4.2 ResolveConflict

```
procedure ResolveConflict(L^r(q_i), L^s(q_j), conflictType, L(q_j))
    //The case of L^r(q_i) →_d L^s(q_j)
    if(conflictType = DELETE)then
       NEW ← ¬( L^r(q_i))
       NegationFree(NEW)
       UPDATED–L^s(q_j) = L^s(q_j) ∧ NEW
       convert UPDATED–L^s(q_j) into DNF
       delete L^s(q_j) from L(q_j)
    endif
    for each implicant L^k(q_j) in UPDATED–L^s(q_j) do
        Simplify(L^k(q_j), S)
        L(q_j) ← L(q_j) ∨ L^k(q_j)
    endfor
endprocedure
```

Example 4.2. Suppose that $Q = q_1\ q_4$, $p_1 = \exists$ and $p_4 = \exists$ as discussed in Example 4.1. We form $L(q_1)$ and $L(q_4)$ via procedures ObtainLogicalFormula and Simplify which yield $L(q_1)$ = (E.salary > 6000 ∧ E.salary ≤ 12000).

$L(q_4)$ = (E.salary > 5500 $\wedge$ E.salary $\leq$ 12000 $\wedge$ E.age $\geq$ 20 $\wedge$ E.age < 65 $\wedge$ E.eid $\geq$ 10000 $\wedge$ E.eid $\leq$ 99999)

We then apply procedure DetectDELConflict given above to detect if these two queries are in conflict with each other. type(q_1) = DELETE and type(q_4) = SELECT so condition (1) is satisfied. q_1 is executed before q_4 and both queries have the property $\exists$ as shown in Figure 4.3(b), so the conditions (2) and (3) are satisfied as well.

$L^{11}(q_1)$ as well as $L^{12}(q_1)$ intersects with both $L^{11}(q_4)$ and $L^{12}(q_4)$, thus all the predicates in $L^1(q_1)$ intersect with at least one predicate in $L^1(q_4)$, which satisfies the last condition. Therefore, we conclude that $q_1 \rightarrow_d q_4$.

Since $L^1(q_1)$ conflicts with $L^1(q_4)$, when generating values of the attributes satisfying $L^1(q_4)$, we have to assure that $L^1(q_1)$ will not be satisfied by these values. Below, we illustrate the steps of ResolveConflict algorithm:

NEW = $\neg L^1(q_1)$ = $\neg$ (E.salary > 6000 $\wedge$ E.salary $\leq$ 12000)

NegationFree(NEW) returns NEW = (E.salary $\leq$ 6000 $\vee$ E.salary > 12000)

UPDATED-$L^1(q_4)$ = $L^1(q_4)$ $\wedge$ NEW =

(E.salary > 5500 $\wedge$ E.salary $\leq$ 12000 $\wedge$ E.age $\geq$ 20 $\wedge$ E.age < 65 $\wedge$ E.eid $\geq$ 10000 $\wedge$ E.eid $\leq$ 99999) $\wedge$ (E.salary $\leq$ 6000 $\vee$ E.salary > 12000)

After converting UPDATED-$L^1(q_4)$ into DNF, the following is obtained:

UPDATED-$L^1(q_4)$ =

(E.salary > 5500 $\wedge$ E.salary $\leq$ 12000 $\wedge$ E.age $\geq$ 20 $\wedge$ E.age < 65 $\wedge$ E.eid $\geq$ 10000 $\wedge$ E.eid $\leq$ 99999 $\wedge$ E.salary $\leq$ 6000)

$\vee$

(E.salary > 5500 $\wedge$ E.salary $\leq$ 12000 $\wedge$ E.age $\geq$ 20 $\wedge$ E.age < 65 $\wedge$ E.eid $\geq$ 10000 $\wedge$ E.eid $\leq$ 99999 $\wedge$ E.salary > 12000)

Clearly, the first implicant has insersecting predicates (e.g., E.salary ≤ 12000 and E.salary ≤ 6000) and the second implicant has contradicting predicates (e.g., E.salary ≤ 12000 and E.salary > 12000) which will cause the second implicant to be eliminated by the procedure Simplify.

After deleting $L^1(q_4)$ and simplifying implicants in UPDATED-$L^1(q_4)$, $L(q_4)$ becomes:

$L(q_4)$ = (E.salary > 5500 ∧ E.salary ≤ 6000 ∧ E.age ≥ 20 ∧ E.age < 65 ∧ E.eid ≥ 10000 ∧ E.eid ≤ 99999)

With that modification to the logical formula, we resolved the conflict between q_1 and q_4, since there are no implicants conflicting with each other anymore.

Example 4.3. In this example, we will illustrate the violation of condition (4) in DELETE conflict definition. Consider the following two queries, each of them having the property ∃:

```
(q1) DELETE FROM Emp
     WHERE salary > 5000 AND age > 50

(q2) SELECT E.eid
     FROM Emp E
     WHERE E.salary < 5500 OR E.age > 55
```

We form the logical formulas similar to the previous example. $L(q_1)$ has one implicant. However, $L(q_2)$ has two implicants, and notice that we only took the conjunction of the implicant itself and the domain constraints of the attributes used in that particular implicant. Below are the logical formulas before simplification:

$L(q_1)$ = (E.salary > 5000 ∧ E.age > 50 ∧ E.salary ≥ 1000 ∧ E.salary ≤ 12000 ∧ E.age ≥ 20 ∧ E.age ≤ 80)

$L(q_2)$ = (E.salary < 5500 ∧ E.eid ≥ 10000 ∧ E.eid ≤ 99999 ∧ E.salary ≥ 1000 ∧ E.salary ≤ 12000)

∨

(E.age > 55 ∧ E.eid ≥ 10000 ∧ E.eid ≤ 99999 ∧ E.age ≥ 20 ∧ E.age ≤ 80)

After simplifying the above formulas, we obtain the following:

$L(q_1)$ = (E.salary > 5000 ∧ E.salary ≤ 12000 ∧ E.age > 50 ∧ E.age ≤ 80)

$L(q_2)$ = (E.salary < 5500 ∧ E.salary ≥ 1000 ∧ E.eid ≥ 10000 ∧ E.eid ≤ 99999) ∨

(E.age > 55 ∧ E.age ≤ 80 ∧ E.eid ≥ 10000 ∧ E.eid ≤ 99999)

Clearly, conditions (1), (2) and (3) are satisfied. To see if condition (4) is satisfied, we have to check if each predicate in $L^1(q_1)$ intersects with at least one predicate in $L^1(q_2)$ or if each predicate in $L^1(q_1)$ intersects with at least one predicate in $L^2(q_2)$, which is the second implicant of $L(q_2)$.

$L^{11}(q_1)$ intersects with $L^{11}(q_2)$ and $L^{12}(q_2)$. However, $L^{12}(q_1)$ does not intersect with any predicate in $L^1(q_2)$, therefore, we skip the implicant $L^1(q_2)$. We observe that $L^{11}(q_1)$ does not intersect with any predicate in $L^2(q_2)$, therefore, we conclude that condition (4) is not satisfied and q_1 is not in DELETE conflict with q_2. This conclusion assures that q_1 will not DELETE any tuple we have generated for q_2.

4.3.2 ∄ (NOT EXISTS) Conflicts

The second type of conflict is the conflict occurring between a query having ∃ property and a query having ∄ property. A ∄ (NOT EXISTS) conflict occurs when a tuple generated for a query violates the ∄ property of another.

Assumption 4.6. Recall from Section 3.3.2 that the motivation behind $\nexists$ property is giving the tester an option to specify there should be no tuples in database instance satisfying the WHERE clause of a particular query. We need this property if a query is used in the condition of a control statement such as IF, FOR, WHILE etc. Since only SELECT queries return a result set, they are the ones used in control statement conditions. Therefore, we assume that $\nexists$ property is only associated with SELECT queries.

Suppose we have two queries q_i and q_j in a test case $Q = q_1 q_2 \ldots q_n$, $1 \leq i \leq n$, $1 \leq j \leq n$, $i \neq j$, q_i is a SELECT query having $\nexists$ property and q_j is a SELECT, UPDATE, or DELETE query having $\exists$ property. A $\nexists$ conflict occurs when the tuples generated for q_j satisfies the WHERE clause of q_i when q_i is executed. The following example illustrates a conflict of this type:

Example 4.4. Consider the following queries (q_4 and q_6 in Figure 4.3(b)), where $p_4 = \exists$ and $p_6 = \nexists$.

```
(q4) SELECT E.eid
     FROM Emp E
     WHERE E.salary > 5500 AND E.age < 65

(q6) SELECT *
     FROM Emp E
     WHERE E.salary < 5700
```

Suppose we generated a tuple T for q_4 where $T.salary = 5510$, $T.age = 60$ and $T.eid = 10001$. As one can observe, T satisfies the WHERE clause of q_4. However, even if we did not generate any tuples for q_6, T will also satisfy the WHERE clause of q_6, which violates $p_6 = \nexists$.

Suppose we are given two queries, q_i and q_j in a test case $Q = q_1 q_2 \ldots q_n$, $1 \leq i \leq n$, $1 \leq j \leq n$, $i \neq j$. We say that q_i is in $\nexists$ conflict with q_j when the tuples generated for q_j satisfies the WHERE clause of q_i when q_i is executed. Formally, q_i is in $\nexists$ conflict with q_j denoted by $q_i \rightarrow_n q_j$, if all of the conditions below hold:

(1). type(q_i) = SELECT, type(q_j) $\in$ {SELECT, UPDATE, DELETE}. If type(q_j) = DELETE, q_i is executed **before** q_j in the order given by Q

(2). $p_i = \nexists$, $p_j = \exists$

(3). $\exists L^r(q_i)$, $\exists L^s(q_j)$ such that each predicate in $L^r(q_i)$, using an attribute of a relation R that is also used in $L^s(q_j)$, intersects with a predicate in $L^s(q_j)$

Note that if type(q_j) = INSERT, inserted tuple might also violate the $\nexists$ property of another query. However, in Assumption 4.5, we have stated that values assigned to input host variables cannot be contradictory. Therefore, we did not include INSERT queries in the previous definition. Another thing to notice in the definition is q_j can be a DELETE query, but there is no conflict if q_i is executed **after** q_j. The reason is if q_j is a DELETE query and executed first, all the tuples generated for q_j will be deleted by q_j itself; therefore, there will be no conflicts.

Below, we give the algorithm DetectNEXConflict to detect the $\nexists$ conflicts. Notice that there are two differences between this algorithm and DetectDELConflict. The first difference is the first **if clause**, and the second difference is conflict is recorded as $\nexists$ conflict instead of DELETE conflict.

Algorithm 4.3 DetectNEXConflict

```
procedure DetectNEXConflict(L(qi), L(qj), CS)
   if (type(qi)=SELECT and pi=∄ and pj=∃ and (type(qj)∈{SELECT, UPDATE} or
                                        (type(qj)=DELETE and i<j)))
```

```
      for each L^r(q_i) in L(q_i) do
         for each L^s(q_j) in L(q_j) do
            if(each predicate in L^r(q_i),using an attribute of a relation R
               that is also used in L^s(q_j), intersects with a predicate in
               L^s(q_j))
               CS = CS ∪ {L^r(q_i) →_n L^s(q_j)}
            endif
         endfor
      endfor
   endif
endprocedure
```

Once we detect the ∄ conflict, resolving strategy is same as resolving DELETE conflicts. Therefore, we expand our general method to resolve conflicts with a slight difference in the **if clause.**

Algorithm 4.4 ResolveConflict

```
procedure ResolveConflict(L^r(q_i), L^s(q_j), conflictType, L(q_j))

 //The case of L^r(q_i)→_d L^s(q_j) and L^r(q_i)→_d L^s(q_j)
   if(conflictType = DELETE or conflictType = NOT EXISTS)then
      NEW ← ¬( L^r(q_i))
      NegationFree(NEW)
      UPDATED–L^s(q_j) = L^s(q_j) ∧ NEW
      convert UPDATED–L^s(q_j)into DNF
      delete L^s(q_j) from L(q_j)
   endif
   for each implicant L^k(q_j) in UPDATED–L^s(q_j) do
```

```
        Simplify(L^k(q_j), S)
        L(q_j) ← L(q_j) ∨ L^k(q_j)
    endfor
endprocedure
```

Example 4.5. Again, consider the following queries (q_4 and q_6 in Figure 4.3(b)), where $p_4 = \exists$ and $p_6 = \nexists$. We will show that $q_6 \rightarrow_n q_4$ by following the definition given above.

```
(q4) SELECT E.eid
     FROM Emp E
     WHERE E.salary > 5500 AND E.age < 65

(q6) SELECT *
     FROM Emp E
     WHERE E.salary < 5700
```

type(q_4) = SELECT and type(q_6) = SELECT and $p_4 = \exists$ and $p_6 = \nexists$, therefore, conditions (1) and (2) are satisfied. To check if condition (3) is satisfied or not, we first obtain the formulas via procedures ObtainLogicalFormula and Simplify which yield.

$L(q_6)$ = (E.salary < 5700 $\wedge$ E.salary $\leq$ 12000)

$L(q_4)$ = (E.salary > 5500 $\wedge$ E.salary $\leq$ 12000 $\wedge$ E.age $\geq$ 20 $\wedge$ E.age < 65 $\wedge$ E.eid $\geq$ 10000 $\wedge$ E.eid $\leq$ 99999)

We observe that $L^{11}(q_6)$ and $L^{12}(q_6)$ intersect with $L^{11}(q_4)$ and $L^{12}(q_4)$. Therefore, condition (3) is satisfied and we conclude that $q_6 \rightarrow_n q_4$.

We have to assure that tuples generated for q_4 will not violate the $\nexists$ property of q_6. In other words, values generated for the attributes of q_4 should not satisfy any implicant of $L(q_6)$. We know that $L^1(q_6)$ conflicts with $L^1(q_4)$, thus, we modify $L^1(q_4)$ as ResolveConflict algorithm suggests:

$L^1(q_4)$ = (E.salary > 5500 ∧ E.salary ≤ 12000 ∧ E.age ≥ 20 ∧ E.age < 65 ∧ E.eid ≥ 10000 ∧ E.eid ≤ 99999) ∧ ¬(E.salary < 5700 ∧ E.salary ≤ 12000)

After simplicifiation, we obtain the following:

$L^1(q_4)$ = (E.salary ≥ 5700 ∧ E.salary ≤ 12000 ∧ E.age ≥ 20 ∧ E.age < 65 ∧ E.eid ≥ 10000 ∧ E.eid ≤ 99999)

4.3.3 UPDATE Conflicts

Generating tuples for queries without considering the effect of UPDATE queries will lead to another type of conflict, which we call UPDATE conflicts. An UPDATE conflict occurs if the modification of a tuple violates the property of another. We illustrate three examples in this subsection. The first example resolves the UPDATE conflicts as we resolve DELETE and ∄ conflicts, and we argue that a different method should be applied to resolve UPDATE conflicts in this example, i.e., one that will allow UPDATE queries to modify tuples of other queries, and reflecting this modification into their logical formula. The second and third example illustrates the cases where ∃ and ∄ properties are violated because of an UPDATE conflict, and illustrates the approach we follow for handling UPDATE conflicts. Also, a definition of UPDATE conflicts and a general method for generating logical formulas considering UPDATE conflicts will be given.

Example 4.6 In some cases, a conflict between an UPDATE query and another query (except INSERT) can be handled similar to DELETE queries. Consider the two queries below:

```
(q1) UPDATE Works
     SET salary = salary * 1.1
     WHERE E.age > 55
```

```
(q2) SELECT E.eid
     FROM Emp E
     WHERE E.age > 50
```

We can handle this case similar to DELETE conflicts. In other words, we can generate tuples for q_2 which will not be retrieved by q_1. Therefore, the logical formula for q_2 is as follows:

$L(q_2)$ = E.age > 50 $\wedge$ E.age $\leq$ 55 $\wedge$ E.eid $\geq$ 10000 $\wedge$ E.eid $\leq$ 99999

However, one would like to avoid generating logical formula like this, because in test cases where there are several DELETE, UPDATE, and SELECT queries with $\nexists$ property, this approach might lead to contradicting logical formulas, which have no solution. Therefore, in our UPDATE conflict resolution method, we will allow UPDATE queries to modify the tuples generated for other queries.

Formally, we say that two queries q_i and q_j in a test case $Q = q_1 q_2 \ldots q_n, 1 \leq i \leq n, 1 \leq j \leq n$, are in UPDATE conflict, denoted by $q_i \rightarrow_u q_j$, if all of the following conditions are satisfied:

(1). type(q_i) = UPDATE and type(q_j) $\neq$ INSERT

(2). $p_i = \exists$ and $p_j = \exists$.

(3). q_i is executed before q_j or $q_i = q_j$ in the order given by Q.

(4). An attribute x that is used in the SET clause of q_i is used anywhere in q_j.

(5). $\exists L^r(q_i)$, $\exists L^s(q_j)$ such that each predicate in $L^r(q_i)$, using an attribute of a relation R that is also used in $L^s(q_j)$, intersects with a predicate in $L^s(q_j)$

The algorithm DetectUPDConflict is given below. Notice that automatically, an UPDATE query is in UPDATE conflict with itself. Again, the differences of this algorithm from the previous algorithms are the first **if clause** and recording the type of conflict to the conflict set.

Algorithm 4.5 DetectUPDConflict

```
procedure DetectUPDConflict(L(q_i), L(q_j), CS)
   if (type(q_i)=UPDATE and type(q_j)≠ INSERT and p_i= p_j=∃ and i<=j and
      an attribute x used in the SET clause of q_i is used anywhere in q_j)
      for each L^r(q_i) in L(q_i) do
         for each L^s(q_j) in L(q_j) do
            if(each predicate in L^r(q_i), using an attribute of a relation R
               that is also used in L^s(q_j),intersects with a predicate in
               L^s(q_j))
               CS = CS ∪ {L^r(q_i) →u L^s(q_j)}
            endif
         endfor
      endfor
   endif
endprocedure
```

As we discussed in the previous example, we allow UPDATE queries to modify tuples generated for another query if they are in conflict. However, we reflect this modification to the logical formula of the affected query, so that conflict detections between the affected query and other queries in Q can be done by considering the result of the update. In the following examples, we illustrate how an UPDATE query can violate an $\exists$ and $\nexists$ property respectively, and we proceed by extending ResolveConflict algorithm by handling UPDATE conflicts so that the algorithm can be used to handle all types of conflicts.

Example 4.7: To illustrate the conflict between an UPDATE query and another query, a simple example is given below. Note that the second query can be any query except INSERT query.

```
(q1) UPDATE Works
     SET months = months +1
(q2) SELECT W.eid
     FROM Works W
     WHERE W.months < 50
```

Suppose that we disregard the effect of q_1 on q_2 and these queries are run in the given order. The logical formula of q_2, will be the following:

$L(q_2)$ = W.months < 50 $\wedge$ W.eid $\geq$ 10000 $\wedge$ W.eid $\leq$ 99999

Suppose that we generate tuples for q_1 and q_2. Further, suppose that *CST* assigns the value 49 for *W.months* and the corresponding tuple having this value for *W.months* is referred to as *T*, which is an element of *Works* relation. If q_1 did not conflict with q_2, then *T* would be retrieved after running q_2. However, since q_1 will run first, it will change the value of *W.months* to 50. As one can observe, running these two queries in the given order leads to a conflict, since *T* has the value 50 for *months* attribute, and cannot be in the result set of q_2 anymore.

Suppose we are given that q_1 is in UPDATE conflict with q_2, i.e., $q_1 \rightarrow_u q_2$. One can foresee the effect of q_1 on q_2, and create $L(q_2)$ accordingly. Consider the $L(q_2)$ shown below:

$L(q_2)$ = (W.months + 1) < 50 $\wedge$ W.eid $\geq$ 10000 $\wedge$ W.eid $\leq$ 99999

Because the predicate (W.months + 1) < 50 violates the predicate definition given in this thesis, we apply algebraic manipulation to this predicate and obtain the following logical formula:

$L(q_2)$ = W.months < 49 $\wedge$ W.eid $\geq$ 10000 $\wedge$ W.eid $\leq$ 99999

In this case, maximum value that *CST* would assign to *W.months* is 48. Then, as one can observe, running these two queries in the given order will not create any problems.

Example 4.8: An UPDATE query can also cause a violation of a NOT EXISTS property of a query. Consider the following example:

```
(q1) UPDATE Emp
     SET age = age + 1

(q2) SELECT E.eid
     FROM Emp E
     WHERE E.age > 25 AND E.salary < 2500

(q3) SELECT E.eid
     FROM Emp E
     WHERE age >= 65
```

Suppose we are given that $p_3 = \nexists$ and $q_3 \rightarrow_u q_2$. Suppose a tuple is generated for q_1 and another tuple T is generated for q_2, satisfying the logical formula shown below:

$L(q_2)$ = E.age > 25 $\wedge$ E.age $\leq$ 80 $\wedge$ E.salary $\geq$ 1000 $\wedge$ E.salary < 2500 $\wedge \neg$ (E.age $\geq$ 65) $\wedge$ E.age $\leq$ 80

After converting this formula into DNF and applying Simplify, we obtain the following:

$L(q_2)$ = E.age > 25 $\wedge$ E.age < 65 $\wedge$ E.salary $\geq$ 1000 $\wedge$ E.salary < 2500

As one can observe, effect of q_1 is not reflected in this logical formula. Suppose the *CST* assigned the value 64 for the *age* attribute of *T*, i.e., *T.age* = 64. When these queries run in the given order, q_1 will update the age value of *T* to 65, i.e., *T*.age = 65 after executing q_1. q_2 will retrieve the *eid* of *T* as intended, however, when q_3 is executed, result set will not be empty, thus NOT EXISTS property is violated.

Now suppose we generated $L(q_2)$ by considering the effect of q_1. Similar to previous example, $L(q_2)$ will be the following:

$L(q_2)$ = E.age > 25 $\wedge$ (E.age + 1 $\leq$ 65) $\wedge$ E.salary $\geq$ 1000 $\wedge$ E.salary < 2500

= E.age > 25 $\wedge$ E.age $\leq$ 64 $\wedge$ E.salary $\geq$ 1000 $\wedge$ E.salary < 2500

As one can observe, update of the tuple generated for q_2 will not create any conflicts or violation of domain constraints.

We can also show an UPDATE query as the following:

(q_1) UPDATE <table_name>

SET $x_1 = f_1(x_1)$

$x_2 = f_2(x_2)$

.

.

.

$x_k = f_k(x_k)$

WHERE <where_simple>

Suppose that there is a query q_2 that uses one or more attributes that are used in the SET clause of q_1. For each x of this sort in q_2, if $f(x)$ increases the value of x, we replace x with the corresponding $f(x)$ from q_1 in the predicate determining the upper bound for x. Otherwise, we replace x with the corresponding $f(x)$ from q_1 in the predicate determining

the lower bound for x. For instance, in Example 4.7, the attribute *months* is used in the SET clause of q_1, which is also used in the WHERE clause of q_2. If we denote *months* by x_1, then $f_1(x_1) = x_1 + 1$. Thus, in $L(q_2)$ we replace *months* by *months* + 1, knowing the value we generated will be incremented by 1 by q_1.

Below, we give a general algorithm DetectAllConflicts that we use to identify all of the conflicts in a sequence of queries $Q = q_1\ q_2\ \ldots\ q_n$, and we extend the algorithm ResolveConflicts to handle UPDATE conflicts. For all conflicts in the conflict set *CS*, we call ResolveConflicts algorithm once by giving the conflicting implicants, conflict type and the logical formula of conflicting queries as an input, and logical formula of each affected query is modified with respect to the type of conflict it is involved. This proposed solution method will be explained in more detail in Section 4.5.

Algorithm 4.6 DetectAllConflicts

```
procedure DetectALLConflicts(Q)
   CS ← ∅
   for each qi in Q
      for each qj where qj is executed after qi in Q
         DetectDELConflict(L(qi), L(qj), CS)
         DetectNEXConflict(L(qi), L(qj), CS)
         DetectUPDConflict(L(qi), L(qj), CS)
      endfor
   endfor
endprocedure
```

Algorithm 4.7 ResolveConflicts

```
procedure ResolveConflicts(CS)
   for each conflict CFLT = (L^r(q_i) →_{d,n,u} L^l(q_j)) in CS do
      L^r(q_i) ← CFLT.getFirstImplicant()
      q_i ← The query L^r(q_i) is derived from
      L^s(q_j) ← CFLT.getSecondImplicant();
      q_j ← The query L^s(q_j) is derived from
      conflictType ← CFLT.getConflictType();
      //The case L^r(q_i) →_d L^s(q_j) and L^r(q_i) →_n L^s(q_j)
      if(conflictType ∈ {DELETE, NOT EXISTS}) then
           NEW ← ¬( L^r(q_i))
           NegationFree(NEW)
           UPDATED–L^s(q_j) = L^s(q_j) ∧ NEW
           convert UPDATED–L^s(q_j) into DNF
           delete L^s(q_j) from L(q_j)
      endif
      for each implicant L^k(q_j) in UPDATED–L^s(q_j) do
           Simplify(L^k(q_j), S)
           L(q_j) ← L(q_j) ∨ L^k(q_j)
      endfor
      //The case L^k(q_i) →_u L^l(q_j)
      if(conflictType = UPDATE) then
         for each attribute x in L^s(q_j) that is in the SET clause of q_i
            if(f(x) increases the value of x) then
               replace x in the predicate determining upper bound by f(x)
               else
```

```
                replace x in the predicate determining lower bound by f(x)
            endif
        endfor
    endif
endfor
endprocedure
```

Finally, AssignSubscripts algorithm is used to assign subscripts to the <attribute>s used in logical formulas. Subscripts are used to differentiate between the tuples generated for different implicants. Suppose that in a logical formula $L(q)$ subscripts are assigned to <attribute>s and the following is obtained:

$$L(q) = (E_3.\text{salary} > 5000 \wedge E_3.\text{salary} \leq 12000) \wedge (E_3.\text{eid} = W_0.\text{eid}) \wedge (E_3.\text{eid} \geq 10000 \wedge E_3.\text{eid} \leq 99999 \wedge W_0.\text{eid} \geq 10000 \wedge W_0.\text{eid} \leq 99999$$

This subscript assignment indicates that solution of $L(q)$ by *CST* will result in two tuples: one for *Emp* table and one for *Works* table. Also, one can infer that before $L(q)$ there are already three assignments made for *Emp*, and tuple generated from this logical formula will be the fourth tuple to be inserted to *Emp*, and the first tuple to be inserted to *Works*. We show the pseudocode of the algorithm we use below:

Algorithm 4.8 AssignSubscripts

```
procedure AssignSubscripts(Q,S)
//Let S = {R1, R2,...,Rm}
//Each counter keeps track of the number of tuples to be inserted to
//corresponding R
   initialize m counters G = {G1, G2,...,Gm} to 0
   for each q in Q do
```

```
      for each implicant L^k(q) in L(q) do
         for each relation R' referred to in L^k(q) do
         //Note that G' is the corresponding counter of R'
            assign subscript G' to R' used in L^k(q)
            give L^k(q) as an input to CST
            if (CST does not report contradiction) then
               G' ← G' + 1
            else
               remove L^k(q) from L(q)
            endif
         endfor
      endfor
   endfor
endprocedure
```

4.4 Expanding the Logical Formulas for Integrity Constraints

In database instance generation, one would like the data generated to satisfy all the integrity constraints given by the schema. The IC types that we will consider are the following: uniqueness and primary key constraint, foreign key constraint, not-null constraint, domain constraint, and table constraint. ICs over several tables (Assertions) are not considered in this thesis, which will be explained in more detail in this section.

A question that must be answered is whether modifying logical formulas with respect to integrity constraints can generate additional conflicts or not. Integrity constraints restrict the input domain of the attributes from which they can take values. For example, a domain constraint restricts the input domain of an attribute with an interval, a key constraint

prevents an attribute to take a value already taken by others, and so on. Therefore, no additional conflicts can occur in the presence of integrity constraints.

4.4.1 Uniqueness and Primary Key Constraints

Generating constraints for the primary key is already discussed in [ZHA01]. Consider the primary key *(PK) eid* of for *Emp* in Figure 4.1 and suppose we generate z tuples for *Emp* relation. Then, the logical formula $L(PK)$ that will be generated for this key will have the following form:

$L(PK) = \bigwedge \text{Emp}_i.\text{eid} \neq \text{Emp}_j.\text{eid}$, for $0 \leq i < j < z$.

If primary key consists of more than one attribute, constraint generated should ensure that no two tuples in the relation can have the same value for all attributes. Consider the primary key *(eid, did)* of *Works* in Figure 4.1 and suppose we generate z tuples for *Emp* relation. Then, logical formula $L(PK)$ that will be generated for this key is:

$L(PK) = \bigwedge ((\text{Works}_i.\text{eid} \neq \text{Works}_j.\text{eid}) \vee (\text{Works}_i.\text{did} \neq \text{Works}_j.\text{did}))$, for $0 \leq i < j < z$.

Suppose that we are given a sequence of queries Q, where logical formulas of the queries are modified as described in ResolveConflicts procedure to handle all the conflicts. Also, subscripts are assigned by AssignSubscripts procedure. The algorithm HandlePKConstraints generates a new logical formula $L(PK)$ which ensures that primary key constraints will not be violated:

Algorithm 4.9 HandlePKConstraints

```
procedure HandlePKConstraints(Q, S, G)
   //Recall that after AssignSubscripts executes, G is the set of counters
   //that indicates how many tuples are to be generated for each relation
    for each relation R' in S do
```

```
    //Note that i and j are the subscripts assigned by AssignSubscript,
    // and are not used to distinguish between the relations in S
    //Suppose G' is the value of the counter of R' in G
     for(int i=0; i<G';i++) do
        for(int j= i+1; j< G'; j++) do
           //Suppose x1,...,xe is the primary key of Rk
           L(PK) ← L(PK) ∧ ((R'i.x1 ≠ R'j.x1)∨ ... ∨ (R'i.xe = R'j.xe))
        endfor
     endfor
  endfor
endprocedure
```

In our proposed method, for every tuple to be generated, we will include the primary key in the constraint we input to *CST*, because primary keys cannot be null and generate constraints for ensuring the uniqueness of each primary key for each tuple. However, for attributes having uniqueness constraints, we will not add any attribute if they are not already present in a logical formula, and generate constraints for ensuring the uniqueness of already existing attributes, because null values do not violate uniqueness constraints.

4.4.2 Foreign Key Constraints

In our example, *Works* relation has two foreign keys: *eid* referring to *Emp* and *did* referring to *Dept*. Consider the foreign key (*FK*) *eid*, and suppose that we want to generate foreign key constraints for a tuple of *Works*, and we generate z tuples for *Emp* relation. Then, as described in [ZHA01], the logical formula $L(FK)$ to ensure referential integrity between *Works* and *Emp* is:

$Works_i.\text{eid} = \text{Emp}_1.\text{eid} \vee Works_i.\text{eid} = \text{Emp}_2.\text{eid} \vee \ldots \vee Works_i.\text{eid} = \text{Emp}_z.\text{eid}$

Note that this constraint should be enforced for every tuple generated for *Works* relation.

Algorithm 4.10 HandleFKConstraints

```
procedure HandleFKConstraints(Q,S, G)
   //Recall that after AssignSubscripts executes, G is the set of counters
   //that indicates how many tuples are to be generated for each relation
   for each relation R' having a foreign key constraint do
      //Note that i and j are the subscripts assigned by AssignSubscripts
      //procedure, and are not used to distinguish the relations in S
      //Suppose G' is the value of the counter of R' in G
      for each foreign key constraint FK in R'
      //Let F be the relation referenced by FK, and G'' be the value of the
      //counter of F in G
         for(int i=0; i< G' ; i++)
         //Suppose x1,...,xe is the foreign key of R'
           L(FK) ← L(FK) ∧ (R'i.x1 = F1.x1 ∧ ... ∧ R'i.xe = F1.xe) ∨
                   (R'i.x1 = F2.x1 ∧ ... ∧ R'i.xe = F2.xe) ∨ ... ∨
                   (R'i.x1 = FG''.x1 ∧ ... ∧ R'i.xe = FG''.xe)
         endfor
      endfor
   endfor
endprocedure
```

4.4.3 Domain Constraints and Not Null Constraints

This constraint can be specified in the schema in the following ways: a CHECK clause can be appended to the left of the attribute's type when creating a table, or a domain can be created with CREATE DOMAIN keyword. In our example, we will use the former to restrict the values of attributes. For example, consider the domain constraint that we

enforced on *months* attribute of *Works* relation. The restriction is that the employment duration cannot be negative, and cannot be longer than 40 years. This constraint can be represented as the following CHECK constraint:

CHECK(months>=0 AND months<=480).

Recall that in Assumption 4.2, we assumed that every numeric attribute should have a domain constraint associated with it. This assumption allows us to detect conflicts that could not be detected by the definitions given earlier. We will illustrate this case with an example.

Example 4.9. Consider the following queries q_1 and q_2 below, where $p_1 = p_2 = \exists$.

```
(q1) DELETE FROM Emp E WHERE E.age > 40

(q2) SELECT E.age
     FROM Emp E
     WHERE E.salary < 3000
```

By only observing the predicates in WHERE clauses of these queries, one can conclude that q_1 and q_2 are not in conflict. Suppose that we are generating tuple T for q_2. Salary of the employee represented by T will be less than 3000 and it will have an arbitrary age value. Note that we might want to generate a value for age attribute because of an IC such as NOT NULL, or because we want tester to retrieve meaningful data after executing queries. Suppose *CST* assigned 45 as a value for the age attribute of T and the queries are run in the given order. T will be deleted by q_1 and cannot be retrieved by q_2.

We cannot detect the possible conflict between these queries without Assumption 4.2, because when generating a logical formula for q_2, we use the domain constraints of the

attributes in the <attribute_list> of q_2. Now, conflict detection can be very easily done as described in Algorithm 4.1. The logical formula of q_2 becomes:

$$L(q_2) = \text{E.salary} < 3000 \wedge \text{E.age} \geq 20 \wedge \text{E.age} \leq 80 \wedge \neg(\text{E.age} > 40)$$
$$= \text{E.salary} < 3000 \wedge \text{E.age} \geq 20 \wedge \text{E.age} \leq 40$$

As one can observe, tuples generated by the constraint obtained from this logical formula will not be deleted by q_1. Handling Domain constraints has been automatically integrated to ObtainLogicalFormula procedure.

If an attribute x of a relation R has a not null constraint in S, then we add the domain constraint of x to every implicant referring to the relation that x belongs.

4.4.4 Constraints over a Single Table and Ternary Logic

Suppose that the designer of the database shown in Figure 4.2 wants to add a regulation, stating that no senior employee (i.e. older than 60 years old) can have a salary less than 3500. As a logical formula, this regulation can be represented as ($age > 60 \Rightarrow salary \geq 3500$), which is equivalent to ($\text{age} \leq 60 \vee \text{salary} \geq 3500$). This can be enforced by the following CHECK constraint: CHECK($\text{age} \leq 60 \vee \text{salary} \geq 3500$).

Consider a query with a logical expression in its WHERE clause. If the value of the logical expression evaluates to NULL for a tuple, then this tuple will not be used by this query. Therefore, in DML, a query will initiate an action (i.e., return, delete, modify) on a tuple only if the tuple makes the logical expression of the query to evaluate to *true*. However, behavior of the CHECK constraints are different than the WHERE clauses of queries. A CHECK constraint is violated only if the logical expression of the constraint evaluates to false, which implies that *unknown* truth value of a logical expression of a CHECK constraint is not considered as a violation by the DBMS. Two cases discussed above will be illustrated by the example below.

Example 4.10: Consider the logical expression L = (age < 40 $\wedge$ salary > 5000). Suppose L is the logical expression used in a WHERE clause of a SELECT query q. Also, suppose that there is a tuple T in the database where $T.age$ = NULL and T.salary = 6000. If we substitute the values, the logical expression becomes:

L' = (NULL < 40 $\wedge$ 6000 > 5000)
= (NULL $\wedge$ True)
= NULL

Therefore, since T caused the logical expression to evaluate to *unknown*, T will not be selected by q.

Now, suppose that L is the logical expression of a CHECK constraint, and we have executed an INSERT query q to insert the tuple T mentioned above. Since logical expression evaluated to *unknown*, it does not violate the CHECK constraint and T will be inserted to the database.

Exploiting Ternary Logic in Tuple Generation

In previous section, we have illustrated the effects of ternary logic in DML and CHECK constraints. These effects can be exploited when generating logical formulas for queries; in fact, we have already made use of Ternary Logic in DML. Recall that in our conflict definitions, we used the following condition: "All the predicates in $L^r(q_i)$ intersect with at least one predicate in $L^l(q_j)$." Consider the example below where this condition is not satisfied.

Example 4.11: Consider the two queries below, where $p_1 = p_2 = \exists$. We disregard augmenting the logical formulas with the domain constraints for this example.

```
(q1) DELETE FROM Emp
```

```
        WHERE salary > 5000 AND age > 55

(q2)    SELECT E.eid
        FROM Emp E
        WHERE E.salary < 5500
```

Notice that the condition mentioned above is not violated, because the predicate (*age* > 55) does not intersect with any predicate in $L^1(q_2)$. Therefore, these two queries are not in conflict. To illustrate how we make use of Ternary Logic, suppose that we generate a tuple T for q_2, where *T.eid* = 10001 and *T.salary* = 5400. Suppose we denote the WHERE clause of q_1 as L, and L' denotes the expression where values from T is substituted into L.

L = (salary > 5000 $\wedge$ age > 55)

L' = (5400 > 5000 $\wedge$ NULL > 55)

= (TRUE $\wedge$ NULL) = NULL

Recall that q_1 would delete T only if T would make the WHERE clause of q_1 evaluate to *true*. However, since L' evaluates to *unknown*, T will not be deleted.

We can make use of Ternary Logic in a similar way for table constraints. First, we describe the method we propose to handle table constraints. Then we illustrate the method with an example.

Consider a query q. Suppose we obtained the $L(q)$ by considering all the conflicts that q is in, and $L^k(q)$ is an implicant of $L(q)$, $1 \leq k \leq \lambda$, where λ is the number of implicants. In addition, suppose that B is a table constraint that is *enforced on* $L^k(q)$ (i.e., $L^k(q)$ contains an attribute that belongs to the table that B is defined in). We present HandleTableConstraints algorithm to handle these types of constraints. For each $L^k(q)$ and for each table constraint B that is *enforced on* $L^k(q)$, the algorithm does the following:

- First, it converts the logical formula of B to DNF. Call this $L(B)$.
- For each implicant $L^j(B)$ in $L(B)$, $1 \leq j \leq \lambda_B$, where λ_B is the number of implicants in $L(B)$ it checks if all the attributes used in $L^j(B)$ are also used in $L^k(q)$. If that is the case, then $L^k(q) = L^k(q) \wedge L^j(B)$. Otherwise, it does not modify $L^k(q)$.
- As the last step, it simplifies $L^k(q)$ by removing predicates that are subsumed or violate the domain constraints.

Algorithm 4.11 HandleTableConstraints

```
procedure HandleTableConstraints(Q,S)
   for each q in Q where p = ∃ do
      for each implicant L^k(q) do
         for each constraint B that is enforced on L^k(q) do
            convert L(B) into DNF
            for each implicant L^j(B) in B
               if(all attributes in L^j(B) are also used in L^k(q))
                  L^k(q) ← L^k(q) ∧ L^j(B);
                  Simplify(L^k(q),S);
               endif
            endfor
         endfor
      endfor
   endfor
endprocedure
```

The first example illustrates the case where we have to modify an implicant of a query. The second example illustrates the case where we make use of Ternary Logic.

Example 4.12. Consider the query q and the logical formula of the CHECK constraint B. Suppose we are given that $L(B)$ is enforced on q. Again, we disregard augmenting the logical formulas with the domain constraints..

```
(q) SELECT E.eid
    FROM Emp E
    WHERE E.salary < 5500 ∧ E.age > 50
```

$L(q)$ = (E.salary < 5500 ∧ E.age > 50)

$L(B)$ = (E.salary > 1000 ∧ E.age <70 ∧ E.age > 18)

$L(B)$ and $L(q)$ are already in DNF and both of them consist of one implicant, which are $L^1(B)$ and $L^1(q)$. Therefore, we check if all of the attributes used in $L^1(B)$ are also used in $L^1(q)$. It is easy to observe that *E.salary* and *E.age* are used in $L^1(q)$, therefore, we extend $L^1(q)$ as shown below:

$L^1(q)$ = (E.salary < 5500 ∧ E.age > 50) ∧ (E.salary > 1000 ∧ E.age <70 ∧ E.age > 18)

After simplification, $L^1(q)$ becomes the following:

$L^1(q)$ = (E.salary > 1000 ∧ E.salary < 5500 ∧ E.age > 50 ∧ E.age <70)

It is easy to observe that any tuple generated from this logical formula will satisfy the CHECK constraint B.

Example 4.13. Consider the query q and the logical formula of the CHECK constraint B. Suppose we are given that $L(B)$ is enforced on q. We also disregard augmenting the logical formulas with the domain constraints for this example.

```
(q) SELECT E.eid
    FROM Emp E
    WHERE E.salary < 5500
```

$L(q)$ = (E.salary < 5500)

$L(B)$ = (E.salary > 1000 $\wedge$ E.age <70 $\wedge$ E.age > 18)

According to our method, we should not modify $L^1(q)$ because it does not use all the attributes that $L(B)$ uses. Suppose we generated a tuple T for q without considering B. Further, suppose *T.eid* = 10002 and *T.salary* = 5200. The values for other attributes will be NULL, because they are not specified in $L^1(q)$. Suppose we substituted the values of T into $L(B)$, which we denote as $L'(B)$.

$L'(B)$= (5200 > 1000 $\wedge$ NULL<70 $\wedge$ NULL > 18)

= (TRUE $\wedge$ NULL $\wedge$ NULL) = NULL

Recall that since CHECK constraint evaluated to NULL, DBMS will not consider this case as a violation. Therefore, we exploit this property in our method to handle CHECK constraints. Note that converting both $L(q)$ and $L(B)$ to DNF is important. We do not want any disjunctions in our logical formulas, because after substitution, if (NULL $\vee$ TRUE) case occurs, we cannot exploit the Ternary Logic anymore.

4.4.5 Assertions

Although assertions are defined in ANSI SQL-99 [SQL99], using assertions for validity checks are proved to be difficult. When an assertion is created, and each time the database is modified, system checks if the assertion is valid. This creates a significant overhead, and at the time of writing this thesis, no major DBMS (Oracle, IBM DB2, MS SQL Server, PostgreSQL, MySQL, and so on) supports assertions. As stated in [DEN05], the reason why commercial DBMSs do not support assertions is not the lag between research and commercialization. Commercial DBMSs main priority is performance, and enforcing assertions is time consuming for most production quality DBMSs. Therefore, we will not consider assertions in this work.

4.5 Proposed Method

Given a sequence of simple queries $Q = q_1\ q_2 \ldots\ q_n$ with instantiated input host variables, the corresponding sequence of properties $P = p_1\ p_2\ \ldots p_n$, and the database schema S, the algorithm GenerateDatabaseInstance presented below generates the database instance D, such that D is consistent with the integrity constraints given in S and executing each query in the order given by Q will satisfy the corresponding property in P, as follows:

```
procedure GenerateDatabaseInstance(Q, P, S)
```

1) For each query q_i in Q, $1 \leq i \leq n$, obtain the logical formula $L(q_i)$ using the procedures ObtainLogicalFormula and Simplify given in Section 2.6.
2) Identify the conflicts among each pair of queries using DetectAllConflicts procedure given in Section 4.3.3. Recall that the set CS contains all the conflicting implicants after executing DetectAllConflicts.
3) Resolve all the conflicts in CS using ResolveConflicts procedure given in Section 4.3.3. Convert the logical formulas of the affected queries into DNF and simplify them using the procedure Simplify given in Section 2.6.
4) For each $L^j(q_i)$ of q_i in Q, where $p_i = \exists$, $1 \leq i \leq n$, $1 \leq j \leq \lambda_i$, modify each implicant being subject to one or more table constraints using HandleTableConstraints procedure given in Section 4.4.4. Then, convert these implicants into DNF, and simplify it using the procedure Simplify given in Section 2.6.
5) For each $L^j(q_i)$ of q_i in Q, where $p_i = \exists$, $1 \leq i \leq n$, $1 \leq j \leq \lambda_i$, suppose R_k is a relation used in $L^j(q_i)$, and X_k is the set of attributes defining the primary key of R_k. For each attribute $x \in X_k$, take conjunction of the domain constraint of x and $L^j(q_i)$. Also, suppose R_k' is a relation referenced by R_k with a foreign key constraint, and X_k' is the set of attributes defining the primary key of R_k'. For each attribute $x' \in X_k'$, take

conjunction of the domain constraint of x' and $L^j(q_i)$. Add the join predicates which joins X_k and X_k'.

6) For each q_i in Q, where $p_i = \exists$ and $1 \leq i \leq n$, assign subscripts to the <attribute>s in each implicant using AssignSubscripts procedure given in Section 4.3.3. Note that this procedure also removes implicants resulting in a contradiction.
7) Obtain the logical formula $L(PK)$ for ensuring primary key and uniqueness constraints using HandlePKConstraints procedure given in Section 4.4.1.
8) Obtain the logical formula L(FK) for ensuring foreign key constraints using HandleFKConstraints procedure given in Section 4.4.2 and simplify the modified implicants using the procedure Simplify given in Section 2.6.
9) Obtain the logical formula $L(Q)$ for the test case Q by taking the conjunction of the following:
   a. Each implicant $L^j(q_i)$ of q_i in Q, where $p_i = \exists$, $1 \leq i \leq n$, $1 \leq j \leq \lambda_i$
   b. $L(PK)$
   c. $L(FK)$
10) Solution of $L(Q)$ will produce the tuples to construct D_0. Recall that an attribute in $L(Q)$ is represented by <table_name>$_k$.<attribute_name>. A tuple T_k to be inserted to the table <table_name> consists of the solutions of each <table_name>$_k$.<attribute_name> in $L(Q)$, and D_0 consists of each T that can be obtained from $L(Q)$.

4.6 Example with ICs and Instance Generation

We will illustrate the proposed method described in Section 4.5 with an example. In our example, we will use the schema shown in Figure 4.4 which differs from the schema shown in Figure 4.2 in an additional integrity constraint over *Emp* table. The test case we will use will be the test case shown in Figure 4.3 (b), which we repeat in Figure 4.5.

```
CREATE TABLE Emp (
  eid INTEGER CHECK(eid >= 10000 AND eid <= 99999),
  name VARCHAR(20),
  age INTEGER CHECK(age >= 20 AND age <= 80),
  salary INTEGER CHECK(salary >= 1000 AND salary <= 12000),
  PRIMARY KEY(eid),
  CHECK(age <= 70 OR salary > 3500));

CREATE TABLE Dept (
  did INTEGER CHECK (did >= 100 AND did <= 999),
  dname VARCHAR(30),
  budget INTEGER CHECK(budget >= 0 AND budget <= 2000000),
  PRIMARY KEY(did));

CREATE TABLE Works (
  eid INTEGER CHECK (eid >= 10000 AND eid <= 99999),
  did INTEGER CHECK (did >= 100 AND did <= 999),
  months INTEGER CHECK (months >= 0 AND months <= 480),
  PRIMARY KEY(eid, did),
  FOREIGN KEY(eid) REFERENCES Emp (eid)
            ON UPDATE CASCADE ON DELETE CASCADE,
  FOREIGN KEY(did) REFERENCES Dept(did)
            ON UPDATE CASCADE ON DELETE CASCADE);
```

Figure 4.4. New Schema with an Additional IC

```
q1 (∃): DELETE FROM Emp WHERE salary > 6000
q2 (∃): DELETE FROM Emp WHERE age > 65
q3 (∃): UPDATE Works SET months = months + 1
q4 (∃): SELECT E.eid FROM Emp E
        WHERE E.salary > 5500 AND E.age < 65
q5 (∄): SELECT W.eid FROM Works W, Emp E
       WHERE E.eid=W.eid AND E.salary < 5000 AND E.age > 60
q6 (∄): SELECT * FROM Emp E WHERE E.salary < 5700
```

Figure 4.5. Test case Q.

In Step 1 of the proposed method given in Section 4.5, the logical formulas for each query in Q are formed where domain constraints are added, implicants containing contradicting predicates or predicates violating domain constraints are deleted and subsumed predicates are removed by ObtainLogicalFormula and Simplify procedures. Logical formulas for each query q in Q are shown below.

$L(q_1)$ = (E.salary > 6000) $\wedge$ (E.salary $\geq$ 1000 $\wedge$ E.salary $\leq$ 12000)

= (E.salary > 6000 $\wedge$ E.salary $\leq$ 12000)

$L(q_2)$ = (E.age > 65) $\wedge$ (E.age $\geq$ 20 $\wedge$ E.age $\leq$ 80)

= (E.age > 65 $\wedge$ E.age $\leq$ 80)

$L(q_3)$ = (W.months $\geq$ 0 $\wedge$ W.months $\leq$ 480)

$L(q_4)$ = (E.salary > 5500 $\wedge$ E.age < 65) $\wedge$ (E.salary $\geq$ 1000 $\wedge$ E.salary $\leq$ 12000) $\wedge$ (E.age $\geq$ 20 $\wedge$ E.age $\leq$ 80) $\wedge$ (E.eid $\geq$ 10000 $\wedge$ E.eid $\leq$ 99999)

= (E.salary > 5500 $\wedge$ E.salary $\leq$ 12000 $\wedge$ E.age $\geq$ 20 $\wedge$ E.age < 65 $\wedge$ E.eid $\geq$ 10000 $\wedge$ E.eid $\leq$ 99999)

$L(q_5)$ = (E.eid = W.eid) $\wedge$ (E.salary < 5000) $\wedge$ (E.age > 60) $\wedge$ (E.age $\geq$ 20 $\wedge$ E.age $\leq$ 80) $\wedge$ (E.eid $\geq$ 10000 $\wedge$ E.eid $\leq$ 99999) $\wedge$ (W.eid $\geq$ 10000 $\wedge$ W.eid $\leq$ 99999) $\wedge$ (E.salary $\geq$ 1000 $\wedge$ E.salary $\leq$ 12000)

= (E.eid = W.eid $\wedge$ E.salary $\geq$ 1000 $\wedge$ E.salary < 5000 $\wedge$ E.age > 60 $\wedge$ E.age $\leq$ 80 $\wedge$ E.eid $\geq$ 10000 $\wedge$ E.eid $\leq$ 99999 $\wedge$ W.eid $\geq$ 10000 $\wedge$ W.eid $\leq$ 99999)

$L(q_6)$ = (E.salary < 5700) $\wedge$ (E.salary $\geq$ 1000 $\wedge$ E.salary $\leq$ 12000)

= (E.salary $\geq$ 1000 $\wedge$ E.salary < 5700)

In Step 2, DetectAllConflicts procedure identifies the conflicts and saves them into conflict set *CS*. *CS* obtained by DetectAllConflicts procedure and the justification for each conflict are given below.

$CS = \{L^1(q_1) \rightarrow_d L^1(q_4), L^1(q_5) \rightarrow_n L^1(q_4), L^1(q_6) \rightarrow_n L^1(q_4)\}$

type(q_1) = DELETE, type(q_4) = SELECT. q_1 is executed before q_4 and $p_1 = p_4 = \exists$. Therefore, DetectDELConflict procedure, which is called by DetectAllConflicts procedure, compares $L^1(q_1)$ and $L^1(q_4)$, and determines that E.salary > 6000 of $L^1(q_1)$ intersects with E.salary > 5500 of $L^1(q_4)$, and E.salary $\leq$ 12000 of $L^1(q_1)$ intersects with E.salary > 5500 and E.salary $\leq$ 12000 of $L^1(q_4)$. Therefore, DetectDELConflict procedure detects a DELETE conflict between $L^1(q_1)$ and $L^1(q_4)$ and saves this conflict into *CS*.

type(q_5) = SELECT, $p_5 = \nexists$, $p_4 = \exists$ and type(q_4) = SELECT. The condition in the first if clause of DetectNEXConflict procedure returns true and the procedure compares the implicants in these two queries to determine if there is a $\nexists$ conflict. Since *Emp* is the only relation used in $L^1(q_4)$, procedure will check if the predicates in $L^1(q_5)$ using an attribute of *Emp* intersect with at least one predicate in $L^1(q_4)$. DetectNEXConflict procedure detects a $\nexists$ conflict and saves this conflict into *CS*.

Similar to above, type(q_6) = SELECT, $p_6 = \nexists$, $p_4 = \exists$ and type(q_4) = SELECT. The condition in the first if clause of DetectNEXConflict procedure returns true and the procedure compares the implicants in these two queries to determine if there is a $\nexists$ conflict. Every predicate in $L^1(q_6)$ intersects with at least one predicate in $L^1(q_4)$, therefore, DetectNEXConflict procedure saves this conflict in *CS*.

In Step 3, first the logical formulas of the affected queries in CS are modified to resolve the conflicts. q_4 is the only affected query, and $L^1(q_4)$ is the only implicant of q_4. Therefore, ResolveConflicts procedure modifies $L(q_4)$ using the following steps:

NEW = $\neg L^1(q_1) \wedge \neg L^1(q_5) \wedge \neg L^1(q_6)$

= $\neg$(E.salary > 6000 $\wedge$ E.salary $\leq$ 12000) $\wedge$ $\neg$(E.eid = W.eid $\wedge$ E.salary $\geq$ 1000 $\wedge$ E.salary < 5000 $\wedge$ E.age > 60 $\wedge$ E.age $\leq$ 80 $\wedge$ E.eid $\geq$ 10000 $\wedge$ E.eid $\leq$ 99999 $\wedge$ W.eid $\geq$ 10000 $\wedge$ W.eid $\leq$ 99999) $\wedge$ $\neg$(E.salary $\geq$ 1000 $\wedge$ E.salary < 5700)

When the procedure NegationFree is applied to NEW, we obtain the following:

NEW = (E.salary $\leq$ 6000 $\vee$ E.salary > 12000) $\wedge$ (E.eid = W.eid $\wedge$ E.eid $\geq$ 10000 $\wedge$ E.eid $\leq$ 99999 $\wedge$ W.eid $\geq$ 10000 $\wedge$ W.eid $\leq$ 99999) $\wedge$ (E.salary < 1000 $\vee$ E.salary $\geq$ 5000 $\vee$ E.age $\leq$ 60 $\vee$ E.age > 80) $\wedge$ (E.salary < 1000 $\vee$ E.salary $\geq$ 5700)

Note that the join predicate E.eid = W.eid and domain constraints of E.eid and W.eid are taken out from the negation as NegationFree procedure dictates.

We obtain UPDATED- $L^1(q_4)$ by conjuncting $L^1(q_4)$ and NEW:

UPDATED- $L^1(q_4)$ = (E.salary > 5500 $\wedge$ E.salary $\leq$ 12000 $\wedge$ E.age $\geq$ 20 $\wedge$ E.age < 65 $\wedge$ E.eid $\geq$ 10000 $\wedge$ E.eid $\leq$ 99999) $\wedge$ (E.salary $\leq$ 6000 $\vee$ E.salary > 12000) $\wedge$ (E.eid = W.eid $\wedge$ E.eid $\geq$ 10000 $\wedge$ E.eid $\leq$ 99999 $\wedge$ W.eid $\geq$ 10000 $\wedge$ W.eid $\leq$ 99999) $\wedge$

(E.salary < 1000 ∨ E.salary ≥ 5000 ∨ E.age ≤ 60 ∨ E.age > 80) ∧ (E.salary < 1000 ∨ E.salary ≥ 5700)

We can rearrange UPDATED- $L^1(q_4)$ given above and group (E.salary > 5500 ∧ E.salary ≤ 12000 ∧ E.age ≥ 20 ∧ E.age < 65 ∧ E.eid ≥ 10000 ∧ E.eid ≤ 99999) and (E.eid = W.eid ∧ E.eid ≥ 10000 ∧ E.eid ≤ 99999 ∧ W.eid ≥ 10000 ∧ W.eid ≤ 99999) in a common parenthesis to obtain the following formula:

UPDATED- $L^1(q_4)$ = (E.salary > 5500 ∧ E.salary ≤ 12000 ∧ E.age ≥ 20 ∧ E.age < 65 ∧ E.eid = W.eid ∧ E.eid ≥ 10000 ∧ E.eid ≤ 99999 ∧ W.eid ≥ 10000 ∧ W.eid ≤ 99999) ∧
(E.salary ≤ 6000 ∨ E.salary > 12000) ∧
(E.salary < 1000 ∨ E.salary ≥ 5000 ∨ E.age ≤ 60 ∨ E.age > 80) ∧
(E.salary < 1000 ∨ E.salary ≥ 5700)

ResolveConflict procedure converts UPDATED- $L^1(q_4)$ into DNF. After this step, UPDATED- $L^1(q_4)$ consists of the following implicants:

UPDATED- $L^{1\text{-}1}(q_4)$ = (E.salary > 5500 ∧ E.salary ≤ 12000 ∧ E.age ≥ 20 ∧ E.age < 65 ∧ E.eid = W.eid ∧ E.eid ≥ 10000 ∧ E.eid ≤ 99999 ∧ W.eid ≥ 10000 ∧ W.eid ≤ 99999 ∧ E.salary ≤ 6000 ∧ E.salary < 1000 ∧ E.salary < 1000)

UPDATED- $L^{1\text{-}2}(q_4)$ = (E.salary > 5500 ∧ E.salary ≤ 12000 ∧ E.age ≥ 20 ∧ E.age < 65 ∧ E.eid = W.eid ∧ E.eid ≥ 10000 ∧ E.eid ≤ 99999 ∧ W.eid ≥ 10000 ∧ W.eid ≤ 99999 ∧ E.salary ≤ 6000 ∧ E.salary < 1000 ∧ E.salary ≥ 5700)

UPDATED- $L^{1\text{-}3}(q_4)$ = (E.salary > 5500 ∧ E.salary ≤ 12000 ∧ E.age ≥ 20 ∧ E.age < 65 ∧ E.eid = W.eid ∧ E.eid ≥ 10000 ∧ E.eid ≤ 99999 ∧ W.eid ≥ 10000 ∧ W.eid ≤ 99999 ∧ E.salary ≤ 6000 ∧ E.salary ≥ 5000 ∧ E.salary < 1000)

UPDATED- $L^{1\text{-}4}(q_4)$ = (E.salary > 5500 ∧ E.salary ≤ 12000 ∧ E.age ≥ 20 ∧ E.age < 65 ∧ E.eid = W.eid ∧ E.eid ≥ 10000 ∧ E.eid ≤ 99999 ∧ W.eid ≥ 10000 ∧ W.eid ≤ 99999 ∧ E.salary ≤ 6000 ∧ E.salary ≥ 5000 ∧ E.salary ≥ 5700)

UPDATED- $L^{1\text{-}5}(q_4)$ = (E.salary > 5500 ∧ E.salary ≤ 12000 ∧ E.age ≥ 20 ∧ E.age < 65 ∧ E.eid = W.eid ∧ E.eid ≥ 10000 ∧ E.eid ≤ 99999 ∧ W.eid ≥ 10000 ∧ W.eid ≤ 99999 ∧ E.salary ≤ 6000 ∧ E.age ≤ 60 ∧ E.salary < 1000)

UPDATED- $L^{1\text{-}6}(q_4)$ = (E.salary > 5500 ∧ E.salary ≤ 12000 ∧ E.age ≥ 20 ∧ E.age < 65 ∧ E.eid = W.eid ∧ E.eid ≥ 10000 ∧ E.eid ≤ 99999 ∧ W.eid ≥ 10000 ∧ W.eid ≤ 99999 ∧ E.salary ≤ 6000 ∧ E.age ≤ 60 ∧ E.salary ≥ 5700)

UPDATED- $L^{1\text{-}7}(q_4)$ = (E.salary > 5500 ∧ E.salary ≤ 12000 ∧ E.age ≥ 20 ∧ E.age < 65 ∧ E.eid = W.eid ∧ E.eid ≥ 10000 ∧ E.eid ≤ 99999 ∧ W.eid ≥ 10000 ∧ W.eid ≤ 99999 ∧ E.salary ≤ 6000 ∧ E.age > 80 ∧ E.salary < 1000)

UPDATED- $L^{1\text{-}8}(q_4)$ = (E.salary > 5500 ∧ E.salary ≤ 12000 ∧ E.age ≥ 20 ∧ E.age < 65 ∧ E.eid = W.eid ∧ E.eid ≥ 10000 ∧ E.eid ≤ 99999 ∧ W.eid ≥ 10000 ∧ W.eid ≤ 99999 ∧ E.salary ≤ 6000 ∧ E.age > 80 ∧ E.salary ≥ 5700)

UPDATED- $L^{1\text{-}9}(q_4)$ = (E.salary > 5500 ∧ E.salary ≤ 12000 ∧ E.age ≥ 20 ∧ E.age < 65 ∧ E.eid = W.eid ∧ E.eid ≥ 10000 ∧ E.eid ≤ 99999 ∧ W.eid ≥ 10000 ∧ W.eid ≤ 99999 ∧ E.salary > 12000 ∧ E.salary < 1000 ∧ E.salary < 1000)

UPDATED- $L^{1\text{-}10}(q_4)$ = (E.salary > 5500 ∧ E.salary ≤ 12000 ∧ E.age ≥ 20 ∧ E.age < 65 ∧ E.eid = W.eid ∧ E.eid ≥ 10000 ∧ E.eid ≤ 99999 ∧ W.eid ≥ 10000 ∧ W.eid ≤ 99999 ∧ E.salary > 12000 ∧ E.salary < 1000 ∧ E.salary ≥ 5700)

UPDATED- $L^{1\text{-}11}(q_4)$ = (E.salary > 5500 ∧ E.salary ≤ 12000 ∧ E.age ≥ 20 ∧ E.age < 65 ∧ E.eid = W.eid ∧ E.eid ≥ 10000 ∧ E.eid ≤ 99999 ∧ W.eid ≥ 10000 ∧ W.eid ≤ 99999 ∧ E.salary > 12000 ∧ E.salary ≥ 5000 ∧ E.salary < 1000)

UPDATED- $L^{1\text{-}12}(q_4)$ = (E.salary > 5500 ∧ E.salary ≤ 12000 ∧ E.age ≥ 20 ∧ E.age < 65 ∧ E.eid = W.eid ∧ E.eid ≥ 10000 ∧ E.eid ≤ 99999 ∧ W.eid ≥ 10000 ∧ W.eid ≤ 99999 ∧ E.salary > 12000 ∧ E.salary ≥ 5000 ∧ E.salary ≥ 5700)

UPDATED- $L^{1\text{-}13}(q_4)$ = (E.salary > 5500 ∧ E.salary ≤ 12000 ∧ E.age ≥ 20 ∧ E.age < 65 ∧ E.eid = W.eid ∧ E.eid ≥ 10000 ∧ E.eid ≤ 99999 ∧ W.eid ≥ 10000 ∧ W.eid ≤ 99999 ∧ E.salary > 12000 ∧ E.age ≤ 60 ∧ E.salary < 1000)

UPDATED- $L^{1\text{-}14}(q_4)$ = (E.salary > 5500 ∧ E.salary ≤ 12000 ∧ E.age ≥ 20 ∧ E.age < 65 ∧ E.eid = W.eid ∧ E.eid ≥ 10000 ∧ E.eid ≤ 99999 ∧ W.eid ≥ 10000 ∧ W.eid ≤ 99999 ∧ E.salary > 12000 ∧ E.age ≤ 60 ∧ E.salary ≥ 5700)

UPDATED- $L^{1\text{-}15}(q_4)$ = (E.salary > 5500 ∧ E.salary ≤ 12000 ∧ E.age ≥ 20 ∧ E.age < 65 ∧ E.eid = W.eid ∧ E.eid ≥ 10000 ∧ E.eid ≤ 99999 ∧ W.eid ≥ 10000 ∧ W.eid ≤ 99999 ∧ E.salary > 12000 ∧ E.age > 80 ∧ E.salary < 1000)

UPDATED- $L^{1\text{-}16}(q_4)$ = (E.salary > 5500 ∧ E.salary ≤ 12000 ∧ E.age ≥ 20 ∧ E.age < 65 ∧ E.eid = W.eid ∧ E.eid ≥ 10000 ∧ E.eid ≤ 99999 ∧ W.eid ≥ 10000 ∧ W.eid ≤ 99999 ∧ E.salary > 12000 ∧ E.age > 80 ∧ E.salary ≥ 5700)

Simplify procedure eliminates the implicants containing predicates contradicting with another, and predicates that violate domain constraints. Remaining predicates will be further simplified by eliminating subsumed predicates. The implicants UPDATED- $L^{1\text{-}4}(q_4)$ and UPDATED- $L^{1\text{-}6}(q_4)$ are the only ones that do not contain predicates contradicting with another or predicates violating domain constraints. However, in UPDATED- $L^{1\text{-}4}(q_4)$ E.salary > 5500 and E.salary ≥ 5000 are subsumed by E.salary ≥

5700 and E.salary ≤ 12000 is subsumed by E.salary ≤ 6000, and in UPDATED- $L^{1\text{-}6}(q_4)$ E.salary > 5500 is subsumed by E.salary ≥ 5700, E.salary ≤ 12000 is subsumed by E.salary ≤ 6000 and E.age < 65 is subsumed by E.age ≤ 60. After these subsumed predicates are eliminated, and $L(q_4)$ will consist of the following implicants:

$L^1(q_4)$ = (E.age ≥ 20 ∧ E.age < 65 ∧ E.eid = W.eid ∧ E.eid ≥ 10000 ∧ E.eid ≤ 99999 ∧ W.eid ≥ 10000 ∧ W.eid ≤ 99999 ∧ E.salary ≤ 6000 ∧ E.salary ≥ 5700)

$L^2(q_4)$ = (E.age ≥ 20 ∧ E.eid = W.eid ∧ E.eid ≥ 10000 ∧ E.eid ≤ 99999 ∧ W.eid ≥ 10000 ∧ W.eid ≤ 99999 ∧ E.salary ≤ 6000 ∧ E.age ≤ 60 ∧ E.salary ≥ 5700)

In Step 4, we further modify the logical formulas of the queries by considering table constraints. The only table constraint enforced in S is B = (age <= 70 OR salary > 3500). HandleTableConstraints procedure converts this constraint into DNF, and obtains two implicants:

$L^1(B)$ = (age ≤ 70)

$L^2(B)$ = (salary > 3500)

For each implicant $L^k(q)$ of the logical formula of each query q whose p is ∃, HandleTableConstraints conjuncts $L^1(B)$ and/or $L^2(B)$ to $L^k(q)$ if age and/or salary is used in $L^k(q)$. $L^1(q_1)$, $L^1(q_4)$ and $L^2(q_4)$ use the attribute salary, and $L^1(q_2)$, $L^1(q_4)$ and $L^2(q_4)$ use the attribute age. Therefore, after HandleTableConstraints procedure, $L^1(q_1)$, $L^1(q_2)$, $L^1(q_4)$, $L^2(q_4)$ will be modified as follows:

$L^1(q_1)$ = (E.salary > 6000 ∧ E.salary ≤ 12000 ∧ E.salary > 3500)

$L^1(q_2)$ = (E.age > 65 ∧ E.age ≤ 80 ∧ E.age ≤ 70)

$L^1(q_4)$ = (E.age ≥ 20 ∧ E.age < 65 ∧ E.eid = W.eid ∧ E.eid ≥ 10000 ∧ E.eid ≤ 99999 ∧ W.eid ≥ 10000 ∧ W.eid ≤ 99999 ∧ E.salary ≤ 6000 ∧ E.salary ≥ 5700) ∧

(E.age ≤ 70 ∨ E.salary > 3500)

$L^2(q_4)$ = (E.age ≥ 20 ∧ E.eid = W.eid ∧ E.eid ≥ 10000 ∧ E.eid ≤ 99999 ∧ W.eid ≥ 10000 ∧ W.eid ≤ 99999 ∧ E.salary ≤ 6000 ∧ E.age ≤ 60 ∧ E.salary ≥ 5700) ∧ (E.age ≤ 70 ∨ E.salary > 3500)

After simplification applied in HandleTableConstraints procedure using the procedure Simplify, we obtain the following implicants. Note that only $L^1(q_2)$ is modified, and the predicates of the CHECK constraint is subsumed in the other implicants.

$L^1(q_1)$ = (E.salary > 6000 ∧ E.salary ≤ 12000)

$L^1(q_2)$ = (E.age > 65 ∧ E.age ≤ 70)

$L^1(q_4)$ = (E.age ≥ 20 ∧ E.age < 65 ∧ E.eid = W.eid ∧ E.eid ≥ 10000 ∧ E.eid ≤ 99999 ∧ W.eid ≥ 10000 ∧ W.eid ≤ 99999 ∧ E.salary ≤ 6000 ∧ E.salary ≥ 5700)

$L^2(q_4)$ = (E.age ≥ 20 ∧ E.eid = W.eid ∧ E.eid ≥ 10000 ∧ E.eid ≤ 99999 ∧ W.eid ≥ 10000 ∧ W.eid ≤ 99999 ∧ E.salary ≤ 6000 ∧ E.age ≤ 60 ∧ E.salary ≥ 5700)

In Step 5, we modify the formulas obtained from queries having ∃ property such that domain constraints of the attributes defining primary keys of each relation in the formula, and that of each relation referenced by the relations in the formula will be incorporated. Since $L^1(q_3)$, $L^1(q_4)$ and $L^2(q_4)$ references *Dept* in their foreign key constraints, the join predicate W.did = D.did and the domain constraints of W.did and D.did will be conjuncted to these implicants. The implicants will be modified as follows (after applying the rule of commutativity):

$L^1(q_1)$ = (E.salary > 6000 ∧ E.salary ≤ 12000 ∧ E.eid ≥ 10000 ∧ E.eid ≤ 99999)

$L^1(q_2)$ = (E.age > 65 ∧ E.age ≤ 70 ∧ E.eid ≥ 10000 ∧ E.eid ≤ 99999)

$L^1(q_3)$ = (W.months ≥ 0 ∧ W.months ≤ 480 ∧ E.eid ≥ 10000 ∧ E.eid ≤ 99999 ∧ W.eid ≥ 10000 ∧ W.eid ≤ 99999 ∧ W.did ≥ 100 ∧ W.did ≤ 999 ∧ D.did ≥ 100 ∧ D.did ≤ 999 ∧ W.eid = E.eid ∧ W.did = D.did)

$L^1(q_4)$ = (E.age ≥ 20 ∧ E.age < 65 ∧ E.salary ≥ 5700 ∧ E.salary ≤ 6000 ∧ E.eid = W.eid ∧ E.eid ≥ 10000 ∧ E.eid ≤ 99999 ∧ W.eid ≥ 10000 ∧ W.eid ≤ 99999 ∧ W.did ≥ 100 ∧ W.did ≤ 999 ∧ D.did ≥ 100 ∧ D.did ≤ 999 ∧ W.eid = E.eid ∧ W.did = D.did)

$L^2(q_4)$ = (E.age ≥ 20 ∧ E.age ≤ 60 ∧ E.salary ≥ 5700 ∧ E.salary ≤ 6000 ∧ E.eid = W.eid ∧ E.eid ≥ 10000 ∧ E.eid ≤ 99999 ∧ W.eid ≥ 10000 ∧ W.eid ≤ 99999 ∧W.did ≥ 100 ∧ W.did ≤ 999 ∧ D.did ≥ 100 ∧ D.did ≤ 999 ∧ W.eid = E.eid ∧ W.did = D.did)

In Step 6, we call AssignSubscripts procedure, which assigns subscripts to the implicants of all queries with ∃ property. First, AssignSubscripts procedure gives $L^1(q_1)$, $L^1(q_2)$, $L^1(q_3)$, $L^1(q_4)$ and $L^2(q_4)$ to CST as an input. CST does not report contradiction for any of these implicants, therefore AssignSubscripts procedure assigns subscripts to all the attributes used in these implicants. The logical formulas after this step are shown below:

$L^1(q_1)$ = (E_0.salary > 6000 ∧ E_0.salary ≤ 12000 ∧ E_0.eid ≥ 10000 ∧ E_0.eid ≤ 99999)

$L^1(q_2)$ = (E_1.age > 65 ∧ E_1.age ≤ 70 ∧ E_1.eid ≥ 10000 ∧ E_1.eid ≤ 99999)

$L^1(q_3)$ = (W_0.months ≥ 0 ∧ W_0.months ≤ 480 ∧ E_2.eid ≥ 10000 ∧ E_2.eid ≤ 99999 ∧ W_0.eid ≥ 10000 ∧ W_0.eid ≤ 99999 ∧ W_0.did ≥ 100 ∧ W_0.did ≤ 999 ∧ D_0.did ≥ 100 ∧ D_0.did ≤ 999 ∧ W_0.eid = E_2.eid ∧ W_0.did = D_0.did)

$L^1(q_4)$ = (E_3.age ≥ 20 ∧ E_3.age < 65 ∧ E_3.salary ≥ 5700 ∧ E_3.salary ≤ 6000 ∧ E_3.eid = W_1.eid ∧ E_3.eid ≥ 10000 ∧ E_3.eid ≤ 99999 ∧ W_1.eid ≥ 10000 ∧ W_1.eid ≤ 99999

$\wedge$ W_1.did $\geq$ 100 $\wedge$ W_1.did $\leq$ 999 $\wedge$ D_1.did $\geq$ 100 $\wedge$ D_1.did $\leq$ 999 $\wedge$ W_1.eid = E_3.eid $\wedge$ W_1.did = D_1.did)

$L^2(q_4)$ = (E_4.age $\geq$ 20 $\wedge$ E_4.age $\leq$ 60 $\wedge$ E_4.salary $\geq$ 5700 $\wedge$ E_4.salary $\leq$ 6000 $\wedge$ E_4.eid = W_2.eid $\wedge$ E_4.eid $\geq$ 10000 $\wedge$ E_4.eid $\leq$ 99999 $\wedge$ W_2.eid $\geq$ 10000 $\wedge$ W_2.eid $\leq$ 99999 $\wedge$ W_2.did $\geq$ 100 $\wedge$ W_2.did $\leq$ 999 $\wedge$ D_2.did $\geq$ 100 $\wedge$ D_2.did $\leq$ 999 $\wedge$ W_2.eid = E_4.eid $\wedge$ W_2.did = D_2.did)

In Step 7 and 8, we obtain *L(PK)* and *L(FK)* which will be incorporated to the final formulas to be given to *CST* for ensuring primary key and foreign key constraints. *L(PK)* and *L(FK)* are given below, which are also shown in Figure 4.7 in HySat syntax.

L(PK) = (E_0.eid != E_1.eid $\wedge$ E_0.eid != E_2.eid $\wedge$ E_0.eid != E_3.eid $\wedge$ E_0.eid != E_4.eid) $\wedge$

(E_1.eid != E_2.eid $\wedge$ E_1.eid != E_3.eid $\wedge$ E_1.eid != E_4.eid) $\wedge$

(E_2.eid != E_3.eid $\wedge$ E_2.eid != E_4.eid) $\wedge$ (E_3.eid != E_4.eid) $\wedge$

(W_0.eid != W_1.eid $\wedge$ W_0.did != W_1.did) $\wedge$

(W_0.eid != W_2.eid $\wedge$ W_0.did != W_2.did) $\wedge$

(W_1.eid != W_2.eid $\wedge$ W_1.did != W_2.did) $\wedge$

(D_0.did != D_1.did $\wedge$ D_0.did != D_2.did) $\wedge$ (D_1.did != D_2.did)

L(FK) = (W_0.eid = E_0.eid $\vee$ W_0.eid = E_1.eid $\vee$ W_0.eid = E_2.eid $\vee$ W_0.eid = E_3.eid $\vee$ W_0.eid = E_4.eid) $\wedge$

(W_1.eid = E_0.eid $\vee$ W_1.eid = E_1.eid $\vee$ W_1.eid = E_2.eid $\vee$ W_1.eid = E_3.eid $\vee$ W_1.eid = E_4.eid) $\wedge$

(W_2.eid = E_0.eid $\vee$ W_2.eid = E_1.eid $\vee$ W_2.eid = E_2.eid $\vee$ W_2.eid = E_3.eid $\vee$ W_2.eid = E_4.eid) $\wedge$

(W_0.did = D_0.did $\vee$ W_0.did = D_1.did $\vee$ W_0.did = D_2.did) $\wedge$

(W_1.did = D_0.did $\vee$ W_1.did = D_1.did $\vee$ W_1.did = D_2.did) $\wedge$

(W_2.did = D_0.did $\vee$ W_2.did = D_1.did $\vee$ W_2.did = D_2.did)

We show the logical formula $L(Q)$ obtained by Step 9 in HySat syntax in Figures 4.6 and 4.7. Figure 4.6 shows the DECL section, and Figure 4.7 shows the EXPR section of the HySat file. We merge and input these two files to HySat. Note that we eliminate the domain constraints from the logical formulas in the EXPR section, and declare unknowns with their corresponding domain constraints in the DECL section, because HySat requires a range of values that each unknown can take in the DECL section.

```
DECL
int [1000,12000] Emp_0_salary;
int [10000,99999] Emp_0_eid;
int [20,80] Emp_1_age;
int [10000,99999] Emp_1_eid;
int [0,480] Works_0_months;
int [10000, 99999] Emp_2_eid;
int [100, 999] Dept_0_did;
int [10000, 99999] Works_0_eid;
int [100,999] Works_0_did;
int [1000,12000] Emp_3_salary;
int [20,80] Emp_3_age;
int [10000,99999] Emp_3_eid;
int [10000,99999] Works_1_eid;
int [100, 999] Dept_1_did;
int [100,999] Works_1_did;
int [1000, 12000] Emp_4_salary;
int [20,80] Emp_4_age;
int [10000, 99999] Emp_4_eid;
int [10000, 99999] Works_2_eid;
int [100,999] Dept_2_did;
int [100,999] Works_2_did;
```

Figure 4.6. DECL section of the final constraint $L(Q)$.

```
EXPR
(Emp_0_salary > 6000 and Emp_0_salary <=12000);

(Emp_1_age > 65 and Emp_1_age <= 70);

(Works_0_eid = Emp_2_eid and Works_0_did = Dept_0_did);

(Emp_3_age >= 20 and Emp_3_age < 65 and Emp_3_eid = Works_1_eid and
 Emp_3_eid = Works_1_eid and Emp_3_salary >= 5700 and
 Emp_3_salary <= 6000 and Works_1_did = Dept_1_did);

(Emp_4_age >= 20 and Emp_4_age <= 60 and Emp_4_eid = Works_2_eid and
 Emp_4_salary >= 5700 and Emp_4_salary <= 6000 and Works_2_did =
 Dept_2_did);

--PK constraint for Emp table
(Emp_0_eid != Emp_1_eid and Emp_0_eid != Emp_2_eid);
(Emp_0_eid != Emp_3_eid and Emp_0_eid != Emp_4_eid);
(Emp_1_eid != Emp_2_eid and Emp_1_eid != Emp_3_eid);
(Emp_1_eid != Emp_4_eid and Emp_2_eid != Emp_3_eid);
(Emp_2_eid != Emp_4_eid and Emp_3_eid != Emp_4_eid);

--PK Constraint for Works table
(Works_0_eid != Works_1_eid or Works_0_did != Works_1_did);
(Works_0_eid != Works_2_eid or Works_0_did != Works_2_did);
(Works_1_eid != Works_2_eid or Works_1_did != Works_2_did);
--PK for Dept table
(Dept_0_did != Dept_1_did and Dept_0_did != Dept_2_did);
(Dept_1_did != Dept_2_did);
--FK Works
(Works_0_eid = Emp_0_eid or Works_0_eid = Emp_1_eid or
 Works_0_eid = Emp_2_eid or Works_0_eid = Emp_3_eid or
 Works_0_eid = Emp_4_eid);
(Works_1_eid = Emp_0_eid or Works_1_eid = Emp_1_eid or
 Works_1_eid = Emp_2_eid or Works_1_eid = Emp_3_eid or
 Works_1_eid = Emp_4_eid);
(Works_2_eid = Emp_0_eid or Works_2_eid = Emp_1_eid or
 Works_2_eid = Emp_2_eid or Works_2_eid = Emp_3_eid or
 Works_2_eid = Emp_4_eid);
(Works_0_did = Dept_0_did or Works_0_did = Dept_1_did or
 Works_0_did = Dept_2_did);
(Works_1_did = Dept_0_did or Works_1_did = Dept_1_did or
 Works_1_did = Dept_2_did);
(Works_2_did = Dept_0_did or Works_2_did = Dept_1_did or
 Works_2_did = Dept_2_did);
```

Figure 4.7. EXPR section for the final constraint $L(Q)$.

The output of *CST* is the solution for each unknown in DECL section, where substituting these values to unknowns causes the constraint in EXPR section to be satisfied. Thus, we have generated the initial database D_0, which is shown below in

Tables 4.2(a), 4.2(b), 4.2(c):

Table 4.2(a). Contents of *Emp* table.

Eid	*Name*	*Age*	*Salary*
47579			10327
66978		67	
83173			
46172		25	5972
23672		46	5884

Table 4.2(b). Contents of *Works* table.

Eid	*Did*	*Months*
83173	968	278
46172	149	
23672	655	

Table 4.2(c). Contents of *Dept* table.

Did	*Dname*	*Budget*
968		
149		
655		

To check if this method successfully resolves the conflicts between queries, we run each query in Q in the given order and check if their properties are violated. First, we run q_1 and Table 4.3 illustrates *Emp* table after running q_1. In this example, employee with *eid* 47579 will be deleted from the database because its salary is above 6000 and since employee 47579 is not in *Works* table, *Works* will stay unmodified. The reason why we added ON UPDATE CASCADE ON DELETE CASCADE statements to foreign keys in *Works* table is that without these statements, any attempt to modify or delete a tuple T of *Emp* or *Dept* table will be rejected if a tuple in *Works* references T. With these statements,

any modification or deletion of T will modify or delete the corresponding tuple in *Works*.

Table 4.3. Contents of *Emp* table after running q_1.

Eid	*Name*	*Age*	*Salary*
66978		67	
83173			
46172		25	5972
23672		46	5884

The next query in Q to be run is q_2, which deletes the employees that are older than 65 years old. The tuple in *Emp* table with *eid* 66978 will be deleted. The contents of *Emp* table after running q_2 is shown in Table 4.4. Till now, properties of q_1 and q_2 are satisfied, because there exists a tuple for each query in the database which satisfy their WHERE clauses.

Table 4.4. Contents of *Emp* table after running q_2.

Eid	*Name*	*Age*	*Salary*
83173			
46172		25	5972
23672		46	5884

Next query in Q to be run is q_3, which increments the *months* attribute of *Works* table by 1 for each employee working in some department. Property of q_3 will be satisfied, since q_3 will update the first tuple. Table 6 illustrates the content of *Works* table after running q_3.

Table 4.5. Contents of *Works* table after running q_3

Eid	*Did*	*Months*
83173	968	279
46172	149	
23672	655	

q_4 selects the employees that are earning more than 5500 and younger than 65. Employees 46172 and 23672 will be returned as a result, and p_4 will be satisfied.

q_5 selects all the employees that worked less than 2 years, but older than 60. Property of q_5 is $\nexists$ and no tuple should satisfy the WHERE clause of this query. After we execute q_5 on the database, we see that no rows are returned, therefore we conclude that $\nexists$ property of q_5 is satisfied.

The last query q_6 retrieves all the employees earning less than 5700. As in q_5, q_6 also have $\nexists$ property. We see that no tuple satisfies the WHERE clause of q_6, therefore, p_6 is satisfied.

As seen in this example, we can use constraint solving techniques to generate a database instance as a test data. In this chapter, we proposed a method to generate a database instance for a test case consisting of simple queries. By using this method, we will propose a method to generate database instances for a test suite and we will consider more complex queries that use ANY, ALL, [NOT] IN, [NOT] EXISTS keywords.

Chapter Five

Extending Simple Queries

5.1 Formalizing complex SQL queries

Although SQL is both an ANSI and ISO standard, current open source and commercial DBMSs supports SQL with extensions. Therefore, each DBA is likely to be written for a specific DBMS, and there might be slight differences between the syntax of the queries in each DBA. In addition, even the standard SQL is a broad query language, which allows the users and developers to write complex queries to retrieve the data. Considering all the functionality provided by SQL is beyond the scope of this thesis. However, we will extend the queries we consider in the previous chapter by using nested queries using [NOT] IN, [NOT] EXISTS, ALL and ANY keywords. We still exclude in particular the following from SQL-92 in this thesis:

- GROUP BY and HAVING clauses
- CHARACTER, CHARACTER VARYING, CHARACTER LARGE OBJECT, BINARY LARGE OBJECT, DATE, TIME, TIMESTAMP and INTERVAL data types
- All ANSI functions except
    - ALL, ANY, [NOT] EXISTS, [NOT] IN clauses
    - Aggregate functions SUM, COUNT, MIN, MAX, AVG if they are used in <attribute_list> a query

Figure 5.1 and 5.2 give the grammar that we use to represent the queries.

```
<sel_query> := SELECT <attribute_list> FROM <table_list>
               WHERE <where_clause>

<attribute_list>:=<attribute>|<attribute>, <attribute_list>

<attribute> := <attribute_name> | <table_alias>.<attribute_name>

<table_list> := <table_name> <table_alias> |
                <table_name> <table_alias>, <table_list>

<where_clause> :=  <predicate> | <predicate> <lop> <where_clause>  |
                   <attribute> <rop> <any_all>(<sel_simple>)|
                   <attribute> <in_nin> (<value_list>) |
                   <attribute> <in_nin> (<sel_simple>) |
                   <exists_nexists> <sel_simple> | ε

<predicate> := <attribute><rop><value> | <attribute> <rop> <attribute>

<aggregate> := <aggr_func> ( <attribute> )

<aggr_func> := MIN | MAX | SUM | AVG | COUNT

<lop> := AND | OR

<any_all> := ANY | ALL

<rop> := < | > | <= | >= | = | !=

<in_nin> := IN | NOT IN

<exists_nexists> := EXISTS | NOT EXISTS

<value> :=  <alphanumeric> | <numeric>

<table_alias> :=  <alphanumeric>

<attribute_name> := <alphanumeric>

<table_name> := <alphanumeric>

<value_list> := <value> | <value> , <value_list>

<alphanumeric> := {∑+ | ∑ = {A, B,…, Z, a, b,…, z, 0, 1,…, 9}}

<numeric> := {∑+ | ∑ = {0, 1,…, 9}}
```

Figure 5.1. Grammar of a SELECT query

```
<del_query> := DELETE FROM <table_name>
               WHERE <where_clause>

<upd_query> := UPDATE <table_name>
               SET <upd_cond_list>
               WHERE <where_clause>

<ins_query> := INSERT INTO <table_name> ( <simple_sel>) |
               <ins_simple>

<ins_simple> := INSERT INTO <table_name>
                VALUES (<value_list>)

<upd_cond_list> := <upd_condition> | <upd_condition> , <upd_cond_list>

<upd_condition> := <attribute> = <value>
```

Figure 5.2. Grammar for DELETE, UPDATE and INSERT queries

Note that Figure 5.1 and 5.2 uses <sel_simple> from Figure 4.1 and Figure 5.2 uses elements from Figure 5.1.

5.2 Generating Logical Formulas for SQL queries

In this section, we categorize the queries following the grammar given in Figure 5.1 and 5.2. For each category given, we will discuss the semantics of the query and how to obtain the logical formula which will contribute to the formation of the constraint to be given to *CST* as an input.

5.2.1 Simple Queries using BETWEEN clauses

Simple queries contain only conjunctions and disjunctions of predicates in their WHERE clauses. More specifically, WHERE clauses of these queries obey the grammar shown below, and do not contain clauses ANY, ALL, [NOT] EXISTS and [NOT] IN.

```
<where_clause> := <predicate> |
                  <predicate> <lop> <where_clause>
```

Chapter 4 dealt with simple queries, therefore, we will only discuss about a small addition that we will make on simple queries. BETWEEN clause can be used to check if the value of an attribute is in a given interval. An example of a simple SELECT query q using BETWEEN clause is shown below.

Example 5.1. Simple SELECT query using BETWEEN clause:

```
SELECT E.eid
FROM Emp E
WHERE E.salary BETWEEN 5500 AND 6000;
```

Intuitively, logical formula $L(q)$ for the query above is $L(q)$ = E.salary >= 5000 $\wedge$ E.salary <= 6000 $\wedge$ (E.eid $\geq$ 10000 $\wedge$ E.eid $\leq$ 99999) $\wedge$ (E.salary $\geq$ 1000 $\wedge$ E.salary $\leq$ 12000).

5.2.2 Simple Queries Using IN/NOT IN Clauses

This category represents the queries containing conjunction and/or disjunction of predicates and an IN/NOT IN clause of the form `<attribute> <in> (<value_list>)`. Note that `(<value_list>)` represents a set of constants, not a subquery. A query q in this category will have the following syntax:

```
SELECT <attribute_list>
FROM <table_list>
WHERE [L'(q) <lop>] a (IN | NOT IN) (c1, c2,…, cm);
or
DELETE FROM <table_name>
WHERE [L'(q) <lop>] a (IN | NOT IN) (c1, c2,…, cm);
or
UPDATE <table_name>
SET <upd_cond_list>
```

```
WHERE [L'(q) <lop>] a (IN | NOT IN) (c1, c2,..., cm);
```

where $L'(q)$ is a logical formula, *lop* is one of the logical operators <lop> shown in Figure 5.1, a is an attribute satisfying <attribute> in Figure 5.1 and $c_1, c_2, \ldots, c_m$ is a multi-set of constants of size m satisfying <value_list> in Figure 5.1.

We use the following algorithm to generate the logical formula for simple queries containing IN/NOT IN clauses:

Algorithm 5.1 GenerateFormula-Simple-N-IN

```
procedure GenerateFormula-Simple-N-IN(q, L(q))
  if(category(q) is NOT IN)then
     L(q) ← [L'(q) <lop>] ¬(a = c1 ∨  a = c2 ∨ ... ∨ a = cm)
     NegationFree(L(q))
  endif
  if(category(q) is IN)then
     L(q) ← [L'(q) <lop>] (a = c1 ∨  a = c2 ∨ ... ∨ a = cm);
  endif
  if(type(q) = SELECT and p = ∃) then
     for each attribute x used in <attribute_list> of q do
        L(q) ← L(q) ∧ getDomainConstraint(x)
     endfor
  endif
  if(type(q) = UPDATE) then
     for each attribute x used in SET clause of q do
        L(q) ← L(q) ∧ getDomainConstraint(x)
     endfor
  endif
```

```
    NegationFree(L(q))
    convert L(q) into DNF by algebraic manipulation
    for each implicant L^k(q) in L(q) do
        for each attribute x used in L^k(q) do
            L^k(q) ← L^k(q) ∧ getDomainConstraint(x)
        endfor
    endfor
endprocedure
```

An example of a simple query q with IN clause is shown below.

Example 5.2. Simple query with IN clause:

```
SELECT E.eid, E.salary
FROM Emp E
WHERE E.salary < 3000 AND
      E.age IN (30, 40, 50, 60, 70);
```

This query retrieves the id's and salaries of the employees whose salaries are less than 3000 and their age is one of the values provided by the list (30, 40, 50, 60, 70). As one can notice, <table_list> = {Emp E}, $L'(q)$ = E.salary < 3000 , a = E.age and <value_list> = (30, 40, 50, 60, 70). Then,

$L^1(q)$ = E.salary < 3000 ∧ E.age = 30 ∧ (E.eid ≥ 10000 ∧ E.eid ≤ 99999) ∧ (E.salary ≥ 1000 ∧ E.salary ≤ 12000) ∧ (E.age ≥ 20 ∧ E.age ≤ 80)

$L^2(q)$ = E.salary < 3000 ∧ E.age = 40 ∧ (E.eid ≥ 10000 ∧ E.eid ≤ 99999) ∧ (E.salary ≥ 1000 ∧ E.salary ≤ 12000) ∧ (E.age ≥ 20 ∧ E.age ≤ 80)

$L^3(q)$ = E.salary < 3000 ∧ E.age = 50 ∧ (E.eid ≥ 10000 ∧ E.eid ≤ 99999) ∧ (E.salary ≥ 1000 ∧ E.salary ≤ 12000) ∧ (E.age ≥ 20 ∧ E.age ≤ 80)

$L^4(q)$ = E.salary < 3000 ∧ E.age = 60 ∧ (E.eid ≥ 10000 ∧ E.eid ≤ 99999) ∧ (E.salary ≥ 1000 ∧ E.salary ≤ 12000) ∧ (E.age ≥ 20 ∧ E.age ≤ 80)

$L^5(q)$ = E.salary < 3000 ∧ E.age = 70 ∧ (E.eid ≥ 10000 ∧ E.eid ≤ 99999) ∧ (E.salary ≥ 1000 ∧ E.salary ≤ 12000) ∧ (E.age ≥ 20 ∧ E.age ≤ 80)

Example 5.3. The above query with NOT IN clause will provide an example for the use of NOT IN clause:

```
SELECT E.eid, E.salary
FROM Emp E
WHERE E.salary < 3000 AND
      E.age NOT IN (30, 40, 50, 60, 70);
```

This new query should retrieve the employees who earn less than 3000 and having an age different from 30, 40, 50, 60 or 70. Then,

$L^1(q)$ = E.salary<3000 ∧ (E.age≠30 ∧ E.age≠40 ∧ E.age≠50 ∧ E.age≠60 ∧ E.age≠70) ∧ (E.eid ≥ 10000 ∧ E.eid ≤ 99999) ∧ (E.salary ≥ 1000 ∧ E.salary ≤ 12000) ∧ (E.age ≥ 20 ∧ E.age ≤ 80)

For ease of presentation, until Section 5.3, we will not take conflict among queries, integrity constraints (other than domain constraints) and DNF conversion into account. Recall from Chapter 4 that generating logical formulas for SELECT, UPDATE and DELETE queries are very similar, and the difference is including the domain constraints of attributes listed in <attribute_list> if the query is SELECT, or the attributes listed in SET clause if the query is an UPDATE. Therefore, for logical formula generation examples, we will only use SELECT queries.

5.2.3 Nested Queries Using IN/NOT IN Clauses

The only difference between this category and the category in Section 5.2.2 is the part following the IN clause. In 5.2.2, the query contains a set of constants. However, in this category, a subquery is replacing this set and the result table of the inner query should contain one attribute having values of same type with the attribute before IN clause.

Assumption 5.1. We assume that developer of the DBA writes meaningful (i.e., semantically correct) queries such that there is only one attribute in the <attribute_list> of IN, NOT IN, ANY and ALL queries, and the type of this attribute matches with the attribute before one of these clauses. Note that this assumption has to be stated since the grammar given in Figures 5.1 and 5.2 cannot capture this information.

A query q in this category will have the following syntax:

```
SELECT <attribute_list>
FROM <table_list>
WHERE [L'(q) <lop>] a (IN | NOT IN) (SELECT a'
                                    FROM <table_list>
                                    WHERE L'(q'));
```

The WHERE clause of this query is very similar to the one shown in Section 5.2.2, except a subquery is used instead of a value list. As one can observe, the subquery is a simple select query, and Assumption 5.2 below states the reason. We use the following algorithm for generating logical formulas of nested IN queries.

Algorithm 5.2 GenerateFormula-N-IN

```
procedure GenerateFormula-N-IN(q, L(q))
```

```
    if(category(q) is NOT IN)
       L(q) ← [(L'(q)) <lop>] (a≠a') ∧ L'(q')
       NegationFree(L(q))
    else
       L(q) ← [(L'(q)) <lop>] (a=a') ∧ L'(q')
    endif
    if(type(q) = SELECT and p = ∃) then
       for each attribute x used in <attribute_list> of q do
          L(q) ← L(q) ∧ getDomainConstraint(x)
       endfor
    endif
    if(type(q) = UPDATE) then
       for each attribute x used in SET clause of q do
           L(q) ← L(q) ∧ getDomainConstraint(x)
       endfor
    endif
    NegationFree(L(q))
    convert L(q) into DNF by algebraic manipulation
    for each implicant L^k(q) in L(q) do
       for each attribute x used in L^k(q) do
          L^k(q) ← L^k(q) ∧ getDomainConstraint(x)
       endfor
    endfor
endprocedure
```

Assumption 5.2. To simplify the discussion, we assumed that the inner queries are simple. However, relaxing this assumption will not require any changes in the proposed method. If the inner query is also nested, one can use a recursive approach to generate the formula. Suppose that the inner query is not a simple query. An example is given below using only the IN clause.

```
SELECT <attribute_list>
FROM <table_list>
WHERE [L'(q) <lop>] a IN (SELECT <attribute_list>
                          FROM <table_list>
                          WHERE [L'(q) <lop>] a IN (<simple_sel>));
```

For the sake of this example, we call the outermost query as q, the query following the IN clause of q as q', and (<simple_sel>) as q''. In the recursive approach, first we generate the logical formula considering q' and q'' with the proposed method for formula generation, which we call $L(q')$. Then, we replace the WHERE clause of the inner query with $L(q')$, thus reducing the problem to generating a logical formula for a nested query where the inner query is a simple query.

Example 5.4. Consider the query q shown below:

```
SELECT E.age
FROM Emp E
WHERE E.eid IN (SELECT W.eid
                FROM Works W, Dept D
                WHERE W.did = D.did AND
                      D.budget < 200000);
```

This query finds the ages and salaries of the employees who work in a department having a budget less than 200.000. By using GenerateFormula-N-IN, we obtain the following $L(q)$:

$L(q)$ = (E.eid = W.eid) $\wedge$ (W.did = D.did $\wedge$ D.budget < 200000) $\wedge$ (E.eid $\geq$ 10000 $\wedge$ E.eid $\leq$ 99999) $\wedge$ (E.age $\geq$ 20 $\wedge$ E.age $\leq$ 80) $\wedge$ (W.eid $\geq$ 10000 $\wedge$ W.eid $\leq$ 99999) $\wedge$ (W.did $\geq$ 100 $\wedge$ W.did $\leq$ 999) $\wedge$(D.did $\geq$ 100 $\wedge$ D.did $\leq$ 999)

Example 5.5. Consider the query given above, but with a NOT IN clause.

```
SELECT E.age, E.salary
FROM Emp E
WHERE E.eid NOT IN (SELECT W.eid
                    FROM Works W, Dept D
                    WHERE W.did = D.did AND
                          D.budget < 200000);
```

The logical formula $L(q)$ is shown below.

$L(q)$ = (E.eid $\neq$ W.eid) $\wedge$ (W.did = D.did $\wedge$ D.budget < 200000) $\wedge$ (E.eid $\geq$ 10000 $\wedge$ E.eid $\leq$ 99999) $\wedge$ (E.age $\geq$ 20 $\wedge$ E.age $\leq$ 80) $\wedge$ (W.eid $\geq$ 10000 $\wedge$ W.eid $\leq$ 99999) $\wedge$ (W.did $\geq$ 100 $\wedge$ W.did $\leq$ 999) $\wedge$(D.did $\geq$ 100 $\wedge$ D.did $\leq$ 999) $\wedge$ (D.budget $\geq$ 0 $\wedge$ D.budget $\leq$ 2000000)

5.2.4 Nested Queries Using EXISTS/ NOT EXISTS Clauses

EXISTS clause can be used in the WHERE clause of a query with a subquery following it, i.e., EXISTS (<sel_query>). An EXISTS clause of this form is said to be satisfied if there is at least one row returned from the subquery. A query q in this category has the following syntax:

```
SELECT <attribute_list>
```

```
FROM <table_list>
WHERE [L'(q) <lop>] (EXISTS | NOT EXISTS) (<simple_sel>);
```

We use the following algorithm to generate formulas for queries with NOT EXISTS/ EXISTS clauses.

Algorithm 5.3 GenerateFormula-N-EXISTS

```
procedure GenerateFormula-N-EXISTS(q, L(q))
   if (category(q) is NOT EXISTS)
      L(q) ← [L'(q) <lop>] ¬(L'(q'))
      NegationFree(L(q))
   else
      L(q) ← [L'(q) <lop>] L'(q')
   endif
   if(type(q) = SELECT and p = ∃) then
      for each attribute x used in <attribute_list> of q do
         L(q) ← L(q) ∧ getDomainConstraint(x)
      endfor
   endif
   if(type(q) = UPDATE) then
      for each attribute x used in SET clause of q do
          L(q) ← L(q) ∧ getDomainConstraint(x)
      endfor
   endif
   NegationFree(L(q))
   convert L(q) into DNF by algebraic manipulation
   for each implicant L^k(q) in L(q) do
```

```
      for each attribute x used in L^k(q) do
         L^k(q) ← L^k(q) ∧ getDomainConstraint(x)
      endfor
   endfor
endprocedure
```

Example 5.6 Consider the following query q, which retrieves the ids of employees having salary greater than 2000, and working in some department for more than 30 years (360 months).

```
SELECT E.eid
FROM Emp E
WHERE E.salary > 2000 AND
      EXISTS (SELECT *
              FROM Works W
              WHERE W.eid = E.eid AND
                    W.months > 360);
```

The logical formula $L(q)$ is the following:

$L(q)$ = E.salary>2000 $\wedge$ (W.eid=E.eid $\wedge$ W.months>360) $\wedge$ (E.eid $\geq$ 10000 $\wedge$ E.eid $\leq$ 99999) $\wedge$ (W.eid $\geq$ 10000 $\wedge$ W.eid $\leq$ 99999) $\wedge$ (W.months $\geq$ 0 $\wedge$ W.months $\leq$ 480)

Similar to IN clause, EXISTS clause can also be combined with a NOT clause. Also, a NOT EXISTS clause followed by a subquery forms a NOT EXISTS clause, i.e., NOT EXISTS (<simple_sel>) which is considered to be satisfied if the subquery returns no rows. Consider the query given above, but with a NOT clause combined with EXISTS clause.

```
SELECT E.eid
FROM Emp E
```

```
WHERE E.salary > 2000 AND
      NOT EXISTS (SELECT *
                  FROM Works W
                  WHERE W.eid = E.eid AND
                        W.months > 360);
```

The logical formula $L(q)$ is the following:

$$L(q) = \text{E.salary}>2000 \wedge \text{W.eid}=\text{E.eid} \wedge \text{W.months} \leq 360 \wedge (\text{E.eid} \geq 10000 \wedge \text{E.eid} \leq 99999) \wedge (\text{W.eid} \geq 10000 \wedge \text{W.eid} \leq 99999) \wedge (\text{W.months} \geq 0 \wedge \text{W.months} \leq 480)$$

5.2.5 Nested Queries Using ALL Clauses

The ALL clause in a WHERE clause of a query is of the form `<attribute> <rop> ALL(<simple_sel>)`. The ALL clause is satisfied if the comparison is *true* for all the values that the subquery returns. A query q in this category has the following syntax:

```
SELECT <attribute_list>
FROM <table_list>
WHERE [L'(q) <lop>]
          a <rop> ALL (SELECT a'
                       FROM <table_list>
                       WHERE L'(q'));
```

Recall that the domain of a' should match with the domain of attribute a, so that the comparison can be made. Suppose we have a sequence of queries $Q = q_1\ q_2 \ldots\ q_n$, and category(q_i) is ALL, where $1 \leq i \leq n$. For each query q_k in Q, where $k \neq i$ and $p_k = \exists$, we store the implicants obtained if type(q_k) $\in$ {SELECT, UPDATE} or type(q_k) = DELETE and $k > i$ in a set *Imp-Set* = {Imp_1, Imp_2,..., Imp_f}, $f \geq 0$. For each implicant *Imp* in *Imp-Set*, we determine if a tuple generated by *Imp* can be retrieved by q' and contains the

attribute a' in itself. If this is the case, we include the attribute a' belonging to *Imp* to the <rop> comparison in q by:

1. Suppose the *Imp* is denoted by $L^y(q_k)$. Then, we mark attribute a' with k as a subscript and y as a superscript.
2. When AssignSubscripts procedure is called and sees a' with k as a subscript and y as a superscript, it assigns the subscript of a' in the y^{th} implicant of q_k to the attribute a' of q'.

   We also generate a tuple for q' in case *Imp-Set* = ∅.

Algorithm 5.4 GenerateFormula-ALL

```
procedure GenerateFormula-ALL(q,L(q), Imp-Set)
   L(q) ← L'(q') ∧ [L'(q) <lop>] (a <rop> a')
   for each implicant L^Y(q_k) in Imp-Set do
      for each implicant Imp_j in L(q') do
         if(L^Y(q_k) has an attribute a_k that corresponds to the attribute a'
in q' and every predicate in Imp_j intersects with at least one predicate in
L^Y(q_k)) then
            L(q) ← L(q) ∧ (a <rop> a^Y_k)
         endif
      endfor
   endfor
   if(type(q) = SELECT and p = ∃) then
      for each attribute x used in <attribute_list> of q do
         L(q) ← L(q) ∧ getDomainConstraint(x)
      endfor
   endif
```

```
    if(type(q) = UPDATE) then
       for each attribute x used in SET clause of q do
          L(q) ← L(q) ∧ getDomainConstraint(x)
       endfor
    endif
    NegationFree(L(q))
    convert L(q) into DNF by algebraic manipulation
    for each implicant L^k(q) in L(q) do
       for each attribute x used in L^k(q) do
          L^k(q) ← L^k(q) ∧ getDomainConstraint(x)
       endfor
    endfor
endprocedure
```

Example 5.7. Consider the following query retrieving the id's of employees earning more than the employees working in the departments having a budget of $400.000:

```
SELECT E.eid
FROM Emp E
WHERE E.salary > ALL (SELECT E2.salary
                      FROM Emp E2, Works W, Dept D
                      WHERE E2.eid = W.eid AND
                            W.did = D.did AND
                            D.budget = 400000);
```

Suppose *Imp-Set* = ∅. According to the algorithm, the formula is the following:

$L(q)$ = (E2.eid = W.eid ∧ W.did = D.did ∧ D.budget = 400000) ∧ (E.salary > E2.salary) ∧ (E.eid ≥ 10000 ∧ E.eid ≤ 99999) ∧ (E2.eid ≥ 10000 ∧ E2.eid ≤ 99999) ∧ (W.eid ≥

10000 ∧ W.eid ≤ 99999) ∧ (W.did ≥ 100 ∧ W.did ≤ 999) ∧(D.did ≥ 100 ∧ D.did ≤ 999) ∧ (D.budget ≥ 0 ∧ D.budget ≤ 2000000) ∧ (E.salary ≥ 1000 ∧ E.salary ≤ 12000) ∧ (E2.salary ≥ 1000 ∧ E2.salary ≤ 12000)

Notice that if *Imp-Set* is not null, the algorithm introduces attributes generated for other tuples in the logical formula. However, this does not create additional conflicts.

5.2.6 Nested Queries Using ANY Clause

The ANY clause in a WHERE clause of a query is of the form `<attribute> <rop> ANY(<simple_sel>)`. The ANY clause is satisfied if the comparison is *true* for any value that the subquery returns. A query *q* in this category has the following syntax:

```
SELECT <attribute_list>
FROM <table_list>
WHERE [L'(q) <lop>] a <rop> ANY (SELECT a'
                                 FROM <table_list>
                                 WHERE L'(q'));
```

Recall that domain of *a'* should match with the domain of *a*, so that the comparison can be made. The only difference between GenerateFormula-ANY and GenerateFormula-ALL is that GenerateFormula-ANY contains disjunctions of (*a* <rop> *a'*) whereas GenerateFormula-ALL contains conjunctions of (*a* <rop> *a'*).

Algorithm 5.5 GenerateFormula-ANY

```
procedure GenerateFormula-ANY(q, L(q), Imp-Set)
   L(q) ← L'(q') ∧ [L'(q) <lop>] (a <rop> a')
   for each implicant Imp in Imp-Set do
      convertL(q') into DNF
   endfor
```

```
for each implicant L^Y(q_k) in Imp-Set do
   for each implicant Imp_j in L(q') do
      if (L^Y(q_k) has an attribute a_k equals to the attribute a' in q' and
      every predicate in Imp_i intersects with at least one predicate in
      Imp_j) then
         L(q) ← L(q) ∨ (a <rop> a^Y_k);
      endif
   endfor
endfor
if (type(q) = SELECT and p = ∃) then
   for each attribute x used in <attribute_list> of q do
      L(q) ← L(q) ∧ getDomainConstraint(x)
   endfor
endif
if (type(q) = UPDATE) then
    for each attribute x used in SET clause of q do
      L(q) ← L(q) ∧ getDomainConstraint(x)
   endfor
endif
NegationFree(L(q))
convert L(q) into DNF by algebraic manipulation
for each implicant L^k(q) in L(q) do
   for each attribute x used in L^k(q) do
      L^k(q) ← L^k(q) ∧ getDomainConstraint(x)
   endfor
endfor
```

```
endprocedure
```

Example 5.8. Consider the following query retrieving the id's of employees earning more than the employees working in the departments having 400000 as their budget value:

```
SELECT E.eid
FROM Emp E
WHERE E.salary > ANY (SELECT E2.salary
                      FROM Emp E2, Works W, Dept D
                      WHERE E2.eid = W.eid AND
                            W.did = D.did AND
                            D.budget = 400000);
```

Suppose *Imp-Set* = ∅. According to the algorithm, the formula is the following:

$L(q)$ = (E2.eid = W.eid ∧ W.did = D.did ∧ D.budget = 400000) ∧ (E.salary > E2.salary) ∧ (E.eid ≥ 10000 ∧ E.eid ≤ 99999) ∧ (E2.eid ≥ 10000 ∧ E2.eid ≤ 99999) ∧ (W.eid ≥ 10000 ∧ W.eid ≤ 99999) ∧ (W.did ≥ 100 ∧ W.did ≤ 999) ∧(D.did ≥ 100 ∧ D.did ≤ 999) ∧ (D.budget ≥ 0 ∧ D.budget ≤ 2000000) ∧ (E.salary ≥ 1000 ∧ E.salary ≤ 12000) ∧ (E2.salary ≥ 1000 ∧ E2.salary ≤ 12000)

5.3 Conflicts among Nested Queries and Integrity Constraints

Recall that we generate one logical formula for each nested query, which is in first order logic like the formulas generated for simple queries. Also, properties and query types remain the same. Therefore, no other conflict other than those presented in Chapter 4 can occur.

To generate the database instance, we use the same algorithm presented in Section 4.5, however, HandleTableConstraints and AssignSubscripts procedures change slightly, due

to the use of attributes taken from other implicants for nested queries using ALL and ANY clauses.

Consider an implicant $L^j(q_i)$ which contains a predicate using an attribute a_k^y from an implicant $L^y(q_k)$. Consider a table constraint B, where an implicant $L^l(B)$ contains a predicate (a θ *Value*), in which attribute a has the same name as a_k^y and belongs to the same relation. Then, if HandleConstraints decides to modify $L^j(q_i)$ by considering B, it modifies the formula such that $L^j(q_i) = L^j(q_i) \wedge L^l(B) \wedge (a_k^y$ θ *Value*). Modified procedure is shown below where the differences from the original version in Section 4.4.4 are in bold:

Algorithm 5.6 HandleTableConstraints

```
procedure HandleTableConstraints(Q,S)
   for each q in Q where p = ∃ do
      for each implicant L^k(q) do
         for each constraint B that is enforced on L^k(q) do
            convert L(B) into DNF
            for each implicant L^j(B) in L(B)
               if(all attributes in L^j(B) are also used in L^k(q))
                  L^k(q) ← L^k(q) ∧ L^j(B)
                  for each attribute a_k^y in L^k(q) do
                     if(L^j(B) contains a predicate(a θ Value))then
                        L^k(q) = L^k(q) ∧ (a_k^y θ Value)
                     endif
                  endfor
                  Simplify(L^k(q),S)
               endif
```

```
            endfor
         endfor
      endfor
   endfor
endprocedure
```

Again, consider an implicant $L^j(q_i)$ which contains a predicate using an attribute a_k^y from an implicant $L^y(q_k)$. The subscript that should be assigned to a_k^y can be determined only after subscripts are assigned to every attribute of $L^y(q_k)$. Therefore, for the implicant $L^j(q_i)$, assigning subscripts to attributes taken from other implicants are delayed until subscripts are assigned to every attribute belonging to $L^y(q_k)$. Also, consider two attributes *a* and *a'*, having different aliases of the same relation. Then, different subscripts have to be assigned to *a* and *a'*. We modify the AssignSubscripts procedure as follows:

Algorithm 5.7 AssignSubscripts

```
procedure AssignSubscripts(Q, S)
//Let S = {R1, R2,...,Rm}
//Each counter keeps track of the number of tuples to be inserted to
//corresponding R
   initialize m counters G = {G1, G2,...,Gm} to 0
   for each q in Q do
      for each implicant L^k(q) in L(q) do
         for each alias R' referred to in L^k(q) do
         //Note that G' is the corresponding counter of R'
            assign subscript G' to R' used in L^k(q)
            give L^k(q) as an input to CST
            if (CST does not report contradiction) then
```

```
            G' ← G' + 1
         else
            remove L^k(q) from L(q)
         endif
      endfor
   endfor
   for each implicant L^i(q_j) in L(q_j), where 1 ≤ j ≤ n, 1 ≤ i ≤ λ_j
      if(L^i(q_j) contains an attribute a_k^Y) then
      //R' is alias of a^Y_k and G^Y_k is the subscript of the attribute
      //a in L^Y(q_k)
         assign subscript G^Y_k to R'
         give L^i(q_j) as an input to CST;
         if (CST outputs contradiction) then
            remove L^i(q_j) from L(q)
         endif
      endif
   endfor
endfor
endprocedure
```

5.4 Generating Database Instances to Run a Test Suite

Given a test suite $\Omega = Q_1\ Q_2\ \ldots\ Q_w$, one can generate a sequence of databases $D' = D^1\ D^2 \ldots\ D^w$, such that test case Q_i can be run on the database D_i, $1 \leq i \leq w$. However, after each test case, all the tuples in the database should be dropped, and new tuples to run the next test case should be inserted. Since resetting a database is an expensive operation, we

extend our method to generate minimum number of database instances to run the test cases in a test suite in the given order to decrease the run time of a test suite. Reader will notice that we will run *CST* more than once, however, *CSTs* accesses memory and since disk access is much slower than memory access, performance gains will be significant for applications containing large number of queries.

In this thesis, we emphasize on the conflicts that can occur between the queries in a test case. Extending database instance generation method to run a test case reveals that queries between test cases in a sequence of test cases can also conflict with each other, which can be resolved in the same way. Recall that if a database instance cannot be generated for a test case, the reason was the incorrect input host variable instantiation or property assignment, which we assumed tester is responsible for them. However, if a contradiction occurs because we tried to generate a database instance to run two test cases one after the other, we do not assume tester's responsibility, and we handle it in our method

In our method, we partition the test suite Ω and obtain a partition $\Omega' = \{\Omega_1\ \Omega_2 \ldots\ \Omega_h\}$, where each element of the partition is a test suite. Also, each element defines a subsequence of the order given by Ω. Since a single database instance will be generated for each element of the partition, we aim to minimize h. The algorithm PartitionTestSuite we propose is as follows:

1. Obtain $L(Q_i)$ for each Q_i in Ω, $1 \leq i \leq w$. If *CST* reports contradiction for $L(Q_i)$, prompt the tester to intervene and solve the cause of the contradiction by assigning different values to input host variables.
2. Initialize $\Omega' = \Omega_1 = Q_1$.
3. Suppose that $\Omega' = \Omega_1\ \Omega_2 \ldots\ \Omega_k$, $k \leq h$ and the last test case of Ω_k is Q_j, $j \leq w$. For each Q_i in Ω, $j < i \leq w$, Q^{ki} is the concatenation of the queries in Ω_k with Q_i. Obtain $L(Q^{ki})$

by calling GenerateDatabaseInstance procedure. If CST runs without reporting a contradiction for $L(Q^{ki})$, then append Q_i to Ω_k. Otherwise, initialize a new partition $\Omega_{k+1} = Q_i$ and append it to Ω' ($= \Omega' \; \Omega_{k+1}$).

4. Repeat Step 3 until each query is assigned to a partition.

We give a pseudocode of this algorithm below:

Algorithm 5.8 PartitionTestSuite

```
procedure PartitionTestSuite(Ω, Ω')
   for Q_i in Ω, 1 ≤ i ≤ w do
      L(Q_i) ← GenerateDatabaseInstance(Q_i, P, S)
      if(CST reports contradiction for L(Q_i))then
         the tester modifies Q_i to resolve contradiction
         return failure
      endif
   endfor
   Ω_1 ← Q_1
   Ω' ← Ω_1
   Let Ω' be Ω_1 Ω_2 ... Ω_k, k ≤ h and Q_j be the last test case of Ω_k
   for each Q_i in Ω, j ≤ i ≤ w
      Q^ki ← Ω_k + Q_i
      L(Q^ki) ← GenerateDatabaseInstance(Q^ki, P^ki, S)
      if(CST reports contradiction for L(Q^ki)) then
         Ω_{k+1} ← Q_i
         Ω' ← Ω' ∪ Ω_{k+1}
       else
       Ω_k ← Ω_k + Q_i
```

```
      endif
   endfor
endprocedure
```

The algorithm GenerateInstances4TestSuite generates database instances by calling ObtainCSTInput algorithm presented in Section 4.5 for each partition.

Algorithm 5.9. GenerateInstances4TestSuite

```
procedure GenerateInstances4TestSuite(Ω')
   for each Ω_e in Ω' do
      Let Q^e be the concatenation of the queries of the test cases in Ω_e
      L(Q^e) ← GenerateDatabaseInstance (Q^e, P^e, S)
      D^e ← Solution of CST given the input L(Q^e)
   endfor
endprocedure
```

5.5 Example of Database Instance Generation with Nested Queries

In this section, we will generate D_0 for a test suite consisting of three test cases $\Omega = Q_1\ Q_2\ Q_3$. Figure 5.3, 5.4 and 5.5 illustrates the queries of the test cases Q_1, Q_2, and Q_3 respectively. Output of PartitionTestSuite algorithm is the concatenation of the queries shown in Figures 5.3, 5.4 and 5.5, which is shown in Figure 5.6. Note that if CST reports a contradiction for $L(Q_1)$, tester should modify the input host variables to eliminate the contradiction. In the example below, we will show that one database instance can be generated to run all test cases in $\Omega = Q_1\ Q_2\ Q_3$. This proves that a single database instance can be generated to run Q_1Q_2, however, we will leave the verification of this intermediary step of PartitionTestSuite algorithm to the reader.

```
q1 (∃): DELETE FROM Emp
         WHERE eid IN (SELECT W.eid FROM Works W
                       WHERE W.months > 360)
q2 (∃): UPDATE Dept SET budget = budget + 50000 WHERE budget > 300000
```

Figure 5.3. Test case Q_1 with instantiated host variables and the properties given by tester

```
q1 (∃): SELECT E.eid
        FROM Emp E
        WHERE E.salary > ALL (SELECT E2.salary
                              FROM Emp E2, Works W, Dept D
                              WHERE E2.eid = W.eid AND
                                    W.did = D.did AND
                                    D.budget > 400000);

q2 (∄): SELECT D.did FROM Dept D WHERE D.budget > 500000
```

Figure 5.4. Test case Q_2 with instantiated host variables and the properties given by tester

```
q1 (∃): SELECT E.salary
        FROM  Emp E
        WHERE E.eid IN (SELECT W.eid
                        FROM Works W, Dept D
                        WHERE W.did = D.did AND W.months > 240
                              AND D.budget= 210000)
```

Figure 5.5. Test case Q_3 with instantiated host variables and the properties given by tester

```
q1 (∃): DELETE FROM Emp
         WHERE eid IN (SELECT W.eid FROM Works W
                       WHERE  W.months > 360)

q2 (∃): UPDATE Dept SET budget = budget + 50000 WHERE budget > 300000

q3 (∃): SELECT E.eid
         FROM Emp E
         WHERE E.salary > ALL (SELECT E2.salary
                               FROM Emp E2, Works W, Dept D
                               WHERE E2.eid = W.eid AND
                                     W.did = D.did AND
                                     D.budget > 400000);

q4 (∄): SELECT D.did FROM Dept D WHERE D.budget > 500000

q5 (∃): SELECT E.salary
         FROM  Emp E
         WHERE E.eid IN (SELECT W.eid
                         FROM Works W, Dept D
                         WHERE W.did = D.did AND W.months > 240
                               AND D.budget= 210000)
```

Figure 5.6. Partition Ω'

We illustrate each step of the proposed method in Section 4.5 as we did in Section 4.6. We start by obtaining the logical formulas of each query in Ω'. Obtaining logical formulas for each nested query category is illustrated earlier in this chapter, and obtaining logical formulas for simple queries were discussed in Chapter 4. For q_1 GenerateFormula-N-IN, for q_2 ObtainLogicalFormula, for q_3 GenerateFormula-ALL, for q_4 ObtainLogicalFormula, and for q_5 GenerateFormula-N-IN will be called. These formulas will be simplified using Simplify procedure and we obtain the following formulas:

$$L(q_1) = (\text{E.eid} = \text{W.eid}) \wedge (\text{W.months} > 360) \wedge (\text{E.eid} \geq 10000 \wedge \text{E.eid} \leq 99999) \wedge (\text{W.eid} \geq 10000 \wedge \text{W.eid} \leq 99999) \wedge (\text{W.months} \geq 0 \wedge \text{W.months} \leq 480)$$

After Simplify, we obtain:

$$L(q_1) = (\text{E.eid} = \text{W.eid} \wedge \text{E.eid} \geq 10000 \wedge \text{E.eid} \leq 99999 \wedge \text{W.eid} \geq 10000 \wedge \text{W.eid} \leq$$

99999 ∧ W.months > 360 ∧ W.months ≤ 480)

$L(q_2)$ = (D.budget > 300000) ∧ (D.budget ≥ 0 ∧ D.budget ≤ 2000000)

After Simplify, we obtain:

$L(q_2)$ = (D.budget > 300000 ∧ D.budget ≤ 2000000)

As described in Section 5.2.5, we need the logical formulas of every query in Ω to generate $L(q_3)$, therefore, we defer obtaining this formula till we obtain the others.

$L(q_4)$ = (D.budget > 500000) ∧ (D.did ≥ 100 ∧ D.did ≤ 999) ∧ (D.budget ≥ 0 ∧ D.budget ≤ 2000000)

After Simplify, we obtain:

$L(q_4)$ = (D.did ≥ 100 ∧ D.did ≤ 999 ∧ D.budget > 500000 ∧ D.budget ≤ 2000000)

$L(q_5)$ = (E.eid = W.eid) ∧ (W.did = D.did ∧ W.months > 240 ∧ D.budget = 210000) ∧ (E.eid ≥ 10000 ∧ E.eid ≤ 99999) ∧ (W.eid ≥ 10000 ∧ W.eid ≤ 99999) ∧ (W.did ≥ 100 ∧ W.did ≤ 999) ∧ (W.months ≥ 0 ∧ W.months ≤ 480) ∧ (D.did ≥ 100 ∧ D.did ≤ 999) ∧ (D.budget ≥ 0 ∧ D.budget ≤ 2000000) ∧ (E.salary ≥ 1000 ∧ E.salary ≤ 12000)

After Simplify, we obtain:

$L(q_5)$ = (E.eid = W.eid ∧ W.did = D.did ∧ D.budget = 210000 ∧ E.eid ≥ 10000 ∧ E.eid ≤ 99999 ∧ W.eid ≥ 10000 ∧ W.eid ≤ 99999 ∧ W.did ≥ 100 ∧ W.did ≤ 999 ∧ W.months > 240 ∧ W.months ≤ 480 ∧ D.did ≥ 100 ∧ D.did ≤ 999 ∧ D.budget ≥ 0 ∧ D.budget ≤ 2000000 ∧ E.salary ≥ 1000 ∧ E.salary ≤ 12000)

Note that GenerateFormula-ALL generates the formula for $L(q_3)$ by considering every tuple to be generated for the database instance, which is identified by the parameter

Imp-Set = $\{L^1(q_2), L^1(q_5)\}$. GenerateFormula-ALL first checks if $L^1(q_2)$ has E.salary attribute. Since $L^1(q_2)$ does not contain this attribute, it will not be considered when generating $L(q_3)$. However, $L^1(q_5)$ contains E.salary, and GenerateFormula-ALL procedure checks if every predicate in $L(q_3')$, which is $L'(q_3')$ = (E2.eid = W.eid $\wedge$ W.did = D.did $\wedge$ D.budget > 400000), intersects with a predicate in $L^1(q_5)$ and this check returns true. Thus, this procedure takes into account the fact that tuple generated for $L(q_5)$ can be retrieved by q_3'. Firstly, GenerateFormula-ALL will produce a logical formula without domain constraints of attributes and without considering the effect of q_3', which is shown below.

$L(q_3)$ = (E2.eid = W.eid $\wedge$ W.did = D.did $\wedge$ D.budget > 400000 $\wedge$ E.salary > E2.salary)

Note that E2 is an alias for *Emp* relation. Thus, E2.salary is different from E_2.salary.

Then, GenerateFormula-ALL checks every implicant generated so far to find implicants using E.salary attribute, and checks if tuples generated for such implicants can be obtained by $L(q_3)$. Since among the elements $L(q_2)$ and $L(q_5)$ of Imp-Set, only $L^1(q_5)$ contains E.salary and every predicate of $L(q_3)$ intersects with at least one predicate in $L^1(q_5)$, $L(q_3)$ will be conjuncted with (E.salary > E^1_5.salary) by the GenerateFormula-ALL procedure.

$L(q_3)$ = (E2.eid = W.eid $\wedge$ W.did = D.did $\wedge$ D.budget > 400000 $\wedge$ E.salary > E2.salary $\wedge$
E.salary > E^1_5.salary)

Then, the procedure adds the domain constraints of E.eid in <attribute_list> of q_3 to $L(q_3)$ and obtains the following formula:

$L(q_3)$ = (E2.eid = W.eid $\wedge$ W.did = D.did $\wedge$ D.budget > 400000 $\wedge$ E.salary > E2.salary $\wedge$
E.salary > E^1_5.salary $\wedge$ E.eid $\geq$ 10000 $\wedge$ E.eid $\leq$ 99999)

GenerateFormula-ALL applies NegationFree to $L(q_3)$ and this formula is converted

into DNF. $L(q_3)$ does not contain any negations and is already in DNF and has one implicant, therefore the above formula will not be modified. The procedure ends by conjuncting domain constraints of attributes used in the implicants of $L^1(q_3)$, and the outcome is shown below.

$L(q_3)$ = (E2.eid = W.eid $\wedge$ W.did = D.did $\wedge$ D.budget > 400000 $\wedge$ E.salary > E2.salary $\wedge$ E.salary > E^1_5.salary $\wedge$ E.eid $\geq$ 10000 $\wedge$ E.eid $\leq$ 99999 $\wedge$ W.eid $\geq$ 10000 $\wedge$ W.eid $\leq$ 99999 $\wedge$ E2.eid $\geq$ 10000 $\wedge$ E2.eid $\leq$ 99999 $\wedge$ D.did $\geq$ 100 $\wedge$ D.did $\leq$ 999 $\wedge$ W.did $\geq$ 100 $\wedge$ W.did $\leq$ 999 $\wedge$ E.salary $\geq$ 1000 $\wedge$ E.salary $\leq$ 12000 $\wedge$ E2.salary $\geq$ 1000 $\wedge$ E2.salary $\leq$ 12000 $\wedge$ D.budget $\geq$ 0 $\wedge$ D.budget $\leq$ 2000000)

After Simplify, we obtain:

$L(q_3)$ = (E2.eid = W.eid $\wedge$ W.did = D.did $\wedge$ E.salary > E2.salary $\wedge$ E.salary > E^1_5.salary $\wedge$ E.eid $\geq$ 10000 $\wedge$ E.eid $\leq$ 99999 $\wedge$ W.eid $\geq$ 10000 $\wedge$ W.eid $\leq$ 99999 $\wedge$ E2.eid $\geq$ 10000 $\wedge$ E2.eid $\leq$ 99999 $\wedge$ D.did $\geq$ 100 $\wedge$ D.did $\leq$ 999 $\wedge$ W.did $\geq$ 100 $\wedge$ W.did $\leq$ 999 $\wedge$ E.salary $\geq$ 1000 $\wedge$ E.salary $\leq$ 12000 $\wedge$ E2.salary $\geq$ 1000 $\wedge$ E2.salary $\leq$ 12000 $\wedge$ D.budget > 400000 $\wedge$ D.budget $\leq$ 2000000)

Note that we do not need to put domain constraint for E^1_5.salary because it already exists in $L(q_5)$ in the form of E.salary. In Step 2, DetectAllConflicts procedure identifies the conflicts and saves them into conflict set *CS*. We give *CS* and justify each conflict below.

$CS = \{L^1(q_1) \rightarrow_d L^1(q_5)$, $L^1(q_4) \rightarrow_n L^1(q_3)$, $L^1(q_4) \rightarrow_n L^1(q_2)$, $L^1(q_2) \rightarrow_u L^1(q_3)$, $L^1(q_2) \rightarrow_u L^1(q_2)$ $\}$

type(q_1) = DELETE and type(q_5) = SELECT, q_1 is executed before q_5 and $p_1 = p_2 = \exists$. Also, every predicate in $L^1(q_1)$ intersects with at least one predicate in $L^1(q_5)$, therefore, $L^1(q_1)$ is in DELETE conflict with $L^1(q_5)$.

type(q_4) = SELECT, $p_4 = \nexists$, $p_3 = \exists$ and type(q_3) = SELECT. WHERE clause of q_4 is (D.budget > 500000) and this predicate intersects with at least one predicate in $L^1(q_3)$. Therefore, $L^1(q_4) \rightarrow_n L^1(q_3)$.

type(q_4) = SELECT, $p_4 = \nexists$, $p_2 = \exists$ and type(q_2) = UPDATE. WHERE clause of q_4 is (D.budget > 500000) and this predicate intersects with at least one predicate in $L^1(q_2)$. Therefore, $L^1(q_4) \rightarrow_n L^1(q_2)$.

$p_2 = \exists$, $p_3 = \exists$, type(q_2) = UPDATE and type(q_3) = SELECT. Also, q_2 is executed before q_3 and *budget* attribute that is used in the SET clause of q_2 is used in the predicate (D.budget > 400000) in q_3. WHERE clause of q_2 is (budget > 300000) and it intersects with at least one predicate in $L^1(q_3)$, thus, $L^1(q_2) \rightarrow_u L^1(q_3)$. Also, $L^1(q_2) \rightarrow_u L^1(q_2)$ because an UPDATE query can be in UPDATE conflict with itself, if an attribute used in the SET clause is also used in WHERE clause, as DetectUPDConflict states.

In Step 3, we modify the logical formulas of the queries to resolve the conflicts. q_5, q_3 and q_2 will be affected. ResolveConflicts: first generates a temporary formula NEW as follows:

$$\text{NEW} = \neg L^1(q_1)$$
$$= \neg(\text{E.eid} = \text{W.eid} \wedge \text{E.eid} \geq 10000 \wedge \text{E.eid} \leq 99999 \wedge \text{W.eid} \geq 10000 \wedge \text{W.eid} \leq 99999 \wedge \text{W.months} > 360 \wedge \text{W.months} \leq 480)$$

Then, NegationFree is applied to NEW. The join predicate E.eid = W.eid and the domain constraints of E.eid and W.eid are taken out of negation, and we obtain the following:

$$\text{NEW} = (\text{E.eid} = \text{W.eid} \wedge \text{E.eid} \geq 10000 \wedge \text{E.eid} \leq 99999 \wedge \text{W.eid} \geq 10000 \wedge \text{W.eid} \leq 99999 \wedge (\text{W.months} \leq 360 \vee \text{W.months} > 480)$$

We obtain UPDATED- $L^1(q_5)$ by conjuncting $L^1(q_5)$ and NEW:

UPDATED- $L^1(q_5)$ = (E.eid = W.eid $\wedge$ W.did = D.did $\wedge$ D.budget = 210000 $\wedge$ E.eid $\geq$ 10000 $\wedge$ E.eid $\leq$ 99999 $\wedge$ W.eid $\geq$ 10000 $\wedge$ W.eid $\leq$ 99999 $\wedge$ W.did $\geq$ 100 $\wedge$ W.did $\leq$ 999 $\wedge$ W.months > 240 $\wedge$ W.months $\leq$ 480 $\wedge$ D.did $\geq$ 100 $\wedge$ D.did $\leq$ 999 $\wedge$ D.budget $\geq$ 0 $\wedge$ D.budget $\leq$ 2000000 $\wedge$ E.salary $\geq$ 1000 $\wedge$ E.salary $\leq$ 12000 $\wedge$ E.eid = W.eid $\wedge$ E.eid $\geq$ 10000 $\wedge$ E.eid $\leq$ 99999 $\wedge$ W.eid $\geq$ 10000 $\wedge$ W.eid $\leq$ 99999) $\wedge$ (W.months $\leq$ 360 $\vee$ W.months > 480)

After this step, ResolveConflicts will convert UPDATED- $L^1(q_5)$ into DNF and obtain the following:

UPDATED- $L^{1\text{-}1}(q_5)$ = (E.eid = W.eid $\wedge$ W.did = D.did $\wedge$ D.budget = 210000 $\wedge$ E.eid $\geq$ 10000 $\wedge$ E.eid $\leq$ 99999 $\wedge$ W.eid $\geq$ 10000 $\wedge$ W.eid $\leq$ 99999 $\wedge$ W.did $\geq$ 100 $\wedge$ W.did $\leq$ 999 $\wedge$ W.months > 240 $\wedge$ W.months $\leq$ 480 $\wedge$ D.did $\geq$ 100 $\wedge$ D.did $\leq$ 999 $\wedge$ D.budget $\geq$ 0 $\wedge$ D.budget $\leq$ 2000000 $\wedge$ E.salary $\geq$ 1000 $\wedge$ E.salary $\leq$ 12000 $\wedge$ E.eid = W.eid $\wedge$ E.eid $\geq$ 10000 $\wedge$ E.eid $\leq$ 99999 $\wedge$ W.eid $\geq$ 10000 $\wedge$ W.eid $\leq$ 99999 $\wedge$ W.months $\leq$ 360)

UPDATED- $L^{1\text{-}2}(q_5)$ = (E.eid = W.eid $\wedge$ W.did = D.did $\wedge$ D.budget = 210000 $\wedge$ E.eid $\geq$ 10000 $\wedge$ E.eid $\leq$ 99999 $\wedge$ W.eid $\geq$ 10000 $\wedge$ W.eid $\leq$ 99999 $\wedge$ W.did $\geq$ 100 $\wedge$ W.did $\leq$ 999 $\wedge$ W.months > 240 $\wedge$ W.months $\leq$ 480 $\wedge$ D.did $\geq$ 100 $\wedge$ D.did $\leq$ 999 $\wedge$ D.budget $\geq$ 0 $\wedge$ D.budget $\leq$ 2000000 $\wedge$ E.salary $\geq$ 1000 $\wedge$ E.salary $\leq$ 12000 $\wedge$ E.eid = W.eid $\wedge$ E.eid $\geq$ 10000 $\wedge$ E.eid $\leq$ 99999 $\wedge$ W.eid $\geq$ 10000 $\wedge$ W.eid $\leq$ 99999 $\wedge$ W.months > 480)

Simplify procedure eliminates UPDATED-$L^{1\text{-}2}(q_5)$ because W.months > 480 contradicts with the predicate W.months $\leq$ 480. In UPDATED-$L^{1\text{-}1}(q_5)$, the predicate W.months $\leq$ 480 is subsumed by W.months $\leq$ 360 and will be eliminated. Also, the

predicates E.eid = W.eid, E.eid ≥ 10000, E.eid ≤ 99999, W.eid ≥ 10000 and W.eid ≤ 99999 exists twice in the formula, therefore one of them is eliminated. After this step, $L^1(q_5)$ will be as follows:

$L^1(q_5)$ = (E.eid = W.eid ∧ W.did = D.did ∧ D.budget = 210000 ∧ E.eid ≥ 10000 ∧ E.eid ≤ 99999 ∧ W.eid ≥ 10000 ∧ W.eid ≤ 99999 ∧ W.did ≥ 100 ∧ W.did ≤ 999 ∧ W.months > 240 ∧ W.months ≤ 360 ∧ D.did ≥ 100 ∧ D.did ≤ 999 ∧ D.budget ≥ 0 ∧ D.budget ≤ 2000000 ∧ E.salary ≥ 1000 ∧ E.salary ≤ 12000)

$L^1(q_2)$ will be modified two times, one for ∄ conflict and one for UPDATE conflict. Below, we show the modification for ∄ conflict. First, ResolveConflicts procedure will create the formula NEW as follows:

NEW = ¬ $L^1(q_4)$

= ¬(D.did ≥ 100 ∧ D.did ≤ 999 ∧ D.budget > 500000 ∧ D.budget ≤ 2000000)

After NegationFree is applied to NEW, we obtain the following:

NEW = (D.did < 100 ∨ D.did > 999 ∨ D.budget ≤ 500000 ∨ D.budget > 2000000)

We obtain UPDATED- $L^1(q_2)$ by conjuncting $L^1(q_2)$ and NEW:

UPDATED- $L^1(q_2)$ = (D.budget > 300000 ∧ D.budget ≤ 2000000) ∧ (D.did < 100 ∨ D.did > 999 ∨ D.budget ≤ 500000 ∨ D.budget > 2000000)

After converting UPDATED- $L^1(q_2)$, following implicants will be generated.

UPDATED- $L^{1\text{-}1}(q_2)$ = (D.budget > 300000 ∧ D.budget ≤ 2000000 ∧ D.did < 100)

UPDATED- $L^{1\text{-}2}(q_2)$ = (D.budget > 300000 ∧ D.budget ≤ 2000000 ∧ D.did > 999)

UPDATED- $L^{1\text{-}3}(q_2)$ = (D.budget > 300000 ∧ D.budget ≤ 2000000 ∧ D.budget ≤ 500000)

UPDATED- $L^{1\text{-}4}(q_2)$ = (D.budget > 300000 ∧ D.budget ≤ 2000000 ∧ D.budget >

2000000)

UPDATED- $L^{1\text{-}1}(q_2)$, UPDATED- $L^{1\text{-}2}(q_2)$ and UPDATED- $L^{1\text{-}4}(q_2)$ will be eliminated by Simplify because they contain the predicates D.did < 100, D.did > 999 and D.budget > 2000000, which violate the domain constraint of D.did and D.budget. Also, D.budget ≤ 500000 subsumes D.budget ≤ 2000000. Thus, $L^1(q_2)$ will be:

$L^1(q_2)$ = (D.budget > 300000 ∧ D.budget ≤ 500000)

Then, $L^1(q_2)$ will be modified for the UPDATE conflict. Since budget value will be increased, new value might exceed the upper bound 500000. Therefore, ResolveConflicts procedure modifies the formula as follows:

$L^1(q_2)$ = (D.budget > 300000 ∧ D.budget + 50000 ≤ 500000)
= (D.budget > 300000 ∧ D.budget ≤ 450000)

$L(q_3)$ will be modified two times, one for ∄ conflict and one for UPDATE conflict. Below, we show the modification for ∄ conflict. ResolveConflicts first generates a temporary formula NEW as follows:

NEW = ¬ $L^1(q_4)$
= ¬(D.did ≥ 100 ∧ D.did ≤ 999 ∧ D.budget > 500000 ∧ D.budget ≤ 2000000)

After NegationFree is applied to NEW, we obtain the following:

NEW = (D.did < 100 ∨ D.did > 999 ∨ D.budget ≤ 500000 ∨ D.budget > 2000000)

We obtain UPDATED- $L^1(q_3)$ by conjuncting $L^1(q_3)$ and NEW:

UPDATED- $L^1(q_3)$ = (E2.eid = W.eid ∧ W.did = D.did ∧ E.salary > E2.salary ∧ E.salary > E^1_5.salary ∧ E.eid ≥ 10000 ∧ E.eid ≤ 99999 ∧ W.eid ≥ 10000 ∧ W.eid ≤ 99999 ∧ E2.eid ≥ 10000 ∧ E2.eid ≤ 99999 ∧ D.did ≥ 100 ∧ D.did ≤ 999 ∧ W.did ≥ 100

$\wedge$ W.did $\leq$ 999 $\wedge$ E.salary $\geq$ 1000 $\wedge$ E.salary $\leq$ 12000 $\wedge$ E2.salary $\geq$ 1000 $\wedge$ E2.salary $\leq$ 12000 $\wedge$ D.budget > 400000 $\wedge$ D.budget $\leq$ 2000000) $\wedge$ (D.did < 100 $\vee$ D.did > 999 $\vee$ D.budget $\leq$ 500000 $\vee$ D.budget > 2000000)

After converting UPDATED- $L^1(q_3)$, following implicants will be generated.

UPDATED- $L^{1\text{-}1}(q_3)$ = (E2.eid = W.eid $\wedge$ W.did = D.did $\wedge$ E.salary > E2.salary $\wedge$ E.salary > $E^1{}_5$.salary $\wedge$ E.eid $\geq$ 10000 $\wedge$ E.eid $\leq$ 99999 $\wedge$ W.eid $\geq$ 10000 $\wedge$ W.eid $\leq$ 99999 $\wedge$ E2.eid $\geq$ 10000 $\wedge$ E2.eid $\leq$ 99999 $\wedge$ D.did $\geq$ 100 $\wedge$ D.did $\leq$ 999 $\wedge$ W.did $\geq$ 100 $\wedge$ W.did $\leq$ 999 $\wedge$ E.salary $\geq$ 1000 $\wedge$ E.salary $\leq$ 12000 $\wedge$ E2.salary $\geq$ 1000 $\wedge$ E2.salary $\leq$ 12000 $\wedge$ D.budget > 400000 $\wedge$ D.budget $\leq$ 2000000 $\wedge$ D.did < 100)

UPDATED- $L^{1\text{-}2}(q_3)$ = (E2.eid = W.eid $\wedge$ W.did = D.did $\wedge$ E.salary > E2.salary $\wedge$ E.salary > $E^1{}_5$.salary $\wedge$ E.eid $\geq$ 10000 $\wedge$ E.eid $\leq$ 99999 $\wedge$ W.eid $\geq$ 10000 $\wedge$ W.eid $\leq$ 99999 $\wedge$ E2.eid $\geq$ 10000 $\wedge$ E2.eid $\leq$ 99999 $\wedge$ D.did $\geq$ 100 $\wedge$ D.did $\leq$ 999 $\wedge$ W.did $\geq$ 100 $\wedge$ W.did $\leq$ 999 $\wedge$ E.salary $\geq$ 1000 $\wedge$ E.salary $\leq$ 12000 $\wedge$ E2.salary $\geq$ 1000 $\wedge$ E2.salary $\leq$ 12000 $\wedge$ D.budget > 400000 $\wedge$ D.budget $\leq$ 2000000) $\wedge$ D.did > 999)

UPDATED- $L^{1\text{-}3}(q_3)$ = (E2.eid = W.eid $\wedge$ W.did = D.did $\wedge$ E.salary > E2.salary $\wedge$ E.salary > $E^1{}_5$.salary $\wedge$ E.eid $\geq$ 10000 $\wedge$ E.eid $\leq$ 99999 $\wedge$ W.eid $\geq$ 10000 $\wedge$ W.eid $\leq$ 99999 $\wedge$ E2.eid $\geq$ 10000 $\wedge$ E2.eid $\leq$ 99999 $\wedge$ D.did $\geq$ 100 $\wedge$ D.did $\leq$ 999 $\wedge$ W.did $\geq$ 100 $\wedge$ W.did $\leq$ 999 $\wedge$ E.salary $\geq$ 1000 $\wedge$ E.salary $\leq$ 12000 $\wedge$ E2.salary $\geq$ 1000 $\wedge$ E2.salary $\leq$ 12000 $\wedge$ D.budget > 400000 $\wedge$ D.budget $\leq$ 2000000 $\wedge$ D.budget $\leq$ 500000)

UPDATED- $L^{1\text{-}4}(q_3)$ = (E2.eid = W.eid $\wedge$ W.did = D.did $\wedge$ E.salary > E2.salary $\wedge$ E.salary > $E^1{}_5$.salary $\wedge$ E.eid $\geq$ 10000 $\wedge$ E.eid $\leq$ 99999 $\wedge$ W.eid $\geq$ 10000 $\wedge$

W.eid ≤ 99999 ∧ E2.eid ≥ 10000 ∧ E2.eid ≤ 99999 ∧ D.did ≥ 100 ∧ D.did ≤ 999 ∧ W.did ≥ 100 ∧ W.did ≤ 999 ∧ E.salary ≥ 1000 ∧ E.salary ≤ 12000 ∧ E2.salary ≥ 1000 ∧ E2.salary ≤ 12000 ∧ D.budget > 400000 ∧ D.budget ≤ 2000000 ∧ D.budget > 2000000)

UPDATED- $L^{1\text{-}1}(q_3)$, UPDATED- $L^{1\text{-}2}(q_3)$ and UPDATED- $L^{1\text{-}4}(q_3)$ will be eliminated by Simplify because they contain the predicates D.did < 100, D.did > 999 and D.budget > 2000000, which violate the domain constraint of D.did and D.budget. Also, D.budget ≤ 500000 subsumes D.budget ≤ 2000000. Thus, $L^1(q_3)$ will be:

$L^1(q_3)$ = (E2.eid = W.eid ∧ W.did = D.did ∧ E.salary > E2.salary ∧ E.salary > E^1_5.salary ∧ E.eid ≥ 10000 ∧ E.eid ≤ 99999 ∧ W.eid ≥ 10000 ∧ W.eid ≤ 99999 ∧ E2.eid ≥ 10000 ∧ E2.eid ≤ 99999 ∧ D.did ≥ 100 ∧ D.did ≤ 999 ∧ W.did ≥ 100 ∧ W.did ≤ 999 ∧ E.salary ≥ 1000 ∧ E.salary ≤ 12000 ∧ E2.salary ≥ 1000 ∧ E2.salary ≤ 12000 ∧ D.budget > 400000 ∧ D.budget ≤ 500000)

Then, ResolveConflicts procedure modifies $L^1(q_3)$ to resolve UPDATE conflict between q_2 and q_3. Again, only the predicate determining the upper bound of *budget* will be modified.

$L^1(q_3)$ = (E2.eid = W.eid ∧ W.did = D.did ∧ E.salary > E2.salary ∧ E.salary > E^1_5.salary ∧ E.eid ≥ 10000 ∧ E.eid ≤ 99999 ∧ W.eid ≥ 10000 ∧ W.eid ≤ 99999 ∧ E2.eid ≥ 10000 ∧ E2.eid ≤ 99999 ∧ D.did ≥ 100 ∧ D.did ≤ 999 ∧ W.did ≥ 100 ∧ W.did ≤ 999 ∧ E.salary ≥ 1000 ∧ E.salary ≤ 12000 ∧ E2.salary ≥ 1000 ∧ E2.salary ≤ 12000 ∧ D.budget > 400000 ∧ D.budget + 50000 ≤ 500000)

= (E2.eid = W.eid ∧ W.did = D.did ∧ E.salary > E2.salary ∧ E.salary > E^1_5.salary ∧ E.eid ≥ 10000 ∧ E.eid ≤ 99999 ∧ W.eid ≥ 10000 ∧ W.eid ≤ 99999 ∧ E2.eid ≥ 10000 ∧ E2.eid ≤ 99999 ∧ D.did ≥ 100 ∧ D.did ≤ 999 ∧ W.did ≥ 100 ∧ W.did ≤

999 ∧ E.salary ≥ 1000 ∧ E.salary ≤ 12000 ∧ E2.salary ≥ 1000 ∧ E2.salary ≤ 12000 ∧ D.budget > 400000 ∧ D.budget ≤ 450000)

In Step 4, we further modify the logical formulas of the queries by considering table constraints. The only table constraint enforced in S is B = (age <= 70 OR salary > 3500). HandleTableConstraints procedure converts this constraint into DNF, and obtains two implicants:

$L^1(B)$ = (age ≤ 70)

$L^2(B)$ = (salary > 3500)

For each implicant $L^k(q)$ of the logical formula of each query q whose p is ∃, HandleTableConstraints conjuncts $L^1(B)$ and/or $L^2(B)$ to $L^k(q)$ if age and/or salary are used in $L^k(q)$. $L^1(q_3)$ and $L^1(q_5)$ will be modified since they contain predicates using E.salary attribute.

$L^1(q_3)$ = (E2.eid = W.eid ∧ W.did = D.did ∧ E.salary > E2.salary ∧ E.salary > E^1_5.salary ∧ E.eid ≥ 10000 ∧ E.eid ≤ 99999 ∧ W.eid ≥ 10000 ∧ W.eid ≤ 99999 ∧ E2.eid ≥ 10000 ∧ E2.eid ≤ 99999 ∧ D.did ≥ 100 ∧ D.did ≤ 999 ∧ W.did ≥ 100 ∧ W.did ≤ 999 ∧ E.salary ≥ 1000 ∧ E.salary ≤ 12000 ∧ E2.salary ≥ 1000 ∧ E2.salary ≤ 12000 ∧ D.budget > 400000 ∧ D.budget ≤ 450000 ∧ E.salary > 3500 ∧ E2.salary > 3500)

$L^1(q_5)$ = (E.eid = W.eid ∧ W.did = D.did ∧ D.budget = 210000 ∧ E.eid ≥ 10000 ∧ E.eid ≤ 99999 ∧ W.eid ≥ 10000 ∧ W.eid ≤ 99999 ∧ W.did ≥ 100 ∧ W.did ≤ 999 ∧ W.months > 240 ∧ W.months ≤ 360 ∧ D.did ≥ 100 ∧ D.did ≤ 999 ∧ D.budget ≥ 0 ∧ D.budget ≤ 2000000 ∧ E.salary ≥ 1000 ∧ E.salary ≤ 12000 ∧ E.salary > 3500)

Below are the modified formulas after simplification.

$L^1(q_3)$ = (E2.eid = W.eid $\wedge$ W.did = D.did $\wedge$ E.salary > E2.salary $\wedge$ E.salary > E^1_5.salary $\wedge$ E.eid $\geq$ 10000 $\wedge$ E.eid $\leq$ 99999 $\wedge$ W.eid $\geq$ 10000 $\wedge$ W.eid $\leq$ 99999 $\wedge$ E2.eid $\geq$ 10000 $\wedge$ E2.eid $\leq$ 99999 $\wedge$ D.did $\geq$ 100 $\wedge$ D.did $\leq$ 999 $\wedge$ W.did $\geq$ 100 $\wedge$ W.did $\leq$ 999 $\wedge$ E.salary > 3500 $\wedge$ E.salary $\leq$ 12000 $\wedge$ E2.salary > 3500 $\wedge$ E2.salary $\leq$ 12000 $\wedge$ D.budget > 400000 $\wedge$ D.budget $\leq$ 450000)

$L^1(q_5)$ = (E.eid = W.eid $\wedge$ W.did = D.did $\wedge$ D.budget = 210000 $\wedge$ E.eid $\geq$ 10000 $\wedge$ E.eid $\leq$ 99999 $\wedge$ W.eid $\geq$ 10000 $\wedge$ W.eid $\leq$ 99999 $\wedge$ W.did $\geq$ 100 $\wedge$ W.did $\leq$ 999 $\wedge$ W.months > 240 $\wedge$ W.months $\leq$ 360 $\wedge$ D.did $\geq$ 100 $\wedge$ D.did $\leq$ 999 $\wedge$ D.budget $\geq$ 0 $\wedge$ D.budget $\leq$ 2000000 $\wedge$ E.salary > 3500 $\wedge$ E.salary $\leq$ 12000)

In Step 5, we modify the formulas such that domain constraints of the attributes defining primary keys of each relation in the formula, and each relation referenced by the relations in the formula will be incorporated. The implicants will be modified as follows.

$L^1(q_1)$ references *Works* relation, however does not contain W.did = D.did and the domain constraints of W.did and D.did. These predicates are added to $L^1(q_1)$ and we obtain the following:

$L^1(q_1)$ = (E.eid = W.eid $\wedge$ E.eid $\geq$ 10000 $\wedge$ E.eid $\leq$ 99999 $\wedge$ W.eid $\geq$ 10000 $\wedge$ W.eid $\leq$ 99999 $\wedge$ W.months > 360 $\wedge$ W.months $\leq$ 480 $\wedge$ D.did $\geq$ 100 $\wedge$ D.did $\leq$ 999 $\wedge$ W.did $\geq$ 100 $\wedge$ W.did $\leq$ 999 $\wedge$ W.did = D.did)

To $L^1(q_2)$, domain constraints of primary key of *Dept* relation will be added:

$L^1(q_2)$ = (D.budget > 300000 $\wedge$ D.budget $\leq$ 450000) $\wedge$ (D.did $\geq$ 100 $\wedge$ D.did $\leq$ 999)

$L^1(q_3)$, $L^1(q_4)$ and $L^1(q_5)$ will not be modified since they contain the domain constraints of primary and/or foreign key constraints.

$L^1(q_3)$ = (E2.eid = W.eid ∧ W.did = D.did ∧ E.salary > E2.salary ∧ E.salary > E^1_5.salary ∧ E.eid ≥ 10000 ∧ E.eid ≤ 99999 ∧ W.eid ≥ 10000 ∧ W.eid ≤ 99999 ∧ E2.eid ≥ 10000 ∧ E2.eid ≤ 99999 ∧ D.did ≥ 100 ∧ D.did ≤ 999 ∧ W.did ≥ 100 ∧ W.did ≤ 999 ∧ E.salary > 3500 ∧ E.salary ≤ 12000 ∧ E2.salary > 3500 ∧ E2.salary ≤ 12000 ∧ D.budget > 400000 ∧ D.budget ≤ 450000)

$L^1(q_4)$ = (D.did ≥ 100 ∧ D.did ≤ 999 ∧ D.budget > 500000 ∧ D.budget ≤ 2000000)

$L^1(q_5)$ = (E.eid = W.eid ∧ W.did = D.did ∧ D.budget = 210000 ∧ E.eid ≥ 10000 ∧ E.eid ≤ 99999 ∧ W.eid ≥ 10000 ∧ W.eid ≤ 99999 ∧ W.did ≥ 100 ∧ W.did ≤ 999 ∧ W.months > 240 ∧ W.months ≤ 360 ∧ D.did ≥ 100 ∧ D.did ≤ 999 ∧ D.budget ≥ 0 ∧ D.budget ≤ 2000000 ∧ E.salary ≥ 1000 ∧ E.salary ≤ 12000 ∧ E.salary > 3500)

In Step 6, we call the modified AssignSubscripts procedure shown in Algorithm 5.7, which assigns subscripts to the implicants of all queries with ∃ property. None of the implicants in our example corresponds to a contradiction. The modification of AssignSubscripts procedure allows us to find the assigned subscript of $E^1_5.salary$ used in $L^1(q_3)$ from $L^1(q_5)$ as E_3.salary. Note that E2.salary in $L^1(q_3)$ becomes E_2.salary after AssignSubscripts procedure. The logical formulas after this step are shown below:

$L^1(q_1)$ = (E_0.eid = W_0.eid ∧ E_0.eid ≥ 10000 ∧ E_0.eid ≤ 99999 ∧ W_0.eid ≥ 10000 ∧ W_0.eid ≤ 99999 ∧ W_0.months ≥ 360 ∧ W_0.months ≤ 480 ∧ D_0.did ≥ 100 ∧ D_0.did ≤ 999 ∧ W_0.did ≥ 100 ∧ W_0.did ≤ 999 ∧ W_0.did = D_0.did)

$L^1(q_2)$ = (D_1.budget > 300000 ∧ D_1.budget ≤ 450000) ∧ (D_1.did ≥ 100 ∧ D_1.did ≤ 999)

$L^1(q_3)$ = (E_2.eid = W_1.eid ∧ W_1.did = D_2.did ∧ E_1.salary > E_2.salary ∧ E_1.salary > E_3.salary ∧ E_1.eid ≥ 10000 ∧ E_1.eid ≤ 99999 ∧ W_1.eid ≥ 10000 ∧ W_1.eid ≤

$99999 \wedge E_2.eid \geq 10000 \wedge E_2.eid \leq 99999 \wedge D_2.did \geq 100 \wedge D_2.did \leq 999 \wedge W_1.did \geq 100 \wedge W_1.did \leq 999 \wedge E_1.salary > 3500 \wedge E_1.salary \leq 12000 \wedge E_2.salary > 3500 \wedge E_2.salary \leq 12000 \wedge D_2.budget > 400000 \wedge D_2.budget \leq 450000)$

$L^1(q_5) = (E_3.eid = W_2.eid \wedge W_2.did = D_3.did \wedge D_3.budget = 210000 \wedge E_3.eid \geq 10000 \wedge E_3.eid \leq 99999 \wedge W_2.eid \geq 10000 \wedge W_2.eid \leq 99999 \wedge W_2.did \geq 100 \wedge W_2.did \leq 999 \wedge W_2.months > 240 \wedge W_2.months \leq 360 \wedge D_3.did \geq 100 \wedge D_3.did \leq 999 \wedge D_3.budget \geq 0 \wedge D_3.budget \leq 2000000 \wedge E_3.salary > 3500 \wedge E_3.salary \leq 12000)$

As in Chapter 4, we will show the formulas for foreign and primary key constraints along with the constraints in HySat format in Figure 5.8. Figure 5.7 illustrates the DECL section of $L(\Omega)$ and Figure 5.8 shows the EXPR section of $L(\Omega)$. Domain constraints of attributes are stated in DECL section as we did in Chapter 4.

```
DECL
int [10000,99999] Emp_0_eid;
int [10000,99999] Works_0_eid;
int [0,480] Works_0_months;
int [100,999] Works_0_did;
int [100, 999] Dept_0_did;
int [0,2000000] Dept_1_budget;
int [100, 999] Dept_1_did;
int [10000, 99999] Emp_2_eid;
int [10000,99999] Works_1_eid;
int [100,999] Works_1_did;
int [100,999] Dept_2_did;
int [0,2000000] Dept_2_budget;
int [1000,12000] Emp_1_salary;
int [1000,12000] Emp_2_salary;
int [1000,12000] Emp_3_salary;
int [10000,99999] Emp_1_eid;
int [10000,99999] Emp_3_eid;
int [10000, 99999] Works_2_eid;
int [100,999] Works_2_did;
int [0,480] Works_2_months;
int [0,2000000] Dept_3_budget;
int [100,999] Dept_3_did;
```

Figure 5.7. DECL section of the final constraint $L(\Omega)$

```
EXPR
(Emp_0_eid = Works_0_eid and Works_0_months > 360 and
 Works_0_did = Dept_0_did);

(Dept_1_budget > 300000 and Dept_1_budget <= 450000);

(Emp_2_eid = Works_1_eid and Works_1_did = Dept_2_did and
 Dept_2_budget > 400000 and Emp_1_salary > Emp_2_salary and
 Emp_1_salary > Emp_3_salary and Emp_1_salary > 3500 and
 Emp_2_salary > 3500 and Dept_2_budget <= 450000);

(Emp_3_eid = Works_2_eid and Works_2_did = Dept_3_did and
 Works_2_months > 240 and Dept_3_budget = 210000 and Works_2_months
 <= 360 and Emp_3_salary > 3500);

--PK constraint for Emp table
(Emp_0_eid != Emp_1_eid and Emp_0_eid != Emp_2_eid);
(Emp_0_eid != Emp_3_eid and Emp_1_eid != Emp_2_eid);
(Emp_1_eid != Emp_3_eid and Emp_2_eid != Emp_3_eid);

--PK Constraint for Works table
(Works_0_eid != Works_1_eid or Works_0_did != Works_1_did);
(Works_0_eid != Works_2_eid or Works_0_did != Works_2_did);
(Works_1_eid != Works_2_eid or Works_1_did != Works_2_did);

--PK for Dept table
(Dept_0_did != Dept_1_did and Dept_0_did != Dept_2_did);
(Dept_0_did != Dept_3_did and Dept_1_did != Dept_2_did and
 Dept_1_did != Dept_3_did and Dept_2_did != Dept_3_did);

--FK Works
(Works_0_eid = Emp_0_eid or Works_0_eid = Emp_1_eid or
 Works_0_eid = Emp_2_eid or Works_0_eid = Emp_3_eid);
(Works_1_eid = Emp_0_eid or Works_1_eid = Emp_1_eid or
 Works_1_eid = Emp_2_eid or Works_1_eid = Emp_3_eid);
(Works_2_eid = Emp_0_eid or Works_2_eid = Emp_1_eid or
 Works_2_eid = Emp_2_eid or Works_2_eid = Emp_3_eid);

(Works_0_did = Dept_0_did or Works_0_did = Dept_1_did or
 Works_0_did = Dept_2_did or Works_0_did = Dept_3_did);
(Works_1_did = Dept_0_did or Works_1_did = Dept_1_did or
 Works_1_did = Dept_2_did or Works_1_did = Dept_3_did);
(Works_2_did = Dept_0_did or Works_2_did = Dept_1_did or
 Works_2_did = Dept_2_did or Works_2_did = Dept_3_did);
```

Figure 5.8. EXPR section of the final constraint $L(\Omega)$

The database instance generated is given below. Due to space constraints, we will not run each query in here like we did in Chapter 4. It is left to the reader to verify that conflicts are avoided and properties of each query are satisfied.

Table 5.1(a). Contents of *Emp* table.

Eid	*Name*	*Age*	*Salary*
62629			
66134			7026
69308			6268
74933			5233

Table 5.1(b). Contents of *Works* table.

Eid	*Did*	*Months*
62629	153	419
69308	791	
74933	734	356

Table 5.1(c). Contents of *Dept* table.

Did	*Dname*	*Budget*
153		
185		424767
734		210000
791		431581

In this chapter, we extended the formula generation method of Chapter 4 and presented different procedures to obtain logical formulas for queries using ANY, ALL, [NOT] IN, and [NOT] EXISTS keywords and we proposed a method for generating database instances for a test suite by using the method described in Section 4.5. In the next chapter, we will discuss about relaxing some assumptions we made in these thesis, integrating our method with the method presented in [EMM07] to obtain test cases, and will give a summary of contributions and directions for future research.

Chapter Six

Conclusions

Since the majority of the applications in industry interact with a database, testing database applications is of crucial importance for the enterprises. For a long time, lots of effort has been devoted to relational database management systems to make them reliable and correct, however, testing database applications is only recently receiving its due attention in the literature. Automating the white-box testing of DBAs face the same challenges as white-box testing of traditional applications, as well as other challenges caused by the use of databases and embedded SQL statements in the host language.

In this thesis, we addressed the problem of database instance generation and proposed a method for automatic generation of database instances given a test suite. First, we explained our query-aware method for generating a consistent database instance given a test case consisting of simple queries. We explained the notion of conflict between queries, and presented different types of conflicts that can occur, and the methods to detect and resolve them. We also incorporated all the integrity constraints that are used in commercial DBMSs to our method to generate a consistent database instance. Then, we expanded our method to handle more complex queries, i.e. nested queries using [NOT] IN, [NOT] EXISTS, ANY and ALL clauses. We also extended our method to generate multiple database instances to run a test suite containing multiple test cases and illustrated the method with examples.

6.1 Final Remarks

Our method is based on the assumptions stated in chapters 4 and 5. A fundamental limitation of the applicability of our method stems from the assumptions 4.3 and 4.5 which relate to instantiation of input host variables in a given test case by the tester.

These assumptions can be relaxed by integrating our database instance generation method to the approach given in [EMM07] as follows.

Given a DBA, Emmi et. al. describe an algorithm and a tool that automatically instantiate program variables and generate a database instance such that each branch of the application is covered. Although only branch coverage criterion is taken into account in this paper, the algorithm can be adapted to other coverage criteria [GOD05]. The algorithm is based on *concolic execution* [GOD05] [SEN05] which runs the application with concrete and symbolic inputs simultaneously, and extended to handle the interactions with the database. In *symbolic execution*, i.e. running the DBA with symbolic inputs, symbolic constraints are generated along a path that allows the algorithm to derive new values for program inputs and tuples that can result the execution to follow an uncovered path. Simultaneously, *concrete execution*, i.e. running the DBA with concrete inputs, helps to retain precision in the symbolic computations by allowing dynamic values to be used in symbolic execution. Before discussing the algorithm in more detail, we list the following contributions of [EMM07] that are related to our work:

- Test input generation algorithm that extends concolic testing for DBAs
- Constraint solver that is capable of solving symbolic constraints consisting of linear arithmetic constraints as well as string constraints (only sting equality, inequality and membership in regular languages)

Also, the methods in the literature presented in Chapter 3, as well as the method we present in Chapters 4 and 5 assume that queries in the DBA can be statically obtained. However, queries can be constructed dynamically in DBAs and it might not be possible to extract the queries precisely by statically looking at the program or by symbolic execution. Since the method proposed in [EMM07] has a simultaneous concrete execution, exact string of the queries can be obtained.

Briefly, the algorithm in [EMM07] first executes the DBA with random inputs. While executing the program, the algorithm simultaneously constructs *path constraints* and a *database constraint.* A path constraint is a constraint on program variables that must hold in order to execute the path, and a database constraint consists of metadata and actual SQL query executed. After each execution, algorithm looks for touched but uncovered branches in the program and negates path constraints and/or database constraint. Satisfying assignment to the resulting constraints are used as an input for the next execution to cover uncovered branches. The algorithm repeats these steps until full branch coverage goal is met. Termination of the algorithm results in a set of values for input host variables and a set of tuples for each test case generated.

The method described in [EMM07] is complementary to our work in terms of instantiating host variables and obtaining a sequence of queries, because database instance generation capabilities of [EMM07] is limited compared to the method presented in this thesis. Below are the limitations of the database instance generation method presented in [EMM07]:

- Constraint solving approach to generate tuples is used in this method similar to our work and [ZHA01]. However, queries are simple, and are restricted to use one relation in them.
- Multiple queries and their interactions are not considered
- Integrity constraints are not considered

Note that generated database instance without considering interactions and integrity constraints may not be consistent, and targeted branch coverage may not be achieved if the branching depends on the result of a query. Therefore, we propose our database instance generation method to be integrated to test case generation method in [EMM07] as follows.

- Note that database constraints in [EMM07] correspond to logical formulas before considering integrity constraints and conflicts in our work. Therefore, instead of directly generating tuples from these formulas, after the concrete execution, we extract SQL queries with input host variables encountered in this execution.
- In the path constraint, if there is any predicate indicating that the result set of a query q_i should be empty, we remove this predicate from the path constraint, and assign $\nexists$ to p_i. Other queries will have $\exists$ property.
- Extracted SQL queries along with the properties form the sequence of queries Q in our method. We generate the database instance as discussed in previous chapters. The generated database instance is the input used for the next execution corresponding to the path constraint generated.

A drawback of this is that we cannot generate database instances to run a test suite as we described in Section 5.4. Instead, one database instance should be generated for each test case in the test suite, which increases the number of resets to the database.

As for relaxing the assumptions 4.1 and 4.2, we propose the following. Assumption 4.1 states that only attributes with numeric domains can be used in the queries. Authors of [EMM07] describe a satisfiability procedure for strings, which allows the constraint solver tool to handle string equality, inequality and membership in regular languages. It is assumed that arithmetic and string constraints do not interact, therefore, solution is not guaranteed but adequate for a number of SQL queries using attributes with string domain. To fully integrate these types of queries to our method, we need a constraint solver that can handle string equality, inequality and membership in regular languages as well as other comparisons that can be done with strings, such as comparisons determining alphabetical order.

Assumption 4.2 states that every numeric attribute should have a domain constraint

associated with it. For the attributes without a domain constraint in S, we generate domain constraints with the minimum and maximum values specified by the data type of the attribute. For string values, minimum and maximum values depend on the data type, as well as the specified size of the attribute and character set used by DBMS. This procedure can be automated easily.

6.2 Summary of Contributions

Currently, there are not many query-aware database instance generation methods in the literature, and the existing methods have restrictive assumptions in terms of the complexity of queries, integrity constraints and interaction between queries. In our method, we fully automate the conflict detection and resolution, and generation of a consistent query-aware database instance with respect to integrity constraints in a schema. of a given DBA. To illustrate the importance of automating this process, consider the following example:

The DBA under test has n procedures, and for ease of calculation, suppose m test cases are obtained from each procedure, and each test case has k queries in it. If done manually, k (k -1) conflict checks should be done by the tester for each test case. Therefore, the total number of conflict checks that the tester has to do is roughly $nmk(k\text{-}1)$. Moreover, tester should assign values for each attribute used in queries, considering the interactions and integrity constraints. Readers should realize that even for small numbers of n, m and k, this process is very tedious and error-prone to be done manually. The complexity of manual checking is exacerbated if the queries are nested. Our approach automates this process.

6.3 Directions for Future Research

SQL is a broad and powerful query language, and it is impossible to cover all the aspects

of it in this work. Although we showed how to handle more complex queries compared to related work, queries used in enterprise level applications are likely to be more complex than what we have considered. These complexities include queries containing GROUP BY and HAVING clauses, usage of SQL functions in queries, nesting other than the WHERE clause and so on. Logical formula generation method must be extended for more complex queries. In addition, one can develop a more powerful constraint solver tool which can handle Boolean combinations of arithmetic and string constraints, as well as SQL functions and regular expressions.

In our method, we generate tuples for each implicant obtained from the logical formula of a query. To minimize the number of tuples in the generated database instance, it is possible to improve the method such that a tuple will not be generated for every implicant obtained. Also, a different approach should be taken for input host variable instantiation to generate database instances to run a test suite, which is explained in Section 5.4, so that number of resets to the database is minimized.

References

[BIN07] Binnig, C. Kossmann, D. and Lo, E. Reverse Query Processing. In *ICDE*, pages 506–515, 2007.

[BIN08] Binnig, C., Kossmann, D., and Lo, E. 2008. Multi-RQP: generating test databases for the functional testing of OLTP applications. In *Proceedings of the 1st international Workshop on Testing Database Systems* (Vancouver, British Columbia, Canada, June 13 - 13, 2008). DBTest '08. ACM, New York, NY, 1-6

[BRU05] Bruno, N. and Chaudhuri, S. Flexible database generators. In *VLDB*, pages 1097–1107, 2005.

[CHA99] Chan, M. Y., Cheung, S. C. Applying White Box Testing to Database Applications, *Technical Report HKUST-CS-9901*, (January 1999)

[CHA02] Chays D., Deng Y., Frankl P. G., Dan S., Vokolos F.I., Weyuker E. J. Agenda: A test generator for database applications. *Technical Report TR-CIS-2002-04,* Department of Computer Science, Polytechnic University, Brooklyn, New York, 2002.

[CHA03] Chays, D. Test data generation for relational database applications. *PhD Thesis*, Department of Computer Science, Polytechnic University, Brooklyn, New York, 2003.

[CHA04] Chays, D., Deng, Y., Frankl, P. G., Dan, S., Vokolos, F. I., and Weyuker, E. J. An AGENDA for Testing Relational Database Applications. *Journal of Software Testing, Verification and Reliability*, 14, 17-44, (January 2004).

[CHA08] Chays, D., Shahid, J., and Frankl, P. G. 2008. Query-based test generation for

database applications. In *Proceedings of the 1st international Workshop on Testing Database Systems* (Vancouver, British Columbia, Canada, June 13 - 13, 2008). DBTest '08. ACM, New York, NY, 1-6.

[CHE99] Chan, M. Y., Cheung, S. C. Testing Database Applications with SQL Semantics. *Proceedings of the 2nd International Symposium on Cooperative Database Systems for Advanced Applications*, March 1999, 363-374.

[CHR06] Christensen, C. A., Gundersborg, S., de Linde, K., and Torp, K. 2006. A Unit-Test Framework for Database Applications. In Proceedings of the 10th international Database Engineering and Applications Symposium (December 11 - 14, 2006). IDEAS. IEEE Computer Society, Washington, DC, 11-20.

[CLA82] Clarke, L. A. Hassell, J. and Richardson, D. J. "A close look at domain testing.". *IEEE Trans. Software Eng.* vol. SE-8, no. 4. pp. 380-390. July 1982.

[DAT97] Date, C. J., Darwen, H. A. *A Guide to the SQL Standard.* Addison-Wesley: New York, (1997).

[DEN05] Deng, Y., Frankl, P., and Chays, D. 2005. Testing database transactions with AGENDA. In Proceedings of the 27th international Conference on Software Engineering (St. Louis, MO, USA, May 15 - 21, 2005). ICSE '05. ACM, New York, NY, 78-87.

[DTM09] DTM Data Generator. http://www.sqledit.com/dg/

[EMM07] Emmi, M., Majumdar, R., and Sen, K. 2007. Dynamic test input generation for database applications. In Proceedings of the 2007 international Symposium on Software Testing and Analysis (London, United Kingdom, July 09 - 12, 2007). ISSTA '07. ACM, New York, NY, 151-162.

[FRA07] Franzle, M. Herde, C. Teige, T. Efficient Solving of Large Non-linear Arithmetic Constraint Systems with Complex Boolean Structure. Journal on Satis_ability, Boolean Modeling and Computation 1 (2007) 209-236.

[GOD05] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In PLDI, 2005.

[HAF05] Haftmann, F., Kossmann, D., and Lo, E. 2005. Parallel execution of test runs for database application systems. In *Proceedings of the 31st international Conference on Very Large Data Bases* (Trondheim, Norway, August 30 - September 02, 2005). Very Large Data Bases. VLDB Endowment, 589-600.

[HOU06] Houkjær, K. Torp, K. and Wind, R. Simple and realistic data generation. In *VLDB*, pages 1243–1246, 2006.

[IBM09] IBM DB2 Test Database Generator. http://www-306.ibm.com/software/data/db2imstools/db2tools/db2tdbg/

[JEA97] Jeavons, P. G. Cohen D. A., Gyssens M. Closure Properties of Constraints, *Journal of the ACM,* 44(4): 527-548, July 1997

[KAP03] Kapfhammer, G. M. and Soffa, M. L. 2003. A family of test adequacy criteria for database-driven applications. SIGSOFT Softw. Eng. Notes 28, 5 (Sep. 2003), 98-107.

[KOS05] F. Haftmann, D. Kossmann, and A. Kreutz. Efficient regression tests for database applications. In *Conference on Innovative Data Systems Research (CIDR)*, pages 95–106, 2005.

[OST88] Ostrand, T. J. and Balcer, M. J. 1988. The category-partition method for specifying and generating fuctional tests. *Commun. ACM* 31, 6 (Jun. 1988),

676-686.

[RAM03] Ramakrishnan, R., Gehrke, J. *Database Management Systems 3rd Ed.* McGraw-Hill Professional. ISBN 0072465638. (2003).

[SEN05] Sen, K. Marinov, D. and Agha, G. CUTE: a concolic unit testing engine for C. In ESEC/SIGSOFT FSE, 2005.

[SQL92] http://www.contrib.andrew.cmu.edu/~shadow/sql/sql1992.txt

[SQL99] http://www.ncb.ernet.in/education/modules/dbms/SQL99/

[STE04] Stephens, J. M. and Poess, M. Mudd: a multi-dimensional data generator. In *WOSP*, pages 104–109, 2004.

[SUA04] Suárez-Cabal, M. J. and Tuya, J. 2004. Using an SQL coverage measurement for testing database applications. SIGSOFT Softw. Eng. Notes 29, 6 (Nov. 2004), 253-262.

[TSA90] Tsai, W.T., Volovik, D., Keefe, T.F. Automated test case generation for programs specified by relational algebra queries. *Software Engineering, IEEE Transactions* on Volume 16, Issue 3, March 1990 Page(s):316 – 324

[TSA93] Tsang, E. *Foundations of Constraint Satisfaction,* Academic Press, London, 1993. Chapters 1, 2,6, 10, pages 1-52,1577188,299.

[WHI78] White, L. J.. Cohen, E. I. and Chandrasekaran, B. "A domain strategy for computer program testing.'' Cornput. Inform. Sci. Res. Center. Ohio State Univ.. Columbus. Tech. Rep. OSU-CISRC-TR-78-4. 1978

[WHI80] White, L. J.. Cohen, E. I. "A domain strategy for computer program testing." *IEEE Trans. Software Eng.,* vol. SE-6, pp. 247-257.May 1980

[ZHA01] Zhang, J., Xu, C., and Cheung, S. C. 2001. Automatic Generation of Database

Instances for White-box Testing. In *Proceedings of the 25th international Computer Software and Applications Conference on invigorating Software Development* (October 08 - 12, 2001). COMPSAC. IEEE Computer Society, Washington, DC, 161-165.