PARTIAL INFORMATION BASED INTEGRITY CONSTRAINT CHECKING

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

By
Ashish Gupta
December 1994

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Jeffrey D. Ullman
(Advisor)

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Jennifer Widom
(Advisor)

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Hector Garcia-Molina

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Inderpal Singh Mumick

Approved for the University Committee on Graduate Studies:

Abstract

Integrity constraints are useful for specifying consistent states of a database, especially in distributed database systems where data may be under the control of multiple database managers. Constraints need to be checked when the underlying database is updated. Integrity constraint checking in a distributed environment may involve a distributed transaction and the expenses associated with it: two phase commit protocols, distributed concurrency control, network communication costs, and multiple interface layers if the databases are heterogeneous. The information used for constraint checking may include the contents of base relations, constraint specifications, updates to the databases, schema restrictions, stored aggregates *etc.* We propose using only a subset of the information potentially available for constraint checking. Thus, only data that is local to a site may be used for constraint checking thus avoiding distributed transactions. The approach is useful also in centralized systems because relatively inexpensively accessible subsets may be used for constraint checking. We discuss constraint checking for the following three subsets of the afore mentioned information.

- *Constraint Subsumption*: How to check one constraint C using a set of other constraint specifications $\mathbf{C}$ and no data, and the knowledge that the constraints in set $\mathbf{C}$ hold in the database?

- *Irrelevant Updates.* How to check a constraint C using the database update, a set of other constraints $\mathbf{C}$, and the knowledge that the constraints $\{C\} \cup \mathbf{C}$ all hold before the update?

- *Local Checking.* How to check a constraint C using the database update, the contents of the updated relation, a set of other constraints $\mathbf{C}$, and the knowledge that the constraints $\{C\} \cup \mathbf{C}$ all hold before the update?

Local checking is the main focus and the main contribution of this thesis.

Acknowledgements

A lot of people helped me make this thesis a reality. It is difficult to do justice to the thanks I owe my teachers and friends.

I thank Jeff for his continued support, encouragement, and advice throughout my stay at Stanford. I thank Jennifer who gave me a lot to be grateful for, both while at IBM and even more when at Stanford. Inderpal initiated me in the ritual of doing and enjoying research at strange hours of the night and mornings. I am fortunate to have you all as friends, advisors, inspirations, and more. From you I have learned and gained a lot technically and personally. I also thank Surajit, Craig, Arthur, and Hector for their advice on numerous occasions.

I worked most closely with Sanjai. It was a wonderful experience and we had a lot of fun. His contribution to this thesis is immense. I thank Sanjai for all the discussions, (dis)agreements, encouragement, criticism, and most of all for his friendship.

All the people in the NAIL! group and in the database group made meetings fun (yes, even before the lunches started). My friends in the CS department, students and staff members, turned corridors and doorways into conversation areas, to be remembered fondly on every visit to the department. Outside the department my family of friends and relatives made living in the US and at Stanford a time to enjoy, a time to learn, a time to forge bonds, and a mixture of innumerable experiences to be cherished. I thank all of them who make me wish that now was still a few years ago. I would like especially to thank Anoop, Anurag, Dinesh, Nita, and Ramana.

I dedicate this thesis to four other members of my family for their love and support, my parents Urmila and Dharampaul, my aunt Nirmala, and my sister Shikha.

Contents

List of Figures

Chapter 1

Introduction

Traditionally databases have been used as an application tool that interfaces with a user, normally human. However, databases are becoming increasingly important as a back-end utility to interface with other applications such as analysis packages, inventory systems, CAD systems, to facilitate interoperability, portability, and easy management of the underlying data. Many of these applications involve the use of multiple data sources that are administered and manipulated by several persons. Changes made by one participant can potentially affect the functioning of many other participants because related data may be owned and administered by multiple participants.

Thus an increasingly important functionality that databases need to provide is the ability to ensure that related data satisfies certain consistency requirements. Referential integrity constraints are one instance of such a consistency requirement. Traditionally, the users of the database were responsible for ensuring that they did not make the database inconsistent. Thus, notions of consistency stayed in the minds of system administrators, users, or in the application programs that manipulated the database. In large systems this process is very cumbersome because people do not work with the same application forever, and legacy code is difficult to read. Thus, the consistency conditions need to be specified explicitly.

The need for imposing consistency is addressed by providing a facility to specify consistency requirements as *integrity constraints* on the data [Bla81, BMM92, CG92, NY83]. Integrity constraints specify those configurations of the data that are considered semantically correct.

EXAMPLE 1.0.1 Consider the following schema:

`emp`(E, D, S) % *employee number E in department D has salary S*
`dept`(D, MS) % *some manager in department D has salary MS*

Consider an integrity constraint $I1$ require that every department referenced by a tuple in the `emp` relation exists in the `dept` relation. This kind of constraint is a referential integrity constraint. □

A concept closely related to integrity constraints is that of views. Views provide the abstraction needed to manage complex data. A view contains derived data defined in terms of base data. Views are especially useful in applications that decompose a single logical structure in the original application into multiple tuples in a relational database or multiple objects in an object oriented database. The original structure can be defined as a view on top of the underlying data elements. When an application needs some data structure, that structure is reproduced by issuing to the underlying database a query that combines the relevant tables or objects as per the view definition. The overhead introduced by using and querying a database should not affect the performance of the original application.

Often, fast access to derived data is provided by computing and storing the answer to frequently asked queries, *i.e.* by *materializing frequently used views.* A materialized view refers to a view whose current value is stored in the database. Index structures can be built on the materialized view. Consequently, database accesses to a materialized view may be much faster than accesses that recompute the view. Materialized views are especially useful in distributed databases because often the view may involve base data that physically is not at the querying site. Storing a materialized view therefore avoids accessing remote data.

EXAMPLE 1.0.2 On the schema of Example 1.0.1 let view `bad_dept` be defined as:

```
view bad_dept(D)
  select  distinct D
  from    emp
  where   not exists (select  D
                      from    dept
                      where   emp.D = dept.D)
```

View `bad_dept` contains the names of all departments that have at least one employee but do not have a salaried manager. □

Integrity constraints and views are closely related. For instance, constraint $I1$ is violated if some department occurs in a tuple of relation `emp` but the department does not occur in any tuple of relation `dept`. In other words, $I1$ is violated if the view `bad_dept` is nonempty. Given that only the nonemptiness of view `bad_dept` is of relevance to representing constraint $I1$, we could have replaced `bad_dept` with a 0-ary view that derives true when $I1$ is violated and false otherwise. In general, the *violation conditions* for integrity constraints can be represented using 0-ary views, *i.e.*, views with no attributes. The violation condition for a constraint C is the condition satisfied by a database that is inconsistent with respect to C. If a database satisfies a constraint-violation condition then the constraint is said to be *violated* by the database or *not to hold* in the database. If the database does not satisfy the constraint-violation condition, then the database satisfies the constraint, or the constraint *holds* in the database. Henceforth, we use the term integrity constraint to refer to the violation condition for a constraint.

Both integrity constraints and materialized views have been discussed in many forms in the literature and are well accepted as useful mechanisms for using and monitoring data and its semantics. Thus, the relationship between constraints and views is important because similar issues are of concern in making both materialized views and constraints usable in real systems.

1.1 Notation

Now we introduce some notation that will be used throughout this thesis. We will operate in the framework of relational databases and we will use set semantics for relations, *i.e.*, duplicates are not retained in relations or views. We use the term *EDB* or Extensional Database to refer to the base or non-derived portion of the database. *IDB* or Intensional Database refers to views defined using the base relations. We mostly use *Datalog* [Ull89] notation for expressing views and integrity constraints. For instance, view `bad_dept` from Example 1.0.1 is defined in Datalog as follows:

r_1: `all_dept`(D) :– `dept`(D, MS).
r_2: `bad_dept`(D) :– `emp`(E, D, S) & *not* `all_dept`(D).

Intuitively, the first rule defines an IDB relation (or view) `all_dept` that contains the names of all departments that occur in the EDB relation `dept`. The second rule r_2 defines view `bad_dept` as containing the names of all those departments that occur in the `emp` relation but do not occur in relation `all_dept`. More formally, each rule is an if-then rule whose *body* appears on the right hand side of :– and consists of *subgoals* that are separated by the symbol " & " that represents logical AND. The *head* of the rule appears on the left hand of :– . Logical negation is represented by *not*.

Words beginning with upper case letters are used to represent variables, and words beginning with lower case letters represent constants, predicate names, and function symbols. We use a *predicate* of the same name as a relation such that the predicate maps a tuple of the relation to true if the tuple is in the relation, and false otherwise. For instance, `dept` and `all_dept` are predicates in rule r_1. Thus, associated with each predicate is a relation that represents all the data for which the predicate is true. We denote relations by uppercase letters, *i.e.*, for predicate p we use the capital letter P to denote the corresponding relation. A subgoal $p(X, Y)$ is said to be true for a given assignment of constants to variables (X, Y) if the "instantiated subgoal" $p(a, b)$ is true, that is tuple (a, b) is in relation P. $P = \{ab, mn\}$ means that the tuples $p(a, b)$ and $p(m, n)$ are in relation P. We consider only *safe* Datalog rules, *i.e.*, rules where every variable in the head also occurs in some positive subgoal in the body. In addition, every variable in a negative subgoal, for instance variable D in subgoal "*not* `all_dept`(D)" also appears in some positive subgoal in the same rule.

If a particular assignment of constants to the variables in rule r makes the body true, then the rule *derives* the instantiated head. For instance, if `dept`$(toy, 911)$ is true, then `all_dept`(toy) is derived by rule r_1.

The view defined by rules r_1 and r_2 can be written in SQL as in Example 1.0.1. An SQL query is a Select-From-Where statement in which the "**select**" clause specifies the attributes that are part of the view, the "**from**" clause specifies the relations that are used in the view definition, and the "**where**" clause specifies the selection conditions used to define the view. In this thesis we use SQL queries with arithmetic inequalities, nested subqueries, and negated subqueries. We do not use aggregation constructs.

An *integrity constraint* can be defined using a SQL or Datalog program that defines a 0-ary view we call `panic`. The interpretation is that `panic` is satisfied if and only if the constraint is violated. For instance, constraint $I1$ can be written in Datalog by rewriting rule r_2 as:

r_3: $\texttt{panic} :- \texttt{emp}(E, D, S)\ \&\ \textit{not}\ \texttt{all\_dept}(D).$

1.2 Problem Statement

A database that is initially consistent with respect to a set of integrity constraints can become inconsistent if the database is updated. Similarly, materialized views can become incorrect if the underlying database is updated. Hence, constraints need to be checked and materialized views need to be updated in response to updates to the database. The brute-force method for updating views and checking constraints, in response to database updates, is first to perform the update and then reevaluate the constraint or recompute the view on the new database. Often *incremental maintenance* and *incremental checking* are used to avoid completely recomputing the view or reevaluating the constraint. Incremental evaluation relies on using the change to the database to limit the recomputation process to only that part of the database that possibly interacts with the changes. We refer to incremental evaluation as a brute-force method also because by default they use all the underlying relations. When a constraint or view involves data from multiple sites, view maintenance and constraint checking could involve a distributed transaction and the expenses associated with it: two phase commit protocols, distributed concurrency control, network communication costs, and multiple interface layers if the databases are heterogeneous. Even in a centralized system, accessing more information involves more time and computation than accessing less information.

Thus, the question of interest to us is how to efficiently maintain materialized views and how to efficiently check constraints.

1.2.1 Our Approach – Use Partial Information

In this thesis we propose using *partial information* as an alternative to the use of brute force techniques for view maintenance and constraint checking. We focus primarily on integrity constraints while keeping in mind that constraints are a special kind of views. Thus, some of the results developed for constraint checking extend to view maintenance. We explain the generalizations at the end of the thesis. The following examples illustrate what we mean by partial information. The examples consider only integrity constraints.

EXAMPLE 1.2.1 Recall constraint $I1$ that requires that every department referenced by a tuple in the `emp` relation exists in the `dept` relation. Consider another constraint $I2$ that requires that if the department referenced by a tuple t in `emp` does not exist in the `dept` relation, then the salary field of t should be 0. $I2$ is written in Datalog as:

$I2$: $\texttt{panic} :- \texttt{emp}(E, D, S)\ \&\ not\ \texttt{all\_dept}(D)\ \&\ S \neq 0.$

We know that if constraint $I1$ holds, then the department attribute of all `emp` tuples exists in relation `dept`. Thus, if constraint $I1$ holds then constraint $I2$ holds also. Thus, if we check constraint $I1$ successfully we need not check constraint $I2$.

Constraint definition $I1$ was used to infer that constraint $I2$ also holds. No base relation was used. □

EXAMPLE 1.2.2 Consider a constraint requiring that all employees in relation `emp` have salaries at most 100000.

$\texttt{panic} :- \texttt{emp}(E, D, S)\ \&\ S > 100000.$

Let tuple $(b, toy, 100)$ be inserted into relation `emp`. Given that $100 \not> 10000$, we can infer that the insertion does not violate the constraint if the constraint was not violated before.

The contents of the base relations `emp` were not used to check the constraint. Only the constraint specification and the update were used, along with the information that $I1$ was not violated before the insertion. □

EXAMPLE 1.2.3 Recall constraint $I1$ defined in Example 1.0.1. Let the constraint hold in database $DB2$ where relation `emp` in database $DB2$ has only one tuple $(mary, toy, 200)$. Let tuple $(john, toy, 100)$ be inserted into relation `emp`. Given that $DB2$ satisfies $I1$, we can infer that department toy has a salaried manager. Therefore, the inserted tuple cannot possibly violate the constraint if the constraint was not violated before.

The contents of the base relation `dept` were not used to check constraint $I1$. Only the old contents of relation `emp` were used, along with the information that $I1$ was not violated before the insertion. □

EXAMPLE 1.2.4 Consider two constraints $I3$ and $I4$ defined on the employee-department database of Example 1.0.1 respectively by rules r_4 and r_5.

r_4: $\texttt{panic} :- \texttt{emp}(E, D, S)\ \&\ D = toy\ \&\ S > 900.$

r_5: $\texttt{panic} :- \texttt{emp}(E, D, S)\ \&\ \texttt{dept}(D, MS)\ \&\ S > MS.$

$I3$ is violated if any employee of the '*toy*' department earns more than 900. $I4$ is violated if any employee (in any department) earns more than any manager of the same department. Consider a database $DB3$ in which both constraints $I3$ and $I4$ hold.

Let tuple $\texttt{dept}(toy, 911)$ be inserted into database $DB3$. This insertion does not affect constraint $I3$ but could potentially violate constraint $I4$ if some employee of the *toy* department has salary > 911. However, database $DB3$ satisfied both the constraints before the insertion. Therefore, all employees of the *toy* department have salaries at most 900, and thus no employee has salary > 911. Therefore, constraint $I4$ is not violated by the new tuple $\texttt{dept}(toy, 911)$.

Constraint definitions $I3$ and $I4$ were used to infer that the update does not violate $I4$ given that neither constraint was violated before the insertion. No base relation was used. □

The above examples illustrate constraint checking using various kinds of information. In general, the available information can include the constraint specification, the update to the database, the updated base relations, the unchanged base relations, and other constraints. As illustrated in the examples, often a subset of this information, *i.e.* partial information, is sufficient to check a constraint.

Why are partial-information-based constraint checking methods interesting? These methods provide alternatives to the brute-force method and often can be more efficient. For instance, the partial information used may be a subset of the relations used by the brute-force method, whereupon fewer relations need to be locked and fewer disk accesses need to be made to read the relations. In addition, these partial-information based methods can be used in scenarios where the brute-force method cannot be used. For instance, the partial information used by our methods could be entirely on the same site in a distributed database, and thereby all remote accesses are avoided. In situations like disconnected networks, high-security databases, mobile computers, *etc.*, it may not be possible to access the other sites in the distributed system. In such situations, partial-information-based methods are invaluable. Recall however, the idea of using partial information is not restricted to distributed databases. The cost of accessing different relations on the same site may be different because, for example, some critical data may be expensive to lock, or some data may be periodically unavailable. Checking constraints using only the cheaper data then becomes attractive.

Partial information could also include information about functional dependencies, schema restrictions, stored aggregates *etc.*, in addition to the contents of base relations, constraints specifications, and updates. However, we do not consider such information in this thesis.

Which Subsets do we Consider?

In this thesis we consider three different subsets of the set of possibly available information for checking constraints. We refer to each of these subsets as an *instance of partial information.* In particular, we consider:

- *Constraint Subsumption*, *i.e.*, how to check one constraint C using a set of other constraint specifications $\mathbf{C}$ and no data, and the knowledge that the constraints in set $\mathbf{C}$ hold in the database. Constraint subsumption was illustrated in Example 1.2.1.
- *Irrelevant Updates*, *i.e.*, how to check a constraint C using the database update, a set of other constraints $\mathbf{C}$, and the knowledge that the constraints $\{C\} \cup \mathbf{C}$ all hold before the update. An irrelevant update was illustrated in Example 1.2.2.
- *Local Checking*, *i.e.*, how to check a constraint C using the database update, the contents of the updated relation, a set of other constraints $\mathbf{C}$, and the knowledge that the constraints $\{C\} \cup \mathbf{C}$ all hold before the update. Local checking was illustrated in Example 1.2.3.

Note, all the methods we consider use the *Initial Consistency Assumption.* This assumption states that given a set of constraints $\mathbf{C}$ and an update made to the database DB, all constraints in $\mathbf{C}$ hold in the database DB before the update is made.

Local checking is the main contribution of this thesis and the instance we study in most detail. Because local checking uses one base relation it is more powerful than subsumption and irrelevant update detection. At the same time because only the updated relation is used, constraint checking can be done without accessing any other relations. Subsumption and irrelevant updates have previously been studied in [BC79, BCL89, Elk90, LS93, ZO93]. We put the results of these papers in the perspective of constraints. For the problem of irrelevant updates, we explore some new aspects that have not been considered before.

Now we discuss some properties of partial-information-based constraint checking methods that make them attractive and viable alternatives to brute-force techniques. These properties are of interest in the case of most partial-information-based methods but we have explored them primarily in the context of local checking.

1.2.2 Properties of Partial-Information-Based Methods

One property of interest is how "well" the available partial information is used. The following example illustrates this idea.

EXAMPLE 1.2.5 Consider a database of intervals of a line and points on the line represented by the following relations:

$\texttt{line}(B, E)$ % *The database has an interval from point B to point E*
$\texttt{point}(P)$ % *Point P is in the database.*

Consider constraint $I5$ on the above relations that requires that no point in relation `point` should lie on any interval in relation `line`.

$I5$: `panic` :- $l(B, E)$ & $r(P)$ & $B \leq P \leq E$.

If a database satisfies constraint $I5$ then all points in relation `point` lie outside every interval in relation `line`.

We want to check constraint $I5$ using local checking, *i.e.* using only the contents of relation `line` when the update to the database consists of inserting tuple (b, e) into `line`. Intuitively, `line`(b, e) does not violate $I5$ if the interval $[t.B, t.E]$ for some existing `line` tuple t contains the interval $[b, e]$. The intuition is that if tuple t does not violate $I5$ then every point in relation `point` lies outside the interval $[t.B, t.E]$ and outside every subset of this interval, in particular the interval $[b, e]$. Tuple t is said to *cover* tuple `line`(b, e).

However, the above method does not completely use the information available in relation `line`. Why? An inserted interval could be contained in the *union* of two or more existing intervals but not be contained in either of these intervals. Thus, a method that uses several existing tuples to cover the inserted tuple is more powerful than a method that considers only one tuple. □

We formalize how well a method uses the available information by defining the notion of *completeness* in Chapter 2. Informally, a complete method is the "most powerful" method given the available information. That is, for a specified amount of information, if any method can check a constraint with that information then a complete method also checks the constraint. For constraint $I5$, it can be proved that if all the tuples in relation `line` are used to cover the inserted tuple, then the resulting method is complete.

Another desirable property of partial-information-based methods is that they should be *query based*. For instance, in Example 1.2.5 a single cover tuple for the inserted tuple `line`(b, e) can be found by the following Datalog query.

`cover`(B, E) :- `line`(B, E) & $B \leq b$ & $E \geq e$.

The above query can be executed using an existing database engine. Partial-information-based techniques that are not query based may have to execute outside the database engine and may have to either duplicate or forgo the use of indices, query optimizers, efficient I/O strategies, *etc.* that are otherwise provided by the database.

The above example would indicate that the queries that implement the partial information method depend on the inserted, deleted, or updated tuple. However, all the local checking algorithms discussed in this thesis derive the queries using parameters instead of the actual update tuple and generate the query once, at compile time. At run time the parameters are instantiated with the update before evaluating the query. For instance, the datalog query stated above uses the parameters b and e from the inserted tuple `line`(b, e). The query is obtained without using the actual value of the parameters b, e. The values of the parameters are obtained at run time. Thus another property of interest is to be able to derive methods using parameters and not have to use the actual update.

Tradeoffs

Both the existence of a partial-information-based method for checking a constraint, and the properties of the method, depend on how complicated the constraint is and on how much resources we are willing to spend on the method. For instance, consider constraint $I5$ from Example 1.2.5. If the local checking method looks at only one tuple in the relation `line` then the method is not complete. However, if the method looks at the entire relation, then we do get a complete method. The first method can be expressed in a nonrecursive, first-order language. However, the second method needs a first-order + fixed-point language because the method needs to look at an arbitrary number of tuples in relation `line`. On the other hand, for the referential integrity constraint $I1$, it is possible to express a complete method in a first-order language.

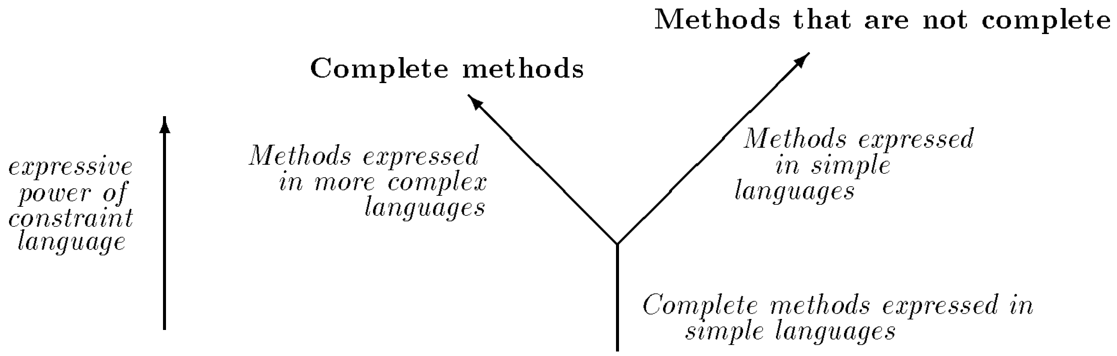

Figure 1.1: Possible Paths to Explore for Local Checking

There is a tradeoff between the properties of the partial-information-based methods, the expressive power of the constraint language, and the expressive power of the language used to encode the constraint checking method. Thus, partial-information-based methods can be developed with an intent to do better in terms of one metric while doing less well on another. We study these tradeoffs for local checking methods.

The strategy for studying the tradeoffs is pictorially encoded in Figure 1.1. We pick increasingly complex constraint specification languages and try to derive local checking methods for these constraint classes. Initially, when we are the stem of the "Y" in Figure 1.1, the methods obtained are complete and expressible in fairly simple languages. When the constraints get more powerful, we need to compromise either the completeness of the resulting methods or we need to resort to using more powerful languages to express the methods. These two options respectively correspond to the right and the left branches of the "Y." For constraints that use a very complex language, it may not be possible to derive complete methods whereas it may still be possible to derive methods that are not complete. Hence, the right branch of the "Y" goes farther. Later in this thesis we instantiate the "Y" with specific constraint languages, methods and the language needed to express them, and their properties in terms of completeness and complexity.

1.2.3 Partial Information Yields Partial Answers

One important point to note about partial-information-based methods is that they may not be able always to check successfully a constraint. For instance, consider the constraint $I2$ from Example 1.2.1 that is violated if an employee in `emp` whose department does not exist in `dept` has salary $\neq 0$. Let a tuple $\texttt{emp}(john, toy, 0)$ be inserted into relation `emp`. It is straightforward to check that this tuple cannot violate $I2$ because its salary field is $= 0$. However, if the inserted tuple was $\texttt{emp}(john, toy, 10)$ then a simple inference is not possible. Now, if there was another tuple $(mary, toy, 50)$ in `emp`, we could use the reasoning of Example 1.2.3 to infer that the department *toy* does indeed exist in `dept` and that $I2$ continues to hold after the insertion. The above line of argument can be extended to illustrate that in some cases both relations `emp` and `dept` may be required to compute the effect of the insertion. For instance, if $(mary, toy, 50)$ were not in relation `emp` then both relations would need to be accessed.

Thus, partial-information-based methods should be treated as the first line of defense for checking constraints but not the last. They are possibly more efficient than methods that use all the available information, but are not a replacement for such methods. Constraint checking strategies that use all the relations involved in a constraint, and are thus not partial-information based, are discussed in [SSMJ90, BMM92]. These strategies optimize constraint checking using incremental checking, finite differencing, storing aggregate information, *etc.* We do not consider such methods in this thesis.

1.3 View Maintenance

Now we briefly consider the use of partial-information-based methods in maintaining materialized views. Note, in the case of views the set of available information can also include the old contents of the materialized view, in addition to the information used for constraint checking. The following example illustrates the use of the old contents of the materialized view:

EXAMPLE 1.3.1 Consider the view `bad_dept` defined in Example 1.0.1. Let the view contain tuples (toy), $(sales)$ for a given database $DB1$, *i.e.*, departments *toy* and *sales* have at least one employee but no salaried manager. Let tuple $(john, toy, 100)$ be inserted into relation `emp`. This new `emp` tuple could contribute the tuple (toy) to view `bad_dept`. However, (toy) is already in the view and therefore we can ignore the possible effect of the insertion on the view because we do not retain duplicates in the view `bad_dept`.

The contents of the base relations `emp` and `dept` were not used to discover that view `bad_dept` did not need to be updated. Only the old contents of view `bad_dept` were used. □

Recall, constraints are a special kind of views, namely 0-ary views. If a constraint holds in a database, then the corresponding 0-ary view is empty; if the constraint does not hold in the database then the view is nonempty. Using the initial consistency assumption for constraint checking is equivalent to saying that the 0-ary view defined by the constraint is materialized and is known to be empty. Thus, for checking constraints, we always essentially use the contents of the materialized view which is always empty.

The primary problem in using partial information for view maintenance arises from the difference between *negative* and *positive* inferences that can be made using partial information. A negative inference means that an update does not contribute to a view and thus the inference process need not go further. A positive inference means that an update does contribute to a view and furthermore, the actual contribution can also be determined using only partial information. Often, the actual contribution cannot be determined using only partial information. Thus, intuitively positive inferences are more "difficult" to make than negative inferences. The following example illustrates the complication introduced by positive inferences.

EXAMPLE 1.3.2 Consider a view `high_paid_emp` defined as follows:

$$\texttt{high\_paid\_emp}(E, D, MS) \text{ :- } \texttt{emp}(E, D, S) \ \& \ \texttt{dept}(D, MS) \ \& \ S > MS.$$

That is, the view contains the name and department of an employee, and the salary of a manager in the same department as the employee, if the employee's salary is greater than the salary of the manager.

Let the available partial information consist of the relation `emp` and view `high_paid_emp`. Let `emp` have tuple $(john, toy, 100)$ and let the view have tuple $(john, toy, 50)$. That is, department *toy* has a manager with salary 50. Let tuple $(mary, toy, 200)$ be inserted into relation `emp`. We can infer, using `emp` and view `high_paid_emp`, that tuple $(mary, toy, 50)$ is inserted into the view. However, we cannot be certain that the inserted tuple does not cause any more insertions into the view. For instance, there may be a manager with salary 125 in department *toy* thereby requiring tuple $(mary, toy, 125)$ to be inserted into `high_paid_emp`. □

Thus, partial information based techniques are more useful in detecting irrelevant updates, that is, updates that do not affect a materialized view, than for determining the exact change that needs to be made to the view. Identifying irrelevant updates implies that fewer updates need to be pushed through an incremental view maintenance process thereby resulting in more efficient view maintenance. An incremental view maintenance process computes the effect of the database update, on the materialized view. The goal is to access as little of the underlying relations and the old contents of the materialized view as possible, to compute the *change* to the view. The incremental view maintenance problem is not considered in this thesis.

1.4 Application Scenario

We have explored integrity constraint checking and management in the context of a collaborative design environment in the Civil Engineering domain [HKG+94]. The environment ties together multiple participants like architects, structural engineers, HVAC (heating, ventilation, air conditioning) experts, contractors, *etc.*, in a distributed, heterogeneous, multiple database environment. Together, these participants design and fabricate a facility like a power plant or a multistory building. Some stages of the project involve one participant more than others, but all stages involve multiple participants. For instance, the architect has a larger contribution in the initial stages of building design but the architect continues to participate and revise the design after the structural engineers and contractors come into the picture.

In a collaborative design environment a lot of information sharing is required, and there are also a large number of integrity constraints that need to be enforced across the shared information. For instance, the contractor needs to know of the design decisions made by the structural engineer and also of the design changes that take place subsequently. All the information is needed in a timely fashion in order to avoid working on out of date designs.

The intuition of Section 1.2 that partial information based constraint checking techniques are useful, is ratified in this environment. One characteristic of collaborative design in Civil Engineering is that the participating databases are usually proprietary and thus it is of utmost importance to maintain autonomy while tying together the various participants. The autonomy results in occasional unavailability of some sites, higher expenses in accessing someone else's data, and often complete inability to access remote data because of security considerations.

1.5 Thesis Outline

For developing our results, we focus on integrity constraints and finally show how to extend our results to materialized views. In all we consider three instances of partial information. The first instance, local checking, is studied in most detail in this thesis and is discussed in Chapters 2, 3, 4, and 5. The other two instances, constraint subsumption and irrelevant updates, are studied in Chapter 6. For each instance of partial information, we vary the expressive power of the constraint specification language and develop constraint checking methods. For local checking we follow the strategy outlined in Figure 1.1. We vary the constraint specification language and develop methods, complete and otherwise, using different classes of languages to express the methods.

Chapter 2

In this chapter first we give the intuition for how partial information is used to check integrity constraints. Then we formalize the local checking problem, *i.e.*, checking constraints using the update to the database, the updated relation, and the constraint specification. We illustrate the issues relevant to the problem and lay the groundwork for the results that appear in subsequent chapters.

We consider integrity constraints that can be expressed using conjunctive queries extended with arithmetic inequalities [Klu88] and discuss how conjunctive query containment results (described in Appendix A) are used to develop complete local checking methods. These methods are not query based.

Chapter 3

In this chapter we consider the class of conjunctive query constraints with arithmetic inequalities introduced in Chapter 2 and study restrictions that need to be placed on the constraint language in order to obtain query-based local checking methods while retaining completeness. This chapter studies points on the left branch and stem of the "Y" in Figure 1.1. First, we consider restrictions that result in complete query-based local checking methods that are expressible in a non-first-order language. Subsequently, we consider stronger restrictions on the constraint language that result in the complete local checking method being expressible in a first-order language. We also discuss the complexity of implementing the non-query-based methods derived in Chapter 2 for the unrestricted class of conjunctive queries with arithmetic inequalities.

Chapter 4

This chapter corresponds to traversing the right branch of the "Y" in Figure 1.1. We explore query-based methods that are not complete. We develop such methods for constraints that are expressed in languages that are subsets of first order logic more expressive than conjunctive queries with arithmetic comparison. The language is less expressive than general Datalog with negation and arithmetic comparisons. For such constraints we give algorithms to derive methods that are not complete but are expressible as nonrecursive queries on the accessible relation. The emphasis of this chapter is to make the constraint language increasingly more expressive and still be able to derive local checking methods.

Chapter 5

In this chapter we study some properties of the local checking techniques of Chapters 2, 3 and 4. We extend local checking to consider deletions to relations that appear negatively in constraints, and for modifications to relations that may occur positively or negatively. For instance, deletions from the negatively occurring relation `dept` could violate referential integrity constraint $I1$ in Example 1.0.1. We also study the application of local checking to materialized view maintenance.

Chapter 6

In this chapter we consider the other two instances of "partial information" and describe how to apply existing results from conjunctive query containment and update independence to these instances. In particular, we describe how to check integrity constraints using only other constraint specifications, and how to check constraints using only the constraint specification and update made to the database. Finally, we compare the results of this thesis with other constraint checking techniques described in the literature.

Chapter 7

In this chapter we summarize the contributions of this thesis and briefly describe a prototype distributed constraint management system that we designed and implemented in the context of the "Collaborative Environment for the Design of Buildings" project. The system uses some of the ideas described in this thesis. The details of the design and implementation can be found in [GT93, GT94, TH93, Tiw94]. We also outline some directions for future work, based on extending partial-information-based methods and on extending the constraint management system we have built.

Appendix A

Partial information based methods often result in a question that involves determining if one conjunctive query is contained in another conjunctive query. In this appendix, we give necessary and sufficient conditions for a conjunctive query with arithmetic comparison predicates to be contained in a union of similar conjunctive queries. We also give restricted classes of conjunctive queries for which stronger results can be stated. The results of this appendix are used in Chapter 3.

Appendix B

Partial information based techniques often produce implications of the form $A \Rightarrow B$ where A and B are sentences that use arithmetic comparison operators. In this appendix we discuss algorithms that manipulate the above implication to derive sufficient conditions that are less expensive to evaluate than evaluating the implication. The algorithms developed in this appendix are used in Chapter 4.

Appendix C

Like Appendix B, this appendix also considers how to solve implications of the form $A \Rightarrow B$, but with certain restrictions on A and B. In particular, if A and B are such that their solutions lie in n-dimensional parallelepipeds then the implication problem is the same as determining if a n-dimensional parallelepiped is contained in a union of other n-dimensional parallelepiped. We discuss how to express this problem in Datalog and how to efficiently evaluate the containment question. The results of this appendix are used in Chapter 3

Appendix D

In Chapter 6 we describe how to check constraints using only the update and the constraint specification. The proposed solution reduces the specified constraint into a different constraint that may be in a language that is more expressive than the language of the original constraint. In this appendix we discuss the relationship of the language of the original constraint to the language of the constraint generated by the reduction. The results of this appendix are used in Chapter 6.

Chapter 2

Notation, Intuition, and Local Checking

Of the three instances of partial information considered in this thesis, local checking is studied in most detail. Local checking involves checking a constraint using the constraint specification, the update, the updated relation, and the initial consistency assumption. This chapter introduces local checking after introducing some preliminary concepts and notation.

All the results on constraint checking in this thesis use the following simple but useful assumption:

Definition 2.0.1 (**Initial Consistency Assumption**) Given integrity constraints $\{C_0, C_2, \ldots, C_m\}$ and an update made to the database DB, the initial consistency assumption states that each C_i holds in the database DB before the update is made. $\Box$

Chapter Outline

As illustrated earlier, the expressive power of the constraint language is an important factor in deriving partial-information-based constraint checking methods. Thus, first we illustrate describe possible constraint specification languages in Section 2.1. Subsequently, we give a pictorial way of thinking about checking integrity constraints using partial information.

In Section 2.3 we introduce the main problem considered in this thesis, namely local checking, and for it we give a pictorial intuition. In Section 2.4 we formalize the pictorial understanding developed in the previous section. In Section 2.5 we illustrate – via a set of examples – some of the interesting properties of local checking that affect our study. In Section 2.6 we discuss the alternate paths that can be followed in the effort to explore local checking, and we outline the paths that we take in the remainder of this thesis using the "Y" introduced in Figure 1.1.

In Section 2.7 we define the first class of constraints for which local checking is explored: conjunctive query constraints with arithmetic inequalities. We discuss how to reduce the problem of locally checking a conjunctive query constraint to a question of conjunctive query containment. In this chapter, we use the results of Appendix A to establish the necessary mathematical background that is required to develop local query-based methods for checking conjunctive query constraints. The actual methods are developed in subsequent chapters.

2.1 Constraint Query Languages

A *constraint query* is a Datalog-like logic program with a distinguished 0-ary goal we call `panic`. The interpretation is that `panic` is satisfied if and only if the constraint is violated. Many possible languages can be used to express constraint queries. Of the languages listed below, we consider a select few when developing partial information based constraint checking methods.

1. Conjunctive queries [C88]. A conjunctive query (CQ) is a single Datalog rule that does not use negation or arithmetic in its body. Equivalently, a CQ is a single select-project-join statement where the selections are restricted to equating attributes with constants and joins are restricted to be equijoins. For example, the following constraint specifies that no employee can be in both the "sales" and the "accounting" departments. The schema of relation `emp` is as described in Example 1.0.1.

   $\texttt{panic} :- \texttt{emp}(E, sales, S) \ \& \ \texttt{emp}(E, accounting, S')$.

2. Unions of CQ's [SY80]. This class can be used to express a set of constraints where each constraint itself can be expressed as a single conjunctive query.

3. Conjunctive queries with arithmetic comparisons [Klu88]. For example, the following two constraints together specify that every employee's salary is within a range for their department as specified by relation `sal_range`.

   $\texttt{panic} :- \texttt{emp}(E, D, S) \ \& \ \texttt{sal\_range}(D, Low, High) \ \& \ S < Low$.
   $\texttt{panic} :- \texttt{emp}(E, D, S) \ \& \ \texttt{sal\_range}(D, Low, High) \ \& \ S > High$.

4. Conjunctive queries with negated subgoals [LS93]. For example, the following query requires that every employee in relation `emp` have health insurance. $\texttt{insured}(e)$ says that employee e has health insurance.

   $\texttt{panic} :- \texttt{emp}(E, D, S) \ \& \ not \ \texttt{insured}(E)$.

5. Other combinations of (2) through (4), *i.e.*, CQ's or unions of CQ's, with or without arithmetic comparisons and with or without negated subgoals.

6. Nonrecursive Datalog, *i.e.*, when intensional relations could be used. Also, we could consider nonrecursive Datalog extended with arithmetic and with negation. For example, the following

set of rules specifies a constraint that is violated whenever an employee with salary greater than 10000 is not in any department.

`panic` :- `emp`(E, D, S) & *not* `all_dept`(D) & $S > 10000$.
`all_dept`(D) :- `dept`(D, MS).

7. Recursive Datalog, including cases with or without arithmetic comparisons, with or without negated subgoals, and with or without intensional relations. For instance, the following constraint requires that no employee is their own boss.

   `panic` :- `boss`(E, E).
   `boss`(M, E) :- `emp`(E, D, S) & `manager`(D, M).
   `boss`(B, E) :- `boss`(B, B') & `boss`(B', E).

8. All of the above with arbitrary interpreted functions such as functions that compute volume, distance, *etc.*

We assume that every constraint query includes exactly one rule with `panic` in the head. Any constraint with multiple `panic` rules can equivalently be expressed as multiple constraints, as illustrated in the salary range example in item 3 above. Thus, each constraint can be viewed as a program that defines the view `panic` such that when the constraint is violated, the corresponding program derives `panic`.

2.2 How to Use Partial Information

An integrity constraint restricts the possible legal states of the database to a subset of all the possible states of the database. That is, for every non-trivial integrity constraint, some databases will not be legal. For instance, consider integrity constraint $I1$ from Example 1.0.1. The constraint prohibits all those states of the database where an employee is in a department that does not appear in the `dept` relation. Thus, if we knew that relation `emp` has tuple $(a, toy, 200)$ then we can infer that all legal states of the database, *i.e.*, databases that do not violate constraint $I1$, have at least one tuple of the form `dept`(toy, ms) where ms is some constant. Thus, all databases that are legal with respect to $I1$ and where relation `emp` has tuple $(a, toy, 200)$, satisfy the following existentially quantified statement:

$\exists MS$: `dept`(toy, MS).

When all the relevant base data is available then full information is available about the database. If only partial information is available then even though the contents of the database are not known, often it is possible to infer certain facts about the unknown parts of the database given that the database satisfies some constraints. The above paragraph illustrates such an inference about the unknown relation `dept` that is made using the tuple `emp`$(a, toy, 200)$ and the initial consistency assumption. These facts can be expressed in some logical or geometric language and furthermore, the facts can be used to infer the effect of database updates on constraints.

It is not possible always to represent easily or to infer facts about the unknown parts of the database using some partial information. For instance, consider the constraint that requires the average salary of all employees in the "*toy*" department to be less than the salary of each manager of the "*toy*" department. If the available partial information consists of the initial consistency assumption and the contents of the relation `emp`, then we can conclude that the salary of a manager in department *toy* is at least as much as the minimum salary of any employee in department *toy* and at most as much as the maximum salary of any employee in department *toy*. Such a fact possibly can be used for checking the constraint if a new manager *tom* is introduced into department *toy*. If *tom* has salary more than the maximum salary of an employee in *toy* then the constraint holds. Aggregation capabilities are needed to express or use the inferred knowledge about salary values in relation `dept`. In general, the language used to express constraints impacts the kind of facts that can be asserted about the unknown portion of the database using the available partial information. A more expressive language may allow more powerful assertions to be made but may make the inference process correspondingly more difficult and expensive. These issues come up many times in the subsequent chapters and are addressed for the constraint classes considered by us.

2.2.1 Pictorial Intuition

Now, we introduce a pictorial way of representing available partial information and the corresponding states of the unknown portions of the database. The representation is informal and is useful in explaining the partial-information-based constraint checking results that we develop in this and subsequent chapters. We divide the picture into two parts: the left side contains the available information and the right side contains the unknown parts of the database. For instance, in Figure 2.1 the known information consists of the constraint $I1$ and the relation `emp` that has a single tuple `emp`$(a, toy, 200)$. The rectangle $\mathbf{D}_R$ represents the set of all possible states for the inaccessible part of the database. A particular inaccessible database state corresponds to a point in the box D_R. Within this rectangle we use a circle to specify the *illegal* states of the database with respect to the available information. Thus, in Figure 2.1 the circle S represents all those database states that violate constraint $I1$ given tuple $(a, toy, 200)$ in relation `emp`. For a given constraint and partial information, the initial consistency assumption implies that the actual state of the unavailable database lies outside the set of illegal states.

We refer to the above figure later in this chapter and also in subsequent chapters.

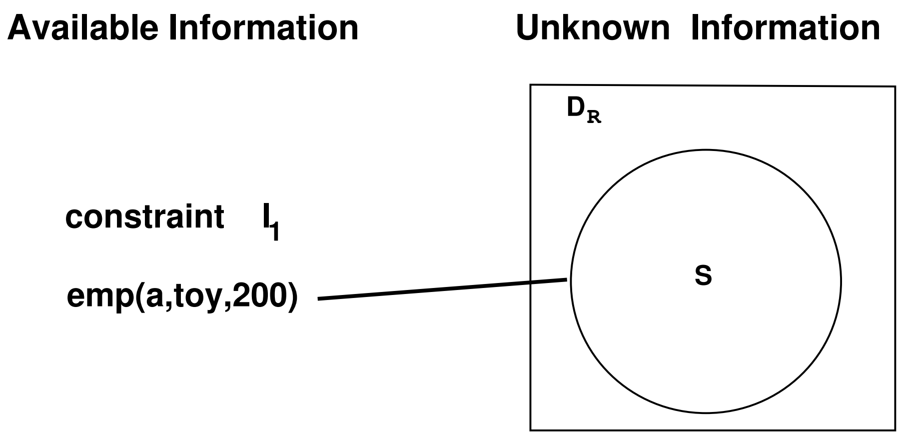

Figure 2.1: Pictorial Representation of Illegal Database States

2.3 What is Local Checking?

Local checking involves checking a constraint using the constraint specification, the update, updated relation, and the initial consistency assumption. Consider the set of constraints $\mathbf{C} = \{C_0, C_1, C_2, \ldots, C_m\}$, and suppose that the predicates in these constraints are $p_0, p_1, p_2, \ldots, p_k$. We assume that the update to the database consists of inserting a single tuple into relation P_0. We want to check the constraints using the old contents of P_0, the tuple inserted into P_0, the set $\mathbf{C}$, and the initial consistency assumption for the set of constraints $\mathbf{C}$. For convenience of presentation, we refer to relation P_0 as L (for the "local" or accessible relation), and we let the other relations be $R_1, R_2, \ldots, R_n$ (for the "remote" or inaccessible relations). We use the symbol $\bar{R}$ to refer to the set of relations $\{R_1, \ldots, R_n\}$. Because the partial information consists of only the local relation, we refer to this instance of partial-information-based constraint checking as local checking. The following example illustrates local checking for referential integrity constraint $I1$.

EXAMPLE 2.3.1 Recall the employee-department database schema from Example 1.0.1 and the referential integrity constraint $I1$ that requires every department referenced by a tuple in the `emp` relation to exist in the `dept` relation.

Let tuple `emp`$(john, toy, 100)$ be inserted into relation `emp`. If `emp` contains a tuple `emp`$(mary, toy, 200)$ and the database satisfies integrity constraint $I1$ before the tuple is inserted, then we can infer that the inserted tuple does not violate constraint $I1$ using the intuition from Example 1.2.3. The argument is that if $I1$ holds before the insertion then relation `dept` has a tuple (toy, ms), for some constant ms, and this tuple ensures that inserted tuple `emp`$(john, toy, 100)$ does not violate constraint $I1$.

Conversely, if a tuple `dept`$(toy, 300)$ is deleted and if relation `dept` has another tuple `dept`(toy, ms), for some constant $ms \neq 300$, then integrity constraint $I1$ can be verified without remotely reading relation `emp`. □

We often refer to the local relation as the accessible relation and we refer to the remote relations as inaccessible.

2.3.1 Pictorial Intuition for Local Checking

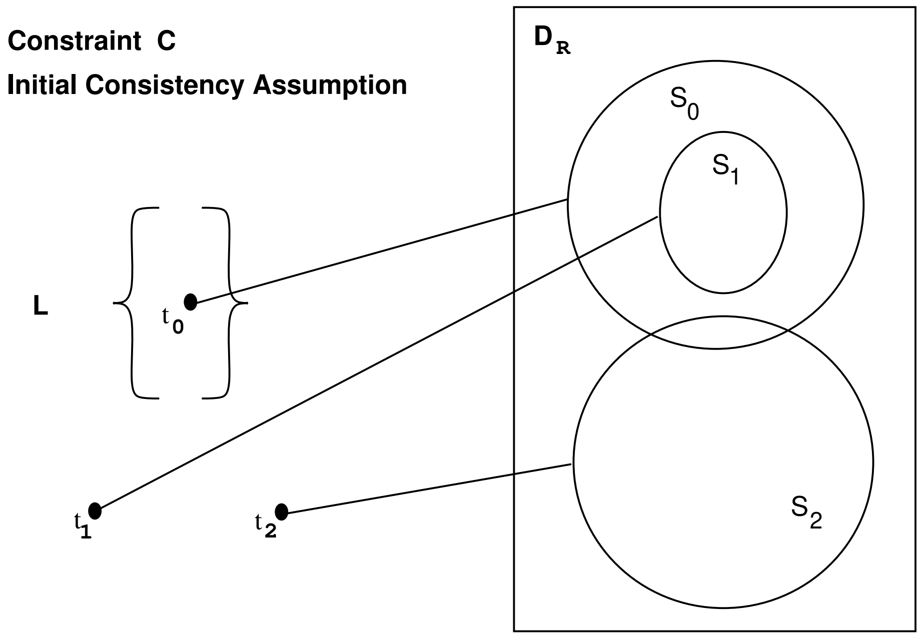

Figure 2.2: Pictorial Representation for Local Checking

Figure 2.2 considers the database with respect to a constraint C and shows the information relevant to C. Relation L, constraint specification C, and the initial consistency assumption are available and therefore are on the left hand side of the figure. The point t_0 represents an existing tuple t_0 in the accessible relation L. Points t_1 and t_2 represent tuples to be inserted into relation L. The box D_R on the right represents all the possible states of inaccessible relations $\bar{R}$. We describe the intuition for checking constraint C using the initial consistency assumption and the contents of the accessible relation L when a tuple is inserted into L. Deletions are discussed later as they do not add much to the intuition.

Circle S_0 represents all possible states of $\bar{R}$ that violate constraint C with tuple t_0, *i.e.*, the set of illegal remote database states with respect to C given t_0. The initial consistency assumption says that the current database state does not violate constraint C. Therefore, the current state of $\bar{R}$ lies outside circle S_0.

Now suppose tuple t_1 is inserted into relation L. Circle S_1 represents all the illegal states of $\bar{R}$ with respect to constraint C given tuple t_1. Circle S_1 is contained in circle S_0. Thus, all states of $\bar{R}$ that violate constraint C with tuple t_1 also violate C with tuple t_0. Given that tuple t_0 is in L and that the current state of $\bar{R}$ does not violate C with t_0 (initial consistency assumption), we infer that the current state of $\bar{R}$ lies outside circle S_0 and therefore outside circle S_1. Hence, the current state of $\bar{R}$ does not violate constraint C with tuple t_1. In this case, we say that the tuple t_0 "covers" the inserted tuple t_1.

Now suppose tuple t_2 is to be inserted into relation L, which contains tuple t_0. Circle S_2 represents all the illegal states of $\bar{R}$ with respect to constraint C given tuple t_2. Circle S_2 is not contained in circle S_0. Thus, tuple t_0 cannot be used to conclude that the current state of $\bar{R}$ does not violate C with tuple t_2.

Local checking focuses on finding cover tuples for the inserted tuple such that the cover tuples guarantee that the inserted tuple does not violate the constraint.

2.4 Formalizing the Local Checking Problem

We formalize the constraint checking strategy outlined above in the following algorithm. That is, we determine how to use the initial consistency assumption for a set of constraints $\mathbf{C} = \{C_0, C_1, \ldots, C_m\}$ each of which uses some subset of the relations $\{L, R_1, \ldots, R_n\}$, tuple t inserted into relation L, and the contents of relation L, to check the constraints $C_i \in \mathbf{C}$.

---

(1) for $i \in 0 \ldots m$ do
(2) if $\forall \bar{R} : C_i(L \cup \{t\}, \bar{R}) \Rightarrow \mathbf{C}(L, \bar{R})$ then okay$(C_i) \leftarrow$ true **else** okay$(C_i) \leftarrow$ false

---

Figure 2.3: Algorithm for locally checking constraints $\{C_0, \ldots, C_m\}$

$C_i(L, \bar{R})$ denotes the result of evaluating constraint C_i on a particular instance of relation L and on an instance of the set of relations $\bar{R}$. $\mathbf{C}(L, \bar{R})$ denotes the union of the results of evaluating constraints $\{C_0, C_1, \ldots, C_m\}$ on a particular instance of relation L and an instance of the set of relations $\bar{R}$. The algorithm checks each constraint C_i when a tuple t is inserted into L, using the old contents of relation L. The algorithm checks if any constraint in $\mathbf{C}$ is violated with relation L and $\bar{R}$ whenever C_i is violated with the relation $L \cup \{t\}$. If the implication of line (2) is true, then we can conclude that a violation of C_i with $L \cup \{t\}$ implies that some constraint $C_j \in \mathbf{C}$ was violated before inserting t, which contradicts the initial consistency assumption. The implication has the set of remote relations $\bar{R}$ universally quantified. Thus, it does not use the contents of the remote relations and uses only the contents of L. Thus the implication in line (2) represents the check for determining if the illegal states of $\bar{R}$ obtained using the tuples in relation $L \cup \{t\}$ and constraint C_i, are contained in the union of the illegal states of $\bar{R}$ obtained using every existing tuple in L and each constraint in set $\mathbf{C}$. If the implication is true, then we can conclude that the new tuple cannot possibly violate any constraint in $\mathbf{C}$ without contradicting the initial consistency assumption.

Thus, the question of interest is whether the implication of line (2) in Figure 2.3 can be solved at all, and if the implication can be solved then can it be solved efficiently and what are the desirable properties of the solution? First we discuss some of the properties that are of interest.

2.5 Properties of Local Checking Methods

Being Query based

Inferring the truth of the implication in line (2) could be expensive if we need to go through a complicated inference procedure. Therefore, we are especially interested in query-based methods for checking the implication. A local constraint checking method is query-based with respect to a query language Q if the checks can be performed by executing queries in Q over the available relation L. That is, a query-based method derives a query on the accessible local relation L such that when this query has a nonempty answer, the implication of line (2), Figure 2.3, can be inferred to be true. We refer to this query on L as the *local test*. The following example illustrates a local test, with respect to the language SQL or nonrecursive Datalog with equality.

EXAMPLE 2.5.1 Consider the scenario of Example 2.3.1. When tuple $(john, toy, 100)$ is inserted into `emp`, an existing tuple in `emp` with department attribute value toy is sufficient to infer that the new `emp` tuple does not violate the referential integrity constraint $I1$. The following query captures this logic for an arbitrary parameterized inserted tuple μ.[1] If the query derives the fact `ins_ok` then we can conclude, based on the contents of `emp`, that μ does not violate constraint $I1$.

`ins_ok` :- `emp`(E, D, S) & $D = \mu.D$

The above query encodes the reasoning of Example 2.3.1. □

The obvious advantage of a query-based approach is that it can rely on an existing query processor to find efficient algorithms for realizing partial information based constraint checking. A method that is not query-based might execute an algorithm directly on large amounts of data and may need to duplicate a lot of the data accessing functionality already available to a query processor. Query based methods can utilize the existing indices, query optimizers, and other sophisticated mechanism designed for efficient query execution. However, the use of a query on the database does not guarantee the most efficient algorithm. Thus, there may be non-query-based methods that are more efficient than any query-based method, but the non-query-based methods may require specialized inference engines, as illustrated in Example 2.5.2 below. These special methods may require enhancing the basic functionality of the database. Thus, we continue to emphasize query-based methods because these do not require enhancing the basic functionality of the database, although we do consider non-query-based methods as well.

[1] We use μ to represent the parameterized insertion and t to represent an actual insertion. A parameterized tuple is a tuple that consists of parameters and not actual values and thus captures the form, and not the content of the insertion.

Completeness

Another interesting property of local constraint checking methods is the completeness of the methods. Example 1.2.5 introduced this idea and the following example develops it further.

EXAMPLE 2.5.2 Consider the two relation database introduced in Example 1.2.5: relation `line` wherein each tuple represents an interval, and relation `point` wherein each tuple represents a point. We assume that relation `line` is local (or accessible) and that relation `point` is remote (or inaccessible). To emphasize the local/remote distinction, we use relation L to represent `line` and relation R to represent `point`. Recall constraint $I5$ from Example 1.2.5 that says that no point in relation `point` (R) can lie in an interval that is in relation `line` (L). The constraint is referred to as the *forbidden intervals* constraint and can be stated as follows.

`panic` :- $l(B, E)$ & $r(P)$ & $B \leq P \leq E$.

That is, if some point P in the remote relation is in some interval $[B, E]$ in the local relation, then constraint $I5$ is violated.

Let relation L have tuple $(3, 6)$ and let the initial consistency assumption be true, *i.e.*, constraint $I5$ holds. Let tuple $(4, 5)$ be inserted into L. We can infer that $l(4, 5)$ does not violate $I5$ by virtue of tuple $(3, 6)$ because the forbidden interval corresponding to the existing tuple contains the forbidden interval $[4, 5]$. Let us see how this inference is represented as a nonrecursive Datalog query (assuming parameterized tuple μ is inserted into relation L):

(A): `ins_ok` :- $l(B, E)$ & $\mu.B \geq B$ & $\mu.E \leq E$.

Thus, if query (A) derives `ins_ok` for an inserted tuple t, then we can infer that $I5$ continues to hold after the insertion of t into L. However, query (A) does not take full advantage of the information in L. Let relation L have tuples $(3, 6)$ and $(5, 10)$ and let the initial consistency assumption be true. Let tuple $(4, 8)$ be inserted into L. We can infer that $l(4, 8)$ does not violate $I5$ by virtue of tuples $(3, 6)$ and $(5, 10)$ because the forbidden intervals corresponding to the two existing tuples together contain the forbidden interval $[4, 8]$. Query (A) would not have made this inference because (A) uses only *one* existing tuple at a time. Thus, query (A) is not complete.

In terms of Figure 2.2, query (A) looks for one existing tuple such that its circle of illegal states contains the circle of illegal states for the inserted tuple. However, it may be the case that the union of the circles for multiple existing tuples together contain the circle of illegal states for the inserted tuple. Let us see how this more involved inference is represented as a query (assuming parameterized tuple μ is inserted into relation L):

(a): `l_union`(B, E) :- $l(B, E)$ *% Each existing tuple in L gives a forbidden interval.*
(b): `l_union`(B, E) :- `l_union`(B, X) & `l_union`(Y, E) & $X \geq Y$ *% Combine intervals.*
(c): `ins_ok` :- `l_union`(B, E) & $\mu.B \geq B$ & $\mu.E \leq E$ *% Forbidden interval for*

% inserted tuple is contained in the union of forbidden intervals for existing tuples.

It can be formally proven that the recursive Datalog rules (a), (b), and (c) capture the information available in relation L "completely." The above query cannot be represented in a language that lacks recursive or iterative constructs. Note, even though query (A) does not use the available information fully, query (A) is correct and is a sufficient local check.

We also make an observation regarding the efficiency of query based methods. The Datalog rules (a), (b), and (c), that constitute a query-based complete local constraint checking method for the forbidden interval constraint, compute the transitive closure of relation L in a naive way. Alternatively, we could use a matrix-multiplication based algorithm that is especially tuned to efficiently compute transitive closure. This method may not be query based but could be more efficient than the Datalog rules. □

Formally, completeness of a local constraint checking method is defined as follows:

Definition 2.5.1 (**Constraint Checking Completeness**) Let $\{C_0, C_1, \ldots, C_m\}$ be a set of constraints over predicates $l, r_1, \ldots, r_n$, let tuple t be inserted into relation L, and let M be a method that checks a constraint C_i $(0 \leq i \leq m)$ using only t, L, and the specifications of $\{C_0, C_1, \ldots, C_m\}$. Method M is *complete* if whenever M determines that C_i may be violated, there exist some relations $R_1, \ldots, R_n$ for predicates $r_1, \ldots, r_n$ such that: (1) no constraint in $\{C_0, C_1, \ldots, C_m\}$ is violated by $R_1, \ldots, R_n$ and L; (2) C_i is violated by $R_1, \ldots, R_n$ and $L \cup \{t\}$. □

That is, a method is complete if, whenever the method cannot determine that a constraint holds, then there is some value of the inaccessible information that together with the value of the accessible information, violates the constraint with insertion t. Informally, a complete method "does the best it can with the information it has."[2] We use the term *complete local test* to refer to the local query produced by a query-based method that is complete. Example 2.5.2 illustrates many points:

- Some query-based methods are "sufficient" and not "complete."
- A complete query-based constraint checking method may need a more expressive query language to express the resulting query than the language needed to represent the query produced by a reasonable sufficient method. Thus, it may not be possible to utilize a complete method in a system because the language supported by the system is not expressive enough.
- Complete methods may be more expensive to evaluate than sufficient ones. For instance, nonrecursive query (A) may be less expensive to execute than the recursive query expressed using rules (a), (b), and (c).

[2]The completeness of a method is impacted by additional information about the database, like functional dependencies, restrictions on domains of attributes *etc.* Using extra information allows more powerful methods, rendering otherwise complete methods incomplete.

- Methods that are not complete may be so to "different degrees." For instance, in Example 2.5.2 consider a method that uses *two* existing intervals to check if the inserted interval does not violate the forbidden interval constraint. Using two tuples is better than using just one but not as good as using an arbitrary number of tuples.

The algorithm in Figure 2.3 is a complete local checking method assuming that the implication in line (2) can be solved. If the constraint is expressed in a language for which implication is undecidable, then it is not possible to have any complete method for such constraints. An example of such a language is Datalog with recursion. Even if the implication is undecidable in general, for some cases it may be possible to infer that the implication holds, thereby yielding sufficient methods. Thus, it is of interest to determine if complete methods exist for given constraints or if only sufficient methods exist, and in each case their properties need to be studied.

2.6 Approaches for Exploring Local Checking Methods

Our strategy is to consider increasingly expressive constraint specification languages and derive local checking methods for them. We follow the path outlined by Figure 2.4 which is an elaboration of the "Y" originally introduced in Figure 1.1.

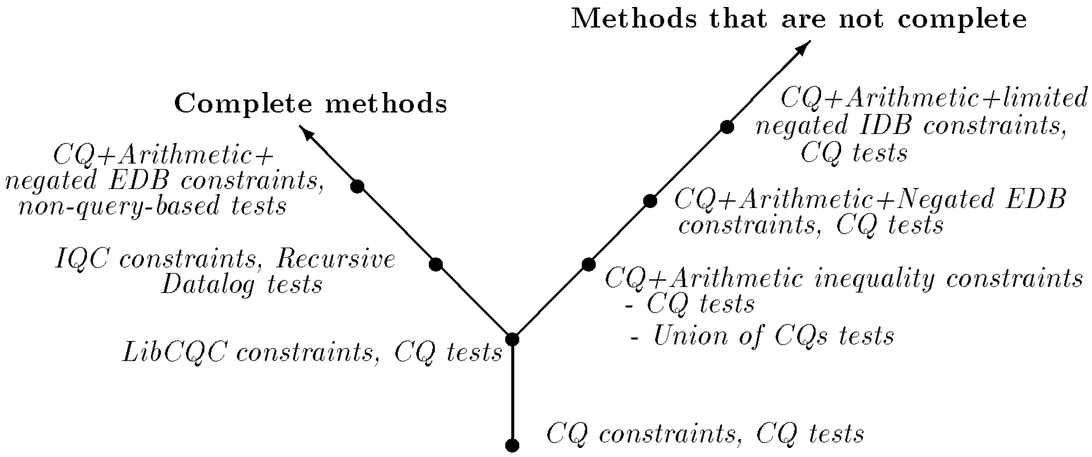

Figure 2.4: Possible Paths to Explore for Local Checking

Informally, the stem of the "Y" represents constraint classes for which complete local checking methods exist and the local query produced by these methods is expressible in a "simple" language such as conjunctive queries. As we consider more expressive constraint languages, we have to choose between complicated complete methods and simpler sufficient methods. We explore the two alternative routes as illustrated by the left and right branches of the "Y" in Figure 2.4.

- *Retain Completeness.* This route corresponds to the left branch of the "Y." The language used to express the local query derived by the complete methods may be quite complex. In some cases, we may not have a query based test.

- *Sacrifice Completeness.* This route corresponds to the right branch of the "Y." Even for constraint classes that do have complete query based methods, if the language used to express the local query is restricted, then completeness may be compromised as illustrated in Example 2.5.2. Additionally, if the implication of line (2) is undecidable, then no complete local checking method will exist but sufficient methods may still exist. Thus it is possible to go further vertically on this branch than on the left branch of the "Y."

Recall, going up vertically in Figure 2.4 corresponds to considering more expressive constraint languages. The first class of constraints that we consider, are conjunctive query constraints with arithmetic inequalities. For this class of constraints, we show how the local checking problem can be reduced to the query containment problem. Using this reduction and the results of Appendix A, we convert the implication problem on line (2) of Figure 2.3 to an implication I_t that involves only arithmetic inequalities. Thus, we build the necessary background for developing local queries for checking conjunctive query constraints that use arithmetic inequalities.

Chapter 3 discusses how to convert the resulting arithmetic implication I_t to a complete local query on L, for two restricted classes of conjunctive queries with arithmetic inequalities. The chapter also discusses how to evaluate implication I_t using a non-query-based method but while retaining completeness [Dav87, SKN89]. The results of Chapter 3 correspond to the stem of the "Y" and the points labelled *IQC*s and LibCQC (we formally define these classes in Chapter 3). The point on the stem "*CQ* constraints" is a special case of LibCQC constraints.

Subsequently, in Chapter 4 we give up completeness and develop sufficient queries for checking the general class of conjunctive queries with arithmetic inequalities. Then we extend the constraint language to a superset of conjunctive queries and develop query-based methods that are not complete. Thus, Chapter 4 considers the right branch of the "Y" in Figure 2.4.

The following sections of this chapter provide the groundwork for the results in Chapters 3 and 4.

2.7 Conjunctive Query Constraints with Arithmetic Inequalities

In the remainder of this chapter, and in the next chapter, we consider constraint queries of the following form:

$$C\colon\ \texttt{panic} :- l(\bar{X})\ \&\ r_1(\bar{Y}_1)\ \&\ \ldots\ \&\ r_n(\bar{Y}_n)\ \&\ c_1\ \&\ \ldots\ \&\ c_k$$

where relation L is accessible and relations $R_1, \ldots, R_n$ are inaccessible. $l, r_1, \ldots, r_n$ are referred to as *ordinary* predicates and c_i is a arithmetic comparison of the form X `op` Y, where each of X and Y is either an argument of some ordinary predicate or a constant, and `op` is one of $<, >, \leq, \geq, =, \neq$.

Predicate l has only one occurrence in the constraint. The remote predicates can appear multiple times. All shared and constant value arguments are represented explicitly by equalities in the c_i's. Therefore, no argument is repeated in any of the ordinary predicates. The conjunction of the arithmetic comparisons is referred to as $I(C)$ and the conjunction of the ordinary predicates is referred to as $O(C)$. We refer to constraint queries of the above form as Conjunctive Query Constraints or *CQCs*.

We use the term *query containment* as defined in [Ull89]. A query $Q1$ is contained in another query $Q2$ if for every database the set of answer derived by $Q1$ is a subset of the answers derived by $Q2$.

2.7.1 Relationship Between Conjunctive Query Containment and Local Checking

We use the forbidden intervals constraint $I5$ from Example 2.5.2 to show that local checking can be cast as a query containment problem. Note, the violation condition for $I5$ can be stated as a CQC.

EXAMPLE 2.7.1 Constraint $I5$ from Page 24 is violated when the following rule derives `panic`.

E: `panic` $:- \, l(X,Y) \;\&\; r(Z) \;\&\; X \leq Z \;\&\; Z \leq Y$.

where l is `line` and r is `point`. Example 2.5.2 argued that if relation L had tuples $(3,6)$ and $(5,10)$, then inserting tuple $(4,8)$ into L did not violate constraint $I5$ if the constraint was valid before the insertion. Let's look at this inference from a query containment perspective. The tuples of the inaccessible remote relation R that violate constraint $I5$ given tuple $l(4,8)$ satisfy the following partially instantiated conjunctive query:

A: `panic` $:- \, r(Z) \;\&\; 4 \leq Z \leq 8$.

That is, all tuples $r(Z)$ such that Z lies in the interval $[4,8]$ violate $I5$ given tuple $l(4,8)$. Similarly, the values of Z that violate $I5$ using tuples $(3,6)$ and $(5,10)$ satisfy the partially instantiated conjunctive queries A_1 and A_2:

A_1: `panic` $:- \, r(Z) \;\&\; 3 \leq Z \leq 6$.

A_2: `panic` $:- \, r(Z) \;\&\; 5 \leq Z \leq 10$.

It is the case that $A \subseteq A_1 \cup A_2$. That is, whenever conjunctive query A derives `panic`, one of A_1 or A_2 also derives `panic`. The initial consistency assumption tells us that neither A_1 or A_2 derive `panic`. Therefore, we can conclude that A does not derive `panic` either and thus the inserted tuple $(4,8)$ does not violate constraint $I5$. $\square$

In terms of Figure 2.1, the partially instantiated query A obtained by instantiating the constraint query C with tuple $l(4,8)$ for relation L is satisfied by every illegal state of the inaccessible data.

The partially instantiated query A represents the illegal circle of states corresponding to tuple $l(4,8)$. The following definition formalizes how to represent the illegal states of the inaccessible data using a partially instantiated conjunctive query.

Definition 2.7.1 ($\mathsf{Red}(t,l,C)$) Consider a CQC C as defined in Section 2.7 and let t be a tuple in relation L for predicate l. The *reduction* of C by tuple t, $\mathsf{Red}(t,l,C)$, is the partially instantiated CQC obtained by instantiating with tuple t the subgoal g that uses predicate l, then replacing all occurrences of the variables in g by the constants they get unified with, and finally eliminating g. □

Note, in $\mathsf{Red}(t,l,C)$ the constants introduced by t are propagated to the arithmetic comparisons $I(C)$ in C. Thus, $\mathsf{Red}(t,l,C)$ characterizes those states of the remote relations that violate CQC C given $L = \{t\}$. This characterization does not apply if the local predicate appears multiple times. The following example illustrates when $\mathsf{Red}(t,l,C)$ does not characterize the illegal states of the remote database given tuple t.

`panic` :- $l(X,Y)$ & $l(Y,Z)$ & $r(X,Z)$.

Predicate l has two occurrences and the constraint could potentially be reduced twice by tuple t: once for each occurrence of l. However, each of the reductions would regard one occurrence of l as being remote. Thus, useful information might be lost and the completeness of local checking could be compromised. Instead, we assume that if a local relation occurs multiple times, we merge all the occurrences into one new relation and work with this new relation as the local relation. Thus, despite the requirement that l occur only once in the constraint, the language we consider has the same expressive power as conjunctive queries with arithmetic constraints, albeit with some preprocessing.

2.7.2 Results on Conjunctive Query Containment

Before we build on Example 2.7.1 to show how conjunctive query containment is used for integrity constraint checking with partial information, we state some results on CQC containment. The results stated below are special cases of more general results stated and proved in Appendix A.

Definition 2.7.2 (**Symbol Mapping**) A symbol mapping h is a function from a set of symbols S to another set of symbols T; *i.e.*, for each symbol $a \in S$, $h(a)$ is a symbol in T. Consider two CQCs q_1 and q_2 and let S be the set of variables in q_1. Let h be a symbol mapping on S where $h(X)$ can be an arbitrary term and h is the identity mapping on predicate names, constants, and function symbols. h is a symbol mapping from q_1 to q_2 if h turns every subgoal in $O(q_1)$ into some subgoal in $O(q_2)$. □

EXAMPLE 2.7.2 Consider the CQCs q_1 and q_2.

q_1: `panic` :– $q(X, Y)$ & $r(U, V)$ & $r(W, Z)$ & $W = V$ & $U = Z$.

q_2: `panic` :– $q(X, Y)$ & $r(U, V)$ & $U < V$.

The set of symbol mappings $\mathcal{M}$ from q_2 to q_1 is:

1. $X \rightarrow X,\ Y \rightarrow Y,\ U \rightarrow U,\ V \rightarrow V$.
2. $X \rightarrow X,\ Y \rightarrow Y,\ U \rightarrow W,\ V \rightarrow Z$. □

The following two theorems on the containment of a CQC in a (set of) CQC(s) are special cases of Theorems A.2.1 and A.2.2 respectively, stated in Appendix A.

Theorem 2.7.1 *Suppose q_1 and q_2 are two CQCs that use the arithmetic comparisons $<, >, \leq, \geq, =, \neq$. Let $\mathcal{M}$ be the set of symbol mappings from CQC q_2 to q_1. Then $q_1 \subseteq q_2$ if and only if $\forall \bar{X}\ \exists h_{in\ \mathcal{M}} : [I(q_1) \Rightarrow h(I(q_2))]$ ($\bar{X}$ is the set of variables in q_1).* □

Proof: Special case of Theorem A.2.1. ∎

Theorem 2.7.2 *Suppose $\{q_1, \ldots, q_m\}$ are CQCs that use the arithmetic comparisons $<, >, \leq, \geq, =$. Let q be another CQC of the same form. Let $\mathcal{M}_i$ be the set of symbol mappings from CQC q_i to q. Then $q \subseteq \{q_1 \cup q_2 \cup \ldots \cup q_k\}$ if and only if $\forall \bar{X}\ \exists q_i\ \exists h_{in\ \mathcal{M}_i} : [I(q) \Rightarrow h(I(q_i))]$.* □

Proof: Special case of Theorem A.2.2. ∎

Theorem A.2.2 that addresses the containment of CQs with arithmetic inequalities is also stated in [ZO93]. Our results were developed independently from the results in [ZO93].

2.7.3 Using Containment for Local Checking

In this section we discuss how to use conjunctive query containment for realizing the algorithm for partial-information-based constraint checking in Figure 2.3. The algorithm checks the implication in line (2), *i.e.*, $\forall \bar{R} : C_i(L \cup \{t\}, \bar{R}) \Rightarrow \mathbf{C}(L, \bar{R})$. Example 2.7.1 illustrates this case when $\mathbf{C}$ contains only constraint $I5$. In this section we consider the general case when $\mathbf{C}$ contains multiple constraints and show how the above implication can be evaluated using conjunctive query containment. The following lemma formalizes the intuition of the example and considers only one constraint in $\mathbf{C}$.

Lemma 2.7.1 *Let C be a CQC and let t be a tuple inserted into relation L for predicate l. Assume C holds before the insertion. Then, $\forall \bar{R} : C(L \cup \{t\}, \bar{R}) \Rightarrow C(L, \bar{R})$ if and only if $\mathsf{Red}(t, l, C)$ is contained in $\bigcup_{s\ in\ L} \mathsf{Red}(s, l, C)$. That is, the complete local method for guaranteeing that constraint C holds after inserting t into L checks whether $\mathsf{Red}(t, l, C)$ is contained in $\bigcup_{s\ in\ L} \mathsf{Red}(s, l, C)$.* □

Proof: The lemma follows from the following observations:

1. $C(\{t\}, \bar{R})$ derives `panic` for a particular set of relations $\bar{R}$ if and only if $\mathsf{Red}(t, l, C)$ derives **panic** with $\bar{R}$. This observation holds because C has only one occurrence of predicate l and this occurrence is not negated (follows from results in [Nic82]).

2. $C(L \cup \{t\}, \bar{R}) \equiv [C(L, \bar{R}) \vee C(\{t\}, \bar{R})]$. This observation holds because C has one non-negative occurrence of predicate l (follows from results in [Nic82]).

3. $C(L \cup \{t\}, \bar{R}) \Rightarrow C(L, \bar{R})$ is equivalent to $C(\{t\}, \bar{R}) \Rightarrow C(L, \bar{R})$. This observation is obtained from Observation 2 and the logical equivalence $(A \vee B) \Rightarrow A$ if and only if $B \Rightarrow A$.

■

Lemma 2.7.1 can be generalized to the case when there are multiple constraints in **C**.

Theorem 2.7.3 *Let $\{C_0, C_1, \ldots, C_m\}$ be CQCs, and let t be a tuple inserted into relation L for predicate l. Assume each C_i holds before the update. The complete local method for guaranteeing that constraint C_0 holds after inserting t into L checks whether $\mathsf{Red}(t, l, C_0)$ is contained in $\bigcup_{j=0}^{m} \bigcup_{s \text{ in } L} \mathsf{Red}(s, l, C_j)$.* □

The results of Theorem 2.7.2 can be used to solve the containment question posed by Theorem 2.7.3 because $\mathsf{Red}(t, l, C_0)$ is a CQC. If we map the containment question of Theorem 2.7.3 to the statement of Theorem 2.7.2, we see that $\mathsf{Red}(t, l, C_0)$ corresponds to query q in Theorem 2.7.2, and each $\mathsf{Red}(s, l, C_j)$ corresponds to some q_p in the set $\{q_1, \ldots, q_k\}$ in Theorem 2.7.2.

The question that remains to be answered is how to decide the containment question $q \subseteq \{q_1 \cup q_2 \cup \ldots \cup q_k\}$. Theorem 2.7.2 states the information needed for the evaluation. This information is listed below along with some observations that hold in the case of Theorem 2.7.3.

1. *Sets of symbol mappings $\mathcal{M}_1, \ldots, \mathcal{M}_m$ from each $\mathsf{Red}(s, l, C_j)$ to $\mathsf{Red}(t, l, C_0)$.*
   Recall, symbol mappings from one CQC to another CQC are computed using only the ordinary predicates involved in the CQCs. Thus, the mappings from $\mathsf{Red}(s, l, C_j)$ to $\mathsf{Red}(t, l, C_0)$ depend only on the remote predicates $r_1, \ldots, r_n$ and do not depend on the tuples t or s. Therefore, for each constraint C_j we can compute the mappings $\mathcal{M}_j$ once, using a parameter μ to represent the inserted tuple t and a parameter α to represent existing tuple s in L. Thus, $\mathcal{M}_j$ is the set of mappings from $\mathsf{Red}(\alpha, l, C_j)$ to $\mathsf{Red}(\mu, l, C_0)$.

2. *$h(I(q_p))$, that is, $h(I(\mathsf{Red}(s, l, C_j)))$.*
   Recall, $I(q)$ represents the conjunction of interpreted subgoals in query q, and h is a mapping from $\mathsf{Red}(\alpha, l, C_j)$ to $\mathsf{Red}(\mu, l, C_0)$. Mapping h is the identity mapping on all constants and affects only the variables in $\mathsf{Red}(\alpha, l, C_j)$. Therefore, h does not affect the parameter α. Similarly, any assignment ρ of the parameter α to a tuple s in L, does not affect the variables in C_j. Therefore, $h(I(\mathsf{Red}(s, l, C_j)))$ is the same as $h(I(\mathsf{Red}(\rho(\alpha), l, C_j)))$ is the same as $\rho(h(I(\mathsf{Red}(\alpha, l, C_j))))$.

3. *$I(q)$, that is,* $I(\mathsf{Red}(t, l, C_0))$.

   As in Item 2, an assignment ρ of parameter μ to tuple t, does not interfere with the variables in C_0 and therefore $I(\mathsf{Red}(t, l, C_0))$ can be obtained by applying ρ to $I(\mathsf{Red}(\mu, l, C_0))$.

The above observations allow us to break the containment question of Theorem 2.7.3, $\mathsf{Red}(t, l, C_0) \subseteq \bigcup_{j=0}^{m} \bigcup_{s \text{ in } L} \mathsf{Red}(s, l, C_j)$, into four steps.

1: Obtain the parameterized query $\mathsf{Red}(\mu, l, C_0)$ and the set of parameterized queries $\{\mathsf{Red}(\alpha, l, C_0), \mathsf{Red}(\alpha, l, C_1), \ldots, \mathsf{Red}(\alpha, l, C_m)\}$. Compute the set of mappings $\mathcal{M}_i$ from each $\mathsf{Red}(\alpha, l, C_i)$ to $\mathsf{Red}(\mu, l, C_0)$.

2: Create a parameterized test condition T that is in the form of an implication.
$\mathsf{LHS}(T)$ is $I(\mathsf{Red}(\mu, l, C_0))$, and $\mathsf{RHS}(T)$ is $\bigvee_{j=0}^{m} \bigvee_{h \text{ in } \mathcal{M}_j} h(I(\mathsf{Red}(\alpha, l, C_j)))$.

3: Obtain the instantiated test condition I_t as follows:
Instantiate parameter μ in $\mathsf{LHS}(T)$ by tuple t to obtain LHS of I_t: $I(\mathsf{Red}(t, l, C_0))$.
Instantiate parameter α in $\mathsf{RHS}(T)$ by each $s \in L$. Add the resulting term as a disjunct to the RHS of I_t to obtain: $\bigvee_{j=0}^{m} \bigvee_{h \text{ in } \mathcal{M}_j} \bigvee_{s \text{ in } L} h(I(\mathsf{Red}(s, l, C_j)))$.
Thus, I_t is $[I(\mathsf{Red}(t, l, C_0)) \Rightarrow \bigvee_{j=0}^{m} \bigvee_{h \text{ in } \mathcal{M}_j} \bigvee_{s \text{ in } L} h(I(\mathsf{Red}(s, l, C_j)))]$.

4: Check if instantiated test condition I_t is true.

Thus, for a given set of constraints it is possible to compute the required symbol mappings and then produce a parameterized test condition T without using either relation L or inserted tuple t. When insertions are made into relation L, then test condition T is instantiated and the resulting implication I_t is evaluated. Test T needs to be derived only once, at constraint specification time, and the remainder can be done when insertions are made into L.

The following example illustrates this process:

EXAMPLE 2.7.3 Consider the "forbidden interval" constraint from Example 2.7.1:

E: $\texttt{panic} :- l(X,Y) \ \& \ r(Z) \ \& \ X \leq Z \ \& \ Z \leq Y.$

Let parameter μ be $l(a, b)$, and parameter α be $l(x, y)$. The first step of the process yields:

$\mathsf{Red}(\mu, l, E)$ is $\quad \texttt{panic} :- r(Z) \ \& \ a \leq Z \leq b.$
$\mathsf{Red}(\alpha, l, E)$ is $\quad \texttt{panic} :- r(Z) \ \& \ x \leq Z \leq y.$

The set of mappings from $\mathsf{Red}(\alpha, l, E)$ to $\mathsf{Red}(\mu, l, E)$ contains just the identity mapping. Therefore, the second step of the process yields the following parameterized test condition:

$\mathsf{LHS}(T)$: $\quad a \leq Z \leq b.$
$\mathsf{RHS}(T)$: $\quad x \leq Z \leq y.$

Now, let us consider a specific database and see how to use step 3 to generate the instantiated test condition I_t from the above parameterized test condition template. We consider the database from Example 2.7.1 where relation L has tuples $l(3,6)$ and $l(5,10)$, and tuple $l(4,8)$ is inserted. Parameter μ is instantiated in $\mathsf{LHS}(T)$ by $(4,8)$ to obtain $4 \leq Z \leq 8$. Parameter α is instantiated by tuples $l(3,6)$ and $l(5,10)$, resulting in two disjuncts in the RHS of I_t; one each for the two existing tuples. Thus, I_t is:

$$I_t\colon\quad 4 \leq Z \leq 8 \;\Rightarrow\; 3 \leq Z \leq 6 \quad \vee$$
$$5 \leq Z \leq 10.$$

I_t evaluates to true because every point in the interval $[4,8]$ is either in the interval $[3,6]$ or in the interval $[5,10]$. □

Verifying the truth of the resulting implication I_t is a complex problem in general. The complexity of evaluating I_t depends on the nature of the arithmetic inequalities permitted in the constraint queries. If the inequalities are restricted adequately, then I_t may be represented as a union of conjunctive queries or as a recursive Datalog program. These restrictions are explored in the next chapter.

Chapter 3

Conjunctive Query Constraints

The previous chapter reduces the problem of locally checking conjunctive query constraints with arithmetic inequalities to an implication I_t of arithmetic inequalities. In this chapter we consider restrictions on the arithmetic inequalities permitted in the constraints so as to be able to solve the arithmetic implication using Datalog and conjunctive queries with arithmetic. We want to generate query-based methods for a subset of the class of conjunctive query constraints with arithmetic inequality. In this chapter we consider the three circled points on the "Y" in Figure 3.1. Namely, the class of conjunctive queries without any arithmetic, and two restricted classes of conjunctive queries with arithmetic inequalities. We consider these points in the reverse order, *i.e.*, top down in the "Y."

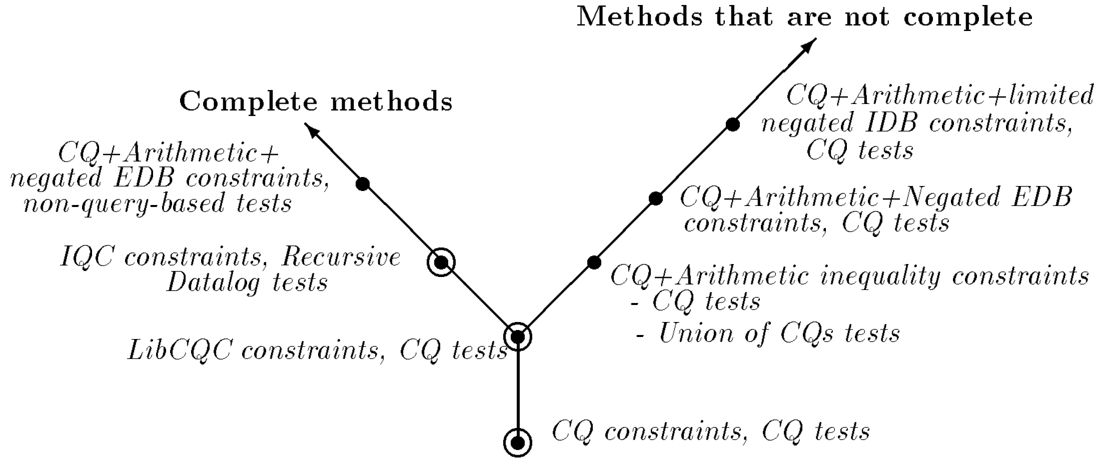

Figure 3.1: Constraint Classes Considered in Chapter 3

Chapter Outline In this chapter first we consider how to generate queries on relation L using the test condition I_t for subclasses of CQCs. The first subclass we consider corresponds to the topmost circled point in the "Y" of Figure 3.1. For this subclass the complete local test is expressible as a recursive Datalog program. Then in Section 3.2 we consider a further restriction on CQCs with arithmetic inequalities such that the complete local test for checking this subclass can be represented as a union of conjunctive queries. This class corresponds to the middle circled point in Figure 3.1. The bottom-most circled class is a special case of the middle circled class.

Section 3.3 discusses the complexity of evaluating the queries derived earlier in the chapter and also discusses other ways of evaluating I_t that are not query-based and apply to the general class of conjunctive query constraints with arithmetic inequalities. These techniques do not convert I_t into a Datalog or SQL query, but instead use complex data structures for efficient inference. That is, instead of using local queries, we consider inference techniques for evaluating I_t for the general class of CQCs. Often these inference techniques are more efficient than query-based methods for the classes that do have query-based methods. The general solution for I_t corresponds to developing a method for a subset of the topmost, uncircled point on the left branch of the "Y." Finally, Section 3.4 remarks on the application of results of Appendix A to local checking of conjunctive query constraints that use arbitrary interpreted predicates.

3.1 Characterizing the Class of IQC Constraints

In this section we describe a subclass of CQCs for which a Datalog program can be used to first instantiate the test condition T to give arithmetic implication I_t, and then evaluate I_t.

Definition 3.1.1 (**Independently Constrained Variable**) Consider a conjunction of arithmetic comparisons that use the operators $<, >, \leq, \geq, =$. A variable X is independently constrained if all the comparisons that involve X are of the form X `op` c where c is a constant. □

Definition 3.1.2 (**Independently Constrained Sentence**) Consider a conjunction of arithmetic comparisons that use the predicates $<, >, \leq, \geq, =$. The sentence is independently constrained if all the variables that appear in the sentence are independently constrained. □

Definition 3.1.3 (**Independently Constrained Query (IQC)**) Consider CQC C where equated attributes are represented by using the same attribute multiple times, and not by explicit equalities.[1] If $I(\mathsf{Red}(t, l, C))$ is an Independently Constrained Sentence, then the query C is an Independently Constrained Query. □

The forbidden interval query E of Example 2.7.1 is an IQC.

EXAMPLE 3.1.1 The forbidden interval query is:

[1] Equated attributes represent natural joins. The definition of IQCs uses shared attributes to represent joins purely for definition purposes. The IQC can actually be written by representing every join as an explicit equality, as assumed in the rest of this thesis. The two alternative representations are equivalent and each can be generated from the other in time $O(v^2)$ where v is the number of attribute occurrences in the query.

E: `panic` :– $l(X, Y)$ & $r(Z)$ & $X \leq Z$ & $Z \leq Y$.

Relation L is local. Consider $\mathsf{Red}(t, l, E)$ for some tuple $t = (a, b)$. The resulting conjunctive query is:

`panic` :– $r(Z)$ & $a \leq Z$ & $Z \leq b$.

The conjunction of arithmetic subgoals in $\mathsf{Red}(t, l, E)$ is referred to as $I(\mathsf{Red}(t, l, E))$ (Section 2.7) and this conjunction is an independently constrained sentence. Thus the original query E is an IQC. □

EXAMPLE 3.1.2 Consider another constraint query:

`panic` :– $l(X, Y)$ & $r_1(W, U)$ & $r_2(Z, V)$ & $V > X$ & $Z = W$.

First, we rewrite the above query in the form required by Definition 3.1.3. That is, we replace equated attributes by the same attribute. In this case we replace attribute Z in $r_2(Z, V)$ by W.

`panic` :– $l(X, Y)$ & $r_1(W, U)$ & $r_2(W, V)$ & $V > X$.

The reduction of the above query with respect to tuple $l(a, b)$ is:

`panic` :– $r_1(W, U)$ & $r_2(W, V)$ & $V > a$.

The conjunction of arithmetic subgoals in the above query form an independently constrained sentence thereby implying that the original query is an IQC. □

In the following sections we consider IQCs, and for these constraints we discuss how test T can be represented by Datalog rules that use relation L and inserted tuple t such that evaluating these rules corresponds to instantiating T and evaluating the resulting implication I_t. The following example illustrates the process for the forbidden intervals constraint.

EXAMPLE 3.1.3 Intuition: Example 2.7.3 described how the parameterized test T is derived and instantiated for the forbidden interval constraint. Recall that T was defined as follows:

$\mathsf{LHS}(T)$: $a \leq Z \leq b$.
$\mathsf{RHS}(T)$: $x \leq Z \leq y$.

where the inserted tuple was represented by parameter $\mu = l(a, b)$, and an existing tuple was represented by parameter $\alpha = l(x, y)$. Example 2.7.3 showed how to instantiate test T using inserted tuple $l(4, 8)$ and existing tuples $l(3, 6)$ and $l(5, 10)$ to obtain I_t as:

I_t: $4 \leq Z \leq 8 \;\Rightarrow\; (3 \leq Z \leq 6 \;\vee\; 5 \leq Z \leq 10)$.

Consider an alternative approach to the instantiation process. The solutions to variable Z in a sentence of the form $x \leq Z \leq y$ is a closed interval that can be represented by a binary predicate whose first attribute represents the lower limit of the solution space and second attribute represents the upper limit. Note, all the conjuncts in test T, both $\mathsf{LHS}(T)$ and $\mathsf{RHS}(T)$, are sentences of the above form. First, we consider $\mathsf{LHS}(T)$. Solutions to the remote variable Z that occurs in $\mathsf{LHS}(T)$ are stored as tuples of relation `ins` and computed by the following rule:

$\texttt{ins}(A, B)$:- $\texttt{inserted\_into\_L}(A, B)$ & $A \leq B$ % *Use the inserted tuple to instantiate* `ins`.

The relation `inserted_into_L` contains the inserted tuples and each inserted tuple t adds one `ins` tuple representing the forbidden interval for t. Thus `inserted_into_L` instantiates parameter μ that originally appears in $\mathsf{LHS}(T)$. Similarly, the solutions to variable Z that occurs in $\mathsf{RHS}(T)$ can be represented by a binary predicate `soln` that is defined as:

$\texttt{soln}(C, D)$:- $l(C, D)$ & $C \leq D$ % *Use existing tuples in relation L to initialize* `soln`.

The above rule uses predicate l to define the basic intervals that are in predicate `soln`. Thus, predicate l instantiates the parameter α that originally appears in $\mathsf{RHS}(T)$ and for each instantiation, a tuple is added to `soln`. These basic intervals may have to be combined in order to obtain larger intervals, as argued earlier. The following rule does this combination.

$\texttt{soln}(C, D)$:- $\texttt{soln}(C, X)$ & $\texttt{soln}(Y, D)$ & $X \geq Y$ % *Combine* `soln` *facts.*

Thus, for a given inserted tuple t and relation L, the above rules compute `ins` and `soln` facts that respectively represent the inserted and existing intervals. The instantiation of test T is done by predicates `inserted_into_L` and l. The resulting implication I_t evaluates to true if and only if the following Datalog rule derives `ins_ok`.

`ins_ok` :- $\texttt{ins}(A, B)$ & $\texttt{soln}(C, D)$ & $A \geq C$ & $B \leq D$.

% *Solution space for inserted tuple is contained in union of spaces for existing tuples.*

□

The above example illustrates how to combine the instantiation of the parametric test T with the evaluation of the resulting arithmetic implication I_t to produce a Datalog program that derives `ins_ok` if implication I_t evaluates to true. In subsequent sections we give the geometric intuition for obtaining such a Datalog program for IQCs in general and then describe the actual process for building the program.

3.1.1 Geometric Intuition

Recall from Page 31 that test T is built using the arithmetic subgoals of the constraints in set $\mathbf{C}$, *i.e.*, T uses $I(\mathsf{Red}(t, l, C))$ for every constraint $C \in \mathbf{C}$. For IQCs, every variable in $I(\mathsf{Red}(t, l, C))$ is compared only to constants (by Definition 3.1.3). Therefore, every variable has constants as the lower and upper limits of its solution space. The set of constants includes plus and minus infinity. Thus, the solution space for the variables that occur in $I(\mathsf{Red}(t, l, C))$ corresponds to a k-dimensional parallelepiped.

EXAMPLE 3.1.4 Consider an IQC $I6$ that extends the forbidden interval constraint to two dimensions. Thus, relation L stores rectangles and R stores points in 2-dimensions and $I6$ is violated if any point in R is in a rectangle in L. Thus, $I6$ is expressed as the following Datalog rule.

$\texttt{panic} :- l(U, V, W, Z)\ \&\ r(X, Y)\ \&\ U \leq X \leq V\ \&\ W \leq Y \leq Z.$

The tuple $l(2, 20, 4, 40)$ in relation L violates constraint $I6$ if relation R contains a tuple $r(a, b)$ such that $2 \leq a \leq 20$ and $4 \leq b \leq 40$. The values of a and b that violate $I6$ given $l(2, 20, 4, 40)$ lie in a rectangle of the above dimension such that a is along one dimension and b is along the other dimension. If $I6$ is not violated by tuple $l(2, 20, 4, 40)$, then we can conclude that the tuples of R lie outside this rectangle.

In general, the initial consistency assumption implies that every tuple of L identifies a prohibited rectangle in R. A tuple t can be inserted safely into L if the prohibited rectangle of solutions for $r(X, Y)$ defined by t is contained in the union of prohibited rectangles defined by the existing tuples in L. □

Recall, for a single constraint C the implication I_t has $I(\mathsf{Red}(t, l, C))$ as its LHS and a disjunction, $\bigvee_{t_i \in L} \bigvee_{h \in \mathcal{M}} h(I(\mathsf{Red}(t_i, l, C)))$, as its RHS. For an IQC C the solution to any $I(\mathsf{Red}(t_i, l, C))$ is a k-dimensional parallelepiped. Thus, verifying I_t corresponds to determining if the k-dimensional parallelepiped corresponding to the LHS of I_t is contained in the union of k-dimensional parallelepipeds for each disjunct in the RHS of I_t. In this section, we discuss a solution to this problem. The solution is developed for an IQC with 2 variables in $I(\mathsf{Red}(t, l, C))$, in which case k is 2 and the parallelepiped is a rectangle.

Computing the union of rectangles is more involved than computing the union of lines (as in the case of Example 3.1.3). Two adjacent rectangles can be combined as illustrated in Figure 3.2(a).

Figure 3.2(b) illustrates how to check if an existing set of rectangles contain a new rectangle. The new rectangle is drawn shaded in the figure. The three existing rectangles are drawn unshaded and have thick outlines. Each existing rectangle can be fragmented into smaller basic rectangles by using the sides of the other existing rectangles as flexion lines. In the figure, the flexion lines are represented by broken lines. The basic rectangles are bounded by a mixture of broken and thick lines and are labelled by the letters $A - T$. No lines pass through a basic rectangle. In Figure 3.2, the basic rectangles D, E, F, G, I, J, K, L together contain the new rectangle. In general, it can be proved that a new rectangle is contained in the union of existing rectangles if and only if it is contained in a rectangle formed by combining some basic rectangles (Appendix C).

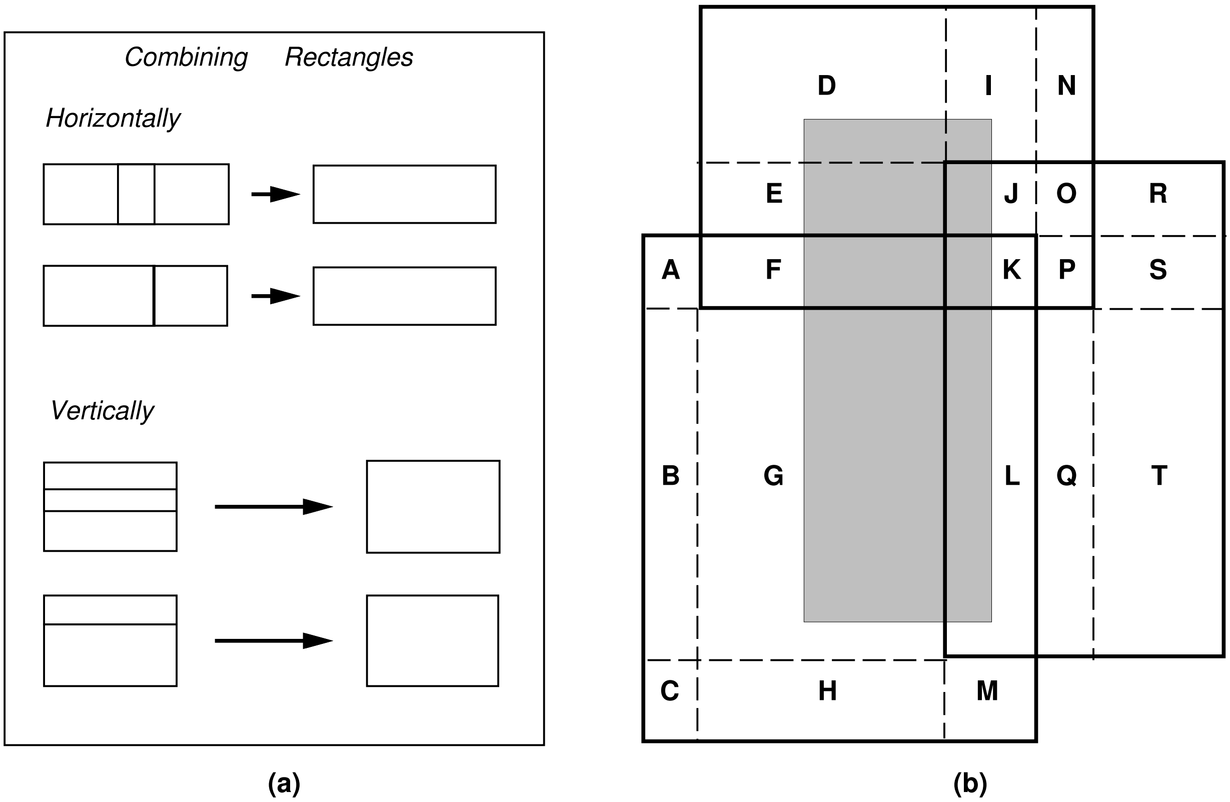

Figure 3.2: Determining Containment of Rectangles

The idea extends to k-dimensions where each existing k-dimensional parallelepiped is broken into smaller, basic k-dimensional parallelepipeds which are then recombined to create the containing parallelepiped. Appendix C discusses this extension.

3.1.2 Building the Datalog Rules

In this section, we describe how to build Datalog rules using the parameterized test T, relation L, and tuple t. The Datalog program instantiates T to derive I_t and then encodes the rectangle containment reasoning outlined above. We give only the intuition for building the Datalog program. The details of the (substantial) formalism are discussed in Appendix C.

Just as a 2-ary predicate is used to represent the prohibited range of values for a single remote variable as in Example 3.1.3, and a 4-ary predicate is used for to represent the prohibited rectangle of values for two independently constrained remote variables as in Example 3.1.4, a $2k$-ary predicate can be used for to represent the k-dimensional parallelepiped for k remote variables. For simplicity, we will restrict the discussion in this section to two variables.

The strategy for building the Datalog program is as follows:

1. Use a 4-ary predicate `ins` to represent the solutions to the variables in the LHS of parameterized test T. That is, `ins` stores solutions to the remote variables that appear in $I(\mathsf{Red}(\mu, l, C))$. Similarly, define predicate `soln` to represent the solution to the variables in each disjunct, $h(I(\mathsf{Red}(\alpha, l, C_j)))$, in the RHS of test T.

2. Inserted tuple t and relation L are used to compute the tuples in the predicates `ins` and `soln` respectively. These tuples represent the inserted and existing rectangles respectively.

3. Datalog rules are used to generate the tuples of `soln` that represent the basic rectangles obtained by fragmenting the underlying existing rectangles.

4. Datalog rules are used to generate new tuples of `soln` that represent combinations of the basic rectangles.

5. A Datalog rule is used to check if the rectangle represented by the tuple in `ins` is contained in some rectangle represented by a tuple in `soln`.

We prove the correctness of the above steps in Appendix C. In this section we give Datalog rules for the steps in the restricted case when the solution spaces are *closed* rectangles *i.e.*, each rectangle includes its boundary points. Predicates `soln` and `ins` each have 4 attributes (X_l, X_h, Y_l, Y_h). We illustrate the steps using IQC $I6$ in Example 3.1.4.

Step 1 Let parameter μ be $l(u, v, w, z)$ and parameter α be $l(x_l, x_h, y_l, y_h)$. Test T can be obtained using the steps detailed on Page 31 as illustrated in Example 2.7.3 and is of the form:

LHS(T): $u \le X \le v \ \wedge \ w \le Y \le z$.
RHS(T): $x_l \le X \le x_h \ \wedge \ y_l \le Y \le y_h$.

The body of the rule defining `ins` is generated from LHS(T). Predicate `inserted_into_L`$(\bar{U})$ is added to the body as a subgoal where $\bar{U}$ is the set of variables appearing in the rule defining `ins`. Thus, the rule for `ins` is:

`ins`(U, V, W, Z) :– `inserted_into_L`(U, V, W, Z) & $U \le V$ & $W \le Z$.

Similarly, `soln` can be defined by Datalog rules, one rule for each disjunct in RHS(T). For our running example, there is only one disjunct and thus `soln` is defined by a single rule:

`soln`(X_l, X_h, Y_l, Y_h) :– $l(X_l, X_h, Y_l, Y_h)$ & $X_l \le X_h$ & $Y_l \le Y_h$.

The subgoal $l(\bar{X})$ appears in the body where $\bar{X}$ is a set of variables.

Step 2 The instantiation of T by the inserted and existing tuples is achieved by the subgoals `inserted_into_L`$(\bar{U})$ and $l(\bar{X})$ in the rules defining `ins` and `soln` respectively. To begin with, we assume that relation `inserted_into_L` contains a single tuple corresponding to a single inserted tuple. Thus, in the rule defining `ins` the set of variables $\bar{U}$ is instantiated by only this tuple. In contrast, set $\bar{X}$ is instantiated by the tuples in relation L resulting in multiple tuples for `soln`. The `ins` tuple gives a solution space for the inserted tuple and each `soln` tuple gives the solution space for an existing tuple in L.

Step 3 Let $\mathcal{R}$ be the set of rectangular solution spaces corresponding to a particular relation L such that `soln` has a tuple for each rectangle in $\mathcal{R}$. The following Datalog rules compute the `soln` facts for the basic rectangles underlying $\mathcal{R}$. Each rule represents a possible way of fragmenting the rectangle represented by $\texttt{soln}(X_l, X_h, Y_l, Y_h)$.

$$
\begin{array}{lll}
\texttt{soln}(X_l, A, Y_l, Y_h) \text{ :- } & \texttt{soln}(X_l, X_h, Y_l, Y_h) \ \& & \textit{\% Fragment along the first attribute.} \\
 & \texttt{soln}(\_, A, \_, \_) \ \& & \textit{\% Use the upper limit of some other} \\
 & X_l < A < X_h & \textit{\% rectangle as the flexion line.} \\
\texttt{soln}(A, X_h, Y_l, Y_h) \text{ :- } & \texttt{soln}(X_l, X_h, Y_l, Y_h) \ \& & \textit{\% Results in two rectangles.} \\
 & \texttt{soln}(\_, A, \_, \_) \ \& & \\
 & X_l < A < X_h \ . &
\end{array}
$$

Similar rules are defined using the lower limit of some rectangle as a flexion point.
Finally, rules are defined for fragmenting the rectangle along the second attribute.

Step 4 Rules for generating larger rectangles by combining basic rectangles are also defined in Datalog. There is one rule for horizontally combining rectangles and one rule for vertically combining the rectangles.

$$\texttt{soln}(X_l, X_h, Y_l, Y_h) \text{ :- } \texttt{soln}(X_l, A, Y_l, Y_h) \ \& \ \texttt{soln}(B, X_h, Y_l, Y_h) \ \& \ A \geq B.$$
$$\texttt{soln}(X_l, X_h, Y_l, Y_h) \text{ :- } \texttt{soln}(X_l, X_h, Y_l, A) \ \& \ \texttt{soln}(X_l, X_h, B, Y_h) \ \& \ A \geq B.$$

Step 5 The space represented by the inserted tuple is contained in some rectangle represented in relation `soln` if the following rule derives `ins_ok`.

$$\texttt{ins\_ok} \text{ :- } \texttt{ins}(U, V, W, Z) \ \& \ \texttt{soln}(X_l, X_h, Y_l, Y_h) \ \& \ U \geq X_l \ \& \ V \leq X_h \ \& \ W \geq Y_l \ \& \ Z \leq Y_h.$$

If multiple tuples are inserted into relation L, then we can modify the rule defining `ins_ok` by making distinguished the attributes of `ins`. For our example, the rule in step 5 is:

$$
\begin{array}{ll}
\texttt{ins\_ok}(U, V, W, Z) \text{ :- } & \texttt{ins}(U, V, W, Z) \ \& \ \texttt{soln}(X_l, X_h, Y_l, Y_h) \ \& \\
 & U \geq X_l \ \& \ V \leq X_h \ \& \ W \geq Y_l \ \& \ Z \leq Y_h.
\end{array}
$$

Thus, the rule for `ins_ok` in step 5 identifies those inserted rectangles that are contained in existing rectangles. All rectangles that are not in relation `ins_ok` but are in relation `ins` correspond to inserted tuples that are not covered by existing tuples in L.

The above discussion assumes that the solution intervals are always closed. In general, an IQC can use $<$ and $>$ (Definition 3.1.1) in which case the solution intervals may be open and each IQC may yield a union of spaces. The strategy outlined in this section generalizes to open intervals and also works for k-dimensional spaces, thereby covering all IQCs.

3.2 Further Restrictions on Conjunctive Query Constraints

Now, we further restrict the class of CQCs to yield classes for which I_t can be evaluated using unions of conjunctive queries with arithmetic. First we consider the geometric intuition for the existence and nature of such subclasses.

3.2.1 Geometric Intuition

EXAMPLE 3.2.1 Consider the forbidden intervals constraint $I5$. Local checking for $I5$ corresponds to determining if an interval on the number line is contained in unions of other intervals. Thus, we needed to combine existing intervals, via a recursive Datalog rule, and then check if the inserted interval is contained in some combination of existing intervals.

Instead, consider the following constraint that requires that each point in R be greater that all points in L.

`panic` :- $l(X)$ & $r(Y)$ & $X \geq Y$.

Thus tuple $l(a)$ prohibits tuples in R from lying in the closed interval $(-\infty, a]$. Hence, to check if inserted tuple $l(b)$ is covered by tuples in relation L we need to find some existing tuple $l(c)$ such that $c \geq b$. It can also be proved, and is intuitively obvious, that such a check constitutes a complete local test.

Thus, each tuple of L prohibits tuples in R to lie in an interval from $-\infty$ to some constant upper bound. The largest tuple in L defines an interval that has the largest upper bound and thus contains the intervals defined by all other tuples in L. Geometrically, the union of a set S of lines that extend from $-\infty$ to some constant upper bound is the same as the largest line in the set. □

The intuition extends to rectangles also. Consider rectangles both of whose dimensions extend from $-\infty$ to some closed upper bound in the domain of real numbers. We call this class of rectangles (and parallelepipeds) "left-infinite, bounded-right-limit" (LIBRL). For a finite set of LIBRL rectangles, two or more LIBRL rectangles can be combined to yield a new LIBRL rectangle if and only if one of the rectangles actually contains all the other rectangles. We refer to this "can be combined if and only if one contains the others" property as the in-2-in-1 property.

It is the case that a LIBRL parallelepiped R is contained in a finite set of LIBRL parallelepipeds $\bar{R}$ if and only if R is contained in one of the parallelepipeds in $\bar{R}$. The implication of this argument is that the parallelepiped containment problem considered earlier in this chapter becomes simpler because we no longer need rules to combine parallelepipeds. There is no need to fragment and combine parallelepipeds because combining parallelepipeds does not yield a larger parallelepiped. We only need to define parallelepipeds, via conjunctive queries, and we need to check containment of one parallelepiped in another, also via a conjunctive query.

The in-2-in-1 property is preserved for at least three extensions of the class of LIBRL parallelepipeds. One extension allows parallelepipeds to have either open or closed upper bounds. The second extension allows each dimension of the parallelepipeds to go from either $-\infty$ to some closed upper bound, or is equal to a constant. The third extension allows each dimension of the parallelepipeds to go from either $-\infty$ to some open upper bound, or is equal to a constant. Note, if the dimensions of the parallelepipeds are permitted to be either constants, or from $-\infty$ to some open upper bound, or $-\infty$ to some closed upper bound, then the in-2-in-1 property is no longer preserved.

We can also define the class of RIBRL parallelepipeds by considering parallelepipeds whose dimensions extend from constants to $+\infty$. This class can be extended exactly as the class of LIBRL parallelepipeds.

3.2.2 Defining LibCQCs

Now we define classes of constraints for which I_t can be expressed and evaluated by unions of conjunctive queries by ensuring that the solution space for the remote variables defines parallelepipeds that have the in-2-in-1 property. In this section we identify some such classes that are collectively represented by "LibCQC" at the junction of the "Y" in Figure 3.1.

Definition 3.2.1 (**"$\leq$" Variable**) Consider a conjunction of arithmetic comparisons. A variable X is a "$\leq$" variable if all the comparisons that involve X are of the form $X \leq c$ or $X < c$ where c is a constant. □

"$\geq$" variables are defined using $\geq$ and $>$ in place of $\leq$ and $<$.

Definition 3.2.2 (**"$\leq$" Sentence**) Consider a conjunction of arithmetic comparisons that use arithmetic inequalities. The sentence is "$\leq$" if all variables in the sentence are "$\leq$" variables. □

"$\geq$" sentences are defined using $\geq$ in place of $\leq$.

Definition 3.2.3 (**LIBRL Conjunctive Query Constraints (LibCQC)**) Consider CQC C where equated attributes are represented by using the same attribute multiple times, and not by explicit equalities. If $I(\mathsf{Red}(t, l, C))$ is a "$\leq$" sentence, then the query C is a LibCQC. □

Note, the definition of LibCQCs disallows the use equijoins. LibCQCs can be redefined to use equijoins by disallowing the use of one of either $\leq$ or $<$ comparisons. Thus we arrive at following two alternative definitions of LibCQCs both of which have the desired in-2-in-1 property for the rectangles representing the solution space regions for the remote variables. (1) A comparison involving a "$\leq$" variable is either of the form $X \leq c$ or of the form $X = c$. (2) A comparison involving a "$\leq$" variable is either of the form $X < c$ or of the form $X = c$. We refer to all three of these classes as LibCQCs. Below is an example of an LibCQC.

EXAMPLE 3.2.2 Consider constraint $I4$ defined in Example 1.2.4:

$$\texttt{panic} :- \texttt{emp}(E, D, S) \ \& \ \texttt{dept}(D, MS) \ \& \ S > MS.$$

Relation $\texttt{emp}$ is local. Consider $\mathsf{Red}(t, \texttt{emp}, I4)$ for some tuple $t = (e, d, s)$. The resulting conjunctive query is:

$$\texttt{panic} :- \texttt{dept}(D, MS) \ \& \ D = d \ \& \ s > MS.$$

The conjunction of arithmetic subgoals in $\mathsf{Red}(t, \texttt{emp}, I4)$ is referred to as $I(\mathsf{Red}(t, \texttt{emp}, I4))$ (Section 2.7) and this conjunction is a "$\leq$" sentence. Thus the original constraint query $I4$ is an LibCQC. □

RibCQCs and extensions thereof can be defined similar to LibCQCs by using the greater than relationship instead of the less than relationship.

3.2.3 Generating Datalog Rules for LibCQCs

A subset of the steps outlined in Section 3.1.2 are sufficient to generate a union of conjunctive queries that check a LibCQC, using L, when a set of tuples is inserted into relation L. Steps 1, 2, and 5 from Page 40 together generate the necessary queries. Note, by omitting Steps 3 and 4 we avoid fragmenting parallelepipeds and recombining them. Thus, the recursive component of the rules is not incorporated. For Example 3.2.2 the rules are as follows:

EXAMPLE 3.2.3 Let the parameter μ be $\texttt{emp}(e, d, s)$ and parameter α be $\texttt{emp}(x, y, z)$. Parameterized test T is:

$$\mathsf{LHS}(T): \quad d = D \ \wedge \ s > MS.$$
$$\mathsf{RHS}(T): \quad y = D \ \wedge \ z > MS.$$

Using the above test and Steps 1, 2, and 5 outlined on Page 39 we can generate the following conjunctive queries for locally checking constraint $I4$.

$$\texttt{ins}(X_l, X_h, -\infty, Y_h) :- \texttt{inserted\_into\_L}(E, D, S) \ \& \ Y_h = S \ \& \ X_l = D \ \& \ X_h = D.$$
$$\texttt{soln}(X_l, X_h, -\infty, Y_h) :- \texttt{emp}(E, D, S) \ \& \ Y_h = S \ \& \ X_l = D \ \& \ X_h = D.$$

$\texttt{ins\_ok} \text{ :- } \texttt{ins}(A_l, A_h, B_l, B_h)\ \&\ \texttt{soln}(X_l, X_h, Y_l, Y_h)\ \&\ A_l = X_l\ \&\ Y_h \geq B_h.$

□

In the case that **C** has a single LibCQC C and there is only one mapping from $\mathsf{Red}(\alpha, l, C)$ to $\mathsf{Red}(\mu, l, C)$, the conjunctive queries used to express the local test for checking C can actually be folded into a single conjunctive query. The above example is an instance of such a query. Intuitively, the folding can be done because of the structure of the parameterized test T for constraints that satisfy the given restrictions. Recall from step 1 on Page 40 that for each disjunct in $\mathsf{RHS}(T)$, one rule is generated for defining `soln`. If there is only one constraint and only one mapping from $\mathsf{Red}(\alpha, l, C)$ to $\mathsf{Red}(\mu, l, C)$, then $\mathsf{RHS}(T)$ has only one disjunct and thus only one rule defining `soln`. Thus, the definitions of predicates `ins` and `soln` can be folded into the rule defining `ins_ok`, thereby resulting in a complete local query that is expressible as a single CQ.

3.3 Complexity Issues

In this section we discuss the complexity of local checking for conjunctive query constraints that use arithmetic inequalities. We comment on the complexity of evaluating I_t as produced in Chapter 2 for the restricted classes LibCQC and IQC, and also for general CQCs.

3.3.1 LibCQC

Consider LibCQCs as defined by Definition 3.2.3. For a LibCQC C_0, the solution space to $\mathsf{Red}(t, l, C_0)$ is a k-dimensional parallelepiped all of whose sides extend to $-\infty$ and are either closed or open at the upper bound. Thus $\mathsf{Red}(t, l, C_0)$ corresponds to a *left-semiinterval conjunctive query (LSCQ)* as defined by Definition A.3.1 in Appendix A. For LSCQs we prove that a LSCQ C is contained in a set of LSCQs S if an only if C is contained in one of the LSCQs in S. Thus, for LibCQCs we can observe that $\mathsf{Red}(t, l, C_0)$ is contained in the union $\cup_{C_j\ in\ \mathbf{C},\ t_i\ in\ L}(\mathsf{Red}(t_i, l, C_j))$ if and only if $\mathsf{Red}(t, l, C_0)$ is contained in some $\mathsf{Red}(t_i, l, C_j)$.

Using the above result the parametric test T can be fragmented into $\Sigma_{j=0}^{m}|\mathcal{M}_j|$ select queries on L, where $|\mathcal{M}_j|$ is the number of mappings from $\mathsf{Red}(\alpha, l, C_j)$ to $\mathsf{Red}(\mu, l, C_0)$. T evaluates to true given t and L if and only if any one of these queries computes a nonempty answer. The total number of queries could be $m * (p^p)$, where p is the maximum number of ordinary predicates in any C_i [Sar90] and m is the number of constraints in set **C**. Each of these queries can be evaluated in time less than $O(|L|)$ time by utilizing indexes on relation L. In terms of the Datalog program derived in Section 3.2.3, $m * (p^p)$ rules are used define the `soln` predicate.

3.3.2 IQC

IQCs are formally defined in Definition 3.1.3. Again the question is how to verify I_t. As pointed out before, the solutions to the variables in $I(q_i)$ are in a k-dimensional parallelepiped and checking I_t is the same as checking containment of a k-dimensional parallelepiped in a union of k-dimensional parallelepipeds. The complexity of the technique described in Section 3.1.1 depends on how much information is maintained about the parallelepipeds. If the basic parallelepipeds are stored then it is possible to determine containment in time $O(k * lg(|L|))$. However the complexity of maintaining the basic parallelepipeds is much higher, as high as $O(|L|^{2k})$ time. If the Datalog rules executed from scratch, and generated the basic rectangles, then too the complexity of checking containment would be $O(|L|^{2k})$.

More efficient methods can be used to determine the parallelepiped containment. However, these methods are not representable in Datalog. If k-dimensional interval trees are used, then containment can be determined in $O(lg^k(|L|))$ time [R93]. The interval tree itself can be constructed a priori in $O(|L|^k)$ time. The use of interval trees is an instance of a more efficient non-query-based alternative to a not-so-efficient query-based method.

3.3.3 General CQCs

A CQC is not an IQC if $\mathsf{Red}(t, l, C)$ includes a comparison X $\mathtt{op}$ Y where both X and Y are variables. In this case, determining the truth of I_t is NP-hard [SKN89]. A naive strategy can be implemented in $O(|L|^k)$ time, based on an extension of r-structures described in [KKR93]. An interval-tree-like storage structure can be used in this case too, by appending each node in the interval tree with additional bits that keep track of the relative order of the variables. We conjecture that the complexity is the same as when interval-trees are used for IQC, *i.e.* $O(lg^k(|L|))$.

In general, the cost of evaluating I_t depends on the kind of arithmetic comparisons allowed in C. If the most powerful comparisons are of the form X $\mathtt{op}$ $Y+c$ or X $\mathtt{op}$ $Y-c$, then the implication problem is decidable in exponential time. However, if the attributes can be added together and compared to another attribute, then verifying I_t may be undecidable.

3.4 Arbitrary Interpreted Predicates

The containment results of Appendix A apply to conjunctive queries that use arbitrary interpreted predicates for which there is a theory to solve implication sentences of the form described in Theorem A.2.2. Therefore, complete local test conditions can be derived for corresponding conjunctive query constraints of such classes.

Chapter 4

More General First Order Constraints

In this chapter we consider local checking for constraint specification languages more expressive than conjunctive queries. For instance, referential integrity constraint $I1$ (Page 1) cannot be expressed as conjunctive query constraints because negation is needed. All the languages considered in this chapter are more expressive than conjunctive query constraints. Many of the constraints that we have encountered in our example domain [GT93, GT94, TH93] require the additional expressive power provided by the language of this chapter. For the classes we consider, we develop algorithms that generate only *sufficient* test conditions for these languages, *i.e.*, tests that are not complete. Even for conjunctive query constraints for which complete methods exist, the techniques developed in this chapter may derive tests that are not complete. We will trade off completeness for the ability to derive sufficient tests for more expressive languages. Thus, in this chapter we explore the right branch of the "Y." In particular, we consider the three circled classes shown in Figure 4.1. All the tests derived in this chapter are expressible as CQs or as unions of CQs.

We start at the lower most point in the right branch of the "Y" and consider progressively more expressive constraint languages. Unlike conjunctive query constraints where deletions can never violate a constraint – because all subgoals were positive – in this chapter deletions are relevant because the languages of this chapter allows negated subgoals. Also, the methods of this chapter do not use multiple constraints to check another constraint. Thus, we assume that the set of constraints $\mathbf{C}$ contains only one constraint.

Chapter Outline

Section 4.1 describes the first constraint language considered in this chapter and explains how the language relates to conjunctive query constraints considered earlier. Section 4.2 then describes the actual test condition for this more expressive constraint language and illustrates the test condition

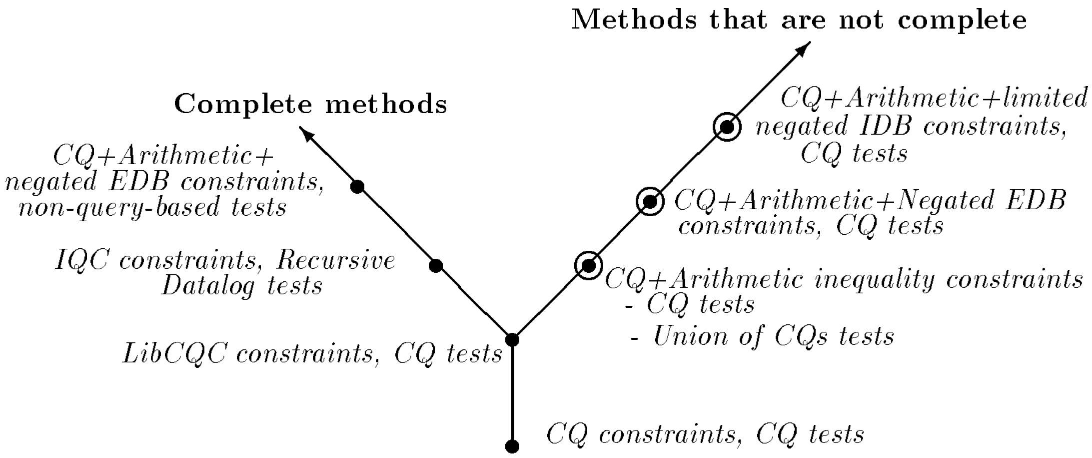

Figure 4.1: Constraint Classes Considered in Chapter 4

via examples. Section 4.3 discusses evaluation issues for the test condition. Section 4.4 relates the test to the containment mappings techniques used in Chapter 3. This section also discusses some generalizations of the test condition and discusses completeness of the test condition. Section 4.5 further extends the expressive power of the constraint specification language and describes how to generate tests for these constraints.

4.1 Language

In this chapter we express constraints using a subset of first order logic (FOL). Unlike the previous chapters where a constraint was expressed as its violation condition, in this chapter a constraint will be expressed as an *assertion*. As assertion is the condition that a database needs to satisfy in order for the constraint to hold. For instance, the integrity constraint assertion for the forbidden interval constraint $I5$ from Example 2.5.2 on Page 24 is:

$$\forall B, E, P : [l(B, E) \wedge r(P) \quad \Rightarrow \quad (B \geq P \vee P \geq E)].$$

That is, constraint $I5$ holds if for every point in relation R and every interval in relation L, either the point is less than the lower limit of the interval or greater than the upper limit of the interval. An assertion can be converted into a (set of) violation condition(s) by negating the assertion. The class of assertions that are considered in this chapter result in violation conditions that cannot always be expressed as conjunctive query constraints.

4.1.1 Syntax

The simplest integrity constraint assertion language we consider captures constraints that require that if the database satisfies some conditions, then the database should also satisfy some other condition. The language considered in this section requires the implied condition to be the existence of a tuple in a relation. For instance, the language can express the assertion that if an employee is in relation `emp` then the department of the employee should exist in relation `dept`. We start with a simple assertion language in order to highlight the essential intuition for the test conditions developed in this chapter. In general, the implied condition may be a complicated condition involving one or more relations in the database. In Section 4.5 we consider such extensions of the simple assertion language stated below.

The assertion language is similar to that used in [CG92] and represents the first two circled dots on the right branch of the "Y" in Figure 4.1. Each assertion is of the form:

$$C: \quad \forall \bar{X}\, \forall \bar{Y}\, \exists \bar{Z} : [(L(\bar{X}) \wedge R_1(\bar{Y}_1) \wedge \ldots \wedge R_k(\bar{Y}_k) \wedge g(\bar{X},\bar{Y},\bar{Z},\bar{c})) \Rightarrow (S_1(\bar{Y}'_1,\bar{Z}'_1) \vee \ldots \vee S_n(\bar{Y}'_n,\bar{Z}'_n))]$$

where:

L represents the accessible local relation.

$R_1,\ldots,R_k,S_1,\ldots,S_n$ represent inaccessible remote relations.

$\bar{X}$ is a set of universally quantified ($\forall$) variables occurring only in L and g.

$\bar{Y}$ is a set of $\forall$ variables occurring only in $R_1,\ldots,R_k,S_1,\ldots,S_n$, and g.

$\bar{Z}$ is a set of $\exists$ variables occurring only in $S_1,\ldots,S_n$, and g.

$\bar{c}$ is a set of constants occurring only in g.

$g(\bar{X},\bar{Y},\bar{Z},\bar{c})$ is a conjunction of equalities ($=$) and inequalities ($<, >, \leq, \geq$).

$\bar{Y}_i \subseteq \bar{Y}$ is the set of $\forall$ variables that occur in relation $R_i, 1 \leq i \leq k$.

$\bar{Y}'_i \subseteq \bar{Y}$ is the set of $\forall$ variables that occur in relation $S_i, 1 \leq i \leq n$.

$\bar{Z}'_i \subseteq \bar{Z}$ is the set of $\exists$ variables that occur in relation $S_i, 1 \leq i \leq n$.

No variable occurs in more than one relation. Constants do not occur in relations.

We often refer to the conjunction $g(\bar{X},\bar{Y},\bar{Z},\bar{c})$ as just g. The integrity constraint assertion must also satisfy some restrictions in order to be *evaluable* [F82, VT91]. The restrictions on the integrity constraint assertions are:

1. Existentially quantified variables can occur only in relations on the right hand side (RHS) of the implication in the integrity constraint assertion. Those existentially quantified variables that occur in g must be equal to a constant or to a universally quantified variable.
2. Universally quantified variables occurring in a relation on the RHS of the assertion must also either occur in some relation on the left hand side (LHS), be equal to a variable that occurs in a relation on the LHS, or be equal to a constant.

In the above restrictions, equality is transitively closed.

Examples

First, we express the referential integrity constraint $I1$ from Example 1.0.1 using the above notation and then we consider constraint $I4$ from Example 1.2.4.

EXAMPLE 4.1.1 Constraint $I1$ requires that every department referenced in the `emp` relation also exist in the `dept` relation.

$$\forall E, D, S \exists D', MS : [(\texttt{emp}(E, D, S) \wedge D' = D) \Rightarrow \texttt{dept}(D', MS)]$$

Constraint $I1$ is violated if there is an employee in a department that does not occur in any tuple in the `dept` relation. We repeat the violation condition for the above constraint from Page 3.

r_1: $\texttt{all\_depts}(D)$:– $\texttt{dept}(D, MS)$.
r_2: `panic` :– $\texttt{emp}(E, D, S)$ & *not* $\texttt{all\_depts}(D)$.

Constraint $I1$ can be expressed as an assertion in the language of this chapter, but the violation condition for $I1$ cannot be expressed using a conjunctive query. Instead the violation condition requires a Datalog program with negation, as above. □

Now consider a constraint assertion that uses arithmetic.

EXAMPLE 4.1.2 Integrity constraint $I4$ on the employee-department database requires that the salary of each employee should be no more than the salary of any manager in the same department as the employee.

$$\forall E, D, S, D', MS : [(\texttt{emp}(E, D, S) \wedge \texttt{dept}(D', MS) \wedge D' = D) \Rightarrow MS \geq S]$$

The above constraint is not in the prescribed form because the RHS uses an arithmetic inequality. In the prescribed form, the above constraint is represented as:

$$\forall E, D, S, D', MS : [(\texttt{emp}(E, D, S) \wedge \texttt{dept}(D', MS) \wedge D' = D \wedge S > MS) \Rightarrow \texttt{false}]$$

Constraint $I4$ is violated if there is at least one employee whose salary is greater than the salary of some manager in the same department. That is, $I4$ is violated when the following query derives `panic`:

`panic` :– $\texttt{emp}(E, D, S)$ & $\texttt{dept}(D, MS)$ & $S > MS$.

Constraint $I4$ can be handled by the techniques of Chapter 3 because its violation condition is expressible as a conjunctive query with arithmetic inequalities. In fact, Example 3.2.3 states a complete check for this constraint. Thus, we use $I4$ to illustrate how the results of this chapter compare with results of the previous chapters. □

4.2 Deriving Test Conditions

This section presents the test condition on the accessible relation L, derived using the constraint specification C and assuming that one tuple μ is inserted into the accessible relation. We also illustrate the test condition using our examples.

4.2.1 Intuition

The test condition for locally checking constraint C is derived using the intuition of Figure 2.2 on Page 21. We would like to prove that the set of illegal remote database states corresponding to the inserted tuple μ, is contained in the union of the sets of illegal remote database states corresponding to existing tuples in L. The initial consistency assumption then allows us to infer that inserting μ does not violate C because no existing tuple in L violates C.

Consider an assertion C of the form described in Section 4.1 and a tuple μ inserted into L. All illegal remote database states corresponding to tuple μ make true the LHS of the assertion, but do not make the RHS true. Thus, we attempt to prove that whenever the LHS of assertion C is true with μ, then the LHS of assertion C is true with some existing tuple in L. If we succeed, then we are guaranteed that if C is violated with μ, then C is violated with some existing tuple in L. The implication involving the LHS of the assertion can be reduced to an implication involving just the arithmetic inequalities as will be argued in the proof of Theorem 4.5.1. Thus the test condition turns out to be an implication involving arithmetic inequalities.

The next section formally states the test condition.

4.2.2 Test Condition

Definition 4.2.1 ($\mathsf{TC}(C, l, \mu)$) Consider an integrity constraint C:

$$C\!: \quad \forall \bar{X}\, \forall \bar{Y}\, \exists \bar{Z} : [(L(\bar{X}) \,\wedge\, R_1(\bar{Y}_1) \,\wedge \ldots \wedge\, R_k(\bar{Y}_k) \,\wedge\, g(\bar{X}, \bar{Y}, \bar{Z}, \bar{c})) \;\Rightarrow\; (S_1(\bar{Y}'_1, \bar{Z}'_1) \,\vee \ldots \vee\, S_n(\bar{Y}'_n, \bar{Z}'_n))]$$

Let μ be a tuple inserted into the accessible relation L. The test condition $\mathsf{TC}(C, l, \mu)$ is:

$$\mathsf{TC}(C, l, \mu)\!: \quad \exists \bar{X}\, \forall \bar{Y}\, \forall \bar{Z} : [L(\bar{X}) \,\wedge\, (g(\mu, \bar{Y}, \bar{Z}, \bar{c}) \;\Rightarrow\; g(\bar{X}, \bar{Y}, \bar{Z}, \bar{c}))]$$

□

Note only relation L is involved in $\mathsf{TC}(C, l, \mu)$, and the test does not refer to any of the inaccessible relations $R_1, \ldots, R_k, S_1, \ldots, S_n$.

The above test is derived at compile time by treating μ as a parameter instead of using the actual value for the inserted tuple. When a tuple is actually inserted into relation L at run time, $\mathsf{TC}(C, l, \mu)$ is instantiated by the inserted tuple and evaluated using the accessible relation. The following theorem proves the correctness of Definition 4.2.1.

Theorem 4.2.1 *Consider an integrity constraint C and a tuple μ inserted into relation L. If the database satisfies integrity constraint assertion C before adding tuple μ, and if the test condition $\mathsf{TC}(C, l, \mu)$ is satisfied by the accessible relation L, then the database satisfies integrity constraint assertion C after inserting tuple μ.* □

Proof: Theorem 4.2.1 is a special case of Theorem 4.5.1 stated in Section 4.5 on page 61. ∎

Test condition $\mathsf{TC}(C, l, \mu)$ requires solving an implication involving the arithmetic inequalities in g. This implication is similar to the arithmetic implication I_t produced as a result of instantiating parameterized test T in Section 2.7.3. The implication in $\mathsf{TC}(C, l, \mu)$ does not have disjuncts in its RHS, making it a restricted case of implication I_t. We explain this difference in more detail in Section 4.4. In Section 4.3 we discuss how the implication of $\mathsf{TC}(C, l, \mu)$ is evaluated.

Note, $R_1, \ldots, R_k$, $S_1, \ldots, S_n$ are not restricted to be base relations. If some inaccessible relation R_i uses L in its definition, or is a repeat occurrence of L, then updating L also causes an update to R_i. In this case we would need two tests to be executed, one that checks the update to L and the other that checks the update to R_i. For each test, all remaining relations are treated as being remote. The correctness of executing multiple tests is discussed in Section 5.2.

Examples

EXAMPLE 4.2.1 Consider the referential integrity constraint $I1$ from Example 4.1.1.

$$\forall E, D, S\, \exists D', MS : [(\mathtt{emp}(E, D, S) \wedge D'{=}D) \;\Rightarrow\; \mathtt{dept}(D', MS)]$$

The accessible relation L is the `emp` relation, g is $D = D'$, and the inserted tuple parameter is (e, d, s). The test according to Definition 4.2.1 is:

$$\mathcal{A}:\quad \exists E, D, S\, \forall D', MS : [\mathtt{emp}(E, D, S) \wedge ((D'{=}d) \quad\Rightarrow\quad (D'{=}D))]$$

Suppose tuple $\mathtt{emp}(john, toy, 50)$ is inserted. The parameterized variable d is replaced by the actual department from the inserted tuple, namely toy, resulting in the test:

$$\exists E, D, S\ \forall D', MS : [\mathtt{emp}(E, D, S) \wedge ((D'{=}toy) \quad\Rightarrow\quad (D'{=}D))]$$

This condition is further simplified in Example 4.3.2. □

EXAMPLE 4.2.2 Consider the integrity constraint $I4$ as stated in Example 4.1.2.

$$\forall E, D, S, D', MS : [(\mathtt{emp}(E, D, S) \wedge \mathtt{dept}(D', MS) \wedge D'{=}D \wedge S \geq MS) \;\Rightarrow\; \mathtt{false}]$$

L is again `emp`, g is $D'{=}D \wedge S \geq MS$, and the inserted tuple parameter is (e, d, s). The test according to Definition 4.2.1 is:

$$\mathcal{B}:\quad \exists E, D, S\ \forall D', MS : [\mathtt{emp}(E, D, S) \wedge ((D'{=}d \wedge s \geq MS) \quad\Rightarrow\quad (D'{=}D \wedge S \geq MS))]$$

When tuple $\mathtt{emp}(john, toy, 50)$ is inserted, the parameter d is replaced by the department from the inserted tuple: toy, and parameter s is replaced by the salary from the inserted tuple: 50. The resulting test is:

$$\exists E, D, S\ \forall D', MS : [\mathtt{emp}(E, D, S) \wedge ((D'{=}toy \wedge 50{\geq}MS) \quad\Rightarrow\quad (D'{=}D \wedge S{\geq}MS))]$$

This condition is further simplified in Example 4.3.3. □

4.3 Evaluating the Test Condition

Test $\mathsf{TC}(C, l, \mu)$ as defined in the previous section produces an implication involving g, where g is a conjunction of arithmetic predicates. If the relation L contains a tuple that satisfies this implication then we say that the relation satisfies the test. There are two approaches to solving this test condition, *i.e.*, for checking if $\mathsf{TC}(C, l, \mu)$ is true given L.

1. Instantiate the test condition with every tuple in L, resulting in a set of arithmetic implications each of which involves constants and the universally quantified variables $\bar{Y}$ and $\bar{Z}$. That is, $\bar{X}$ in the implication $(g(\mu, \bar{Y}, \bar{Z}, \bar{c}) \Rightarrow g(\bar{X}, \bar{Y}, \bar{Z}, \bar{c}))$ is instantiated with every tuple in L, and each instantiation yields one implication. If any one of these implications is true, the test condition is also true.

   EXAMPLE 4.3.1 Consider the test derived in Example 4.2.2 when $(john, toy, 50)$ is inserted into `emp`.

   $$\exists E, D, S\ \forall D', MS : [\mathtt{emp}(E, D, S) \wedge ((D'{=}toy \wedge 50{\geq}MS) \quad\Rightarrow\quad (D'{=}D \wedge S{\geq}MS))]$$

   Let relation `emp` have two tuples $(mary, toy, 100)$ and $(bob, shoes, 200)$. After factoring in the extent of the accessible relation `emp` the test condition can be rewritten as:

   $$\forall D', MS : [(D'{=}toy \wedge 50{\geq}MS) \quad\Rightarrow\quad (D'{=}toy \wedge 100{\geq}MS)] \quad\vee$$
   $$\forall D', MS : [(D'{=}toy \wedge 50{\geq}MS) \quad\Rightarrow\quad (D'{=}shoes \wedge 200{\geq}MS)]$$

   □

   The implications produced by the above approach can be evaluated by the methods discussed in Section 3.3. There we considered how to evaluate the implication I_t of the form $A \Rightarrow B_1 \vee B_2 \vee \ldots \vee B_k$ where each of $A, B_1, \ldots, B_k$ is a conjunction of arithmetic inequalities. In contrast, test condition $\mathsf{TC}(C, l, \mu)$ generates a disjunction of implications, each of the form $A \Rightarrow B$. It is less expensive to verify each such disjunct than it is to verify I_t. Geometrically, checking $\mathsf{TC}(C, l, \mu)$ involves determining if a parallelepiped is contained in one of a set of parallelepipeds whereas checking I_t involves determining if a parallelepiped is contained in the union of a set of parallelepipeds.

2. Eliminate the universally quantified inaccessible variables in $\bar{Y}$ and $\bar{Z}$ from the implication $(g(\mu, \bar{Y}, \bar{Z}, \bar{c}) \Rightarrow g(\bar{X}, \bar{Y}, \bar{Z}, \bar{c}))$ in the test condition $\mathsf{TC}(C, l, \mu)$. Eliminating the variables in $\bar{Y}$ and $\bar{Z}$ results in a set of restrictions $\mathcal{I}$ on the variables in $\bar{X}$ that serve as a selection condition, or query, on the accessible relation L. This selection condition is a sufficient test, *i.e.*, if some

tuple $l(\bar{X})$ in L satisfies condition $\mathcal{I}$, then the implication $(g(\mu, \bar{Y}, \bar{Z}, \bar{c}) \Rightarrow g(\bar{X}, \bar{Y}, \bar{Z}, \bar{c}))$ is also true. Appendix B describes how to eliminate the sets of variables $\bar{Y}$ and $\bar{Z}$ to obtain $\mathcal{I}$.

EXAMPLE 4.3.2 Consider the test condition $\mathcal{A}$ from Example 4.2.1.

$$\mathcal{A}: \quad \exists E, D, S \forall D', MS : [\texttt{emp}(E, D, S) \wedge ((D' = d) \quad \Rightarrow \quad (D' = D))]$$

The universally quantified variables D', MS can be eliminated from $\mathcal{A}$ to give a condition $\mathcal{I}_A$ that involves only variables of the accessible relation `emp`. MS can be eliminated from $\mathcal{A}$ simply by deleting MS from $\mathcal{A}$. Variable D' is eliminated by propagating the equality $D' = d$ as explained in Appendix B. If $\mathcal{I}_A$ is satisfiable then we are guaranteed that the test $\mathcal{A}$ is true for all variables D', MS.

$$\mathcal{I}_A: \quad \exists E, D, S : [\texttt{emp}(E, D, S) \wedge D = d]$$

Condition $\mathcal{I}_A$ is evaluated by querying relation `emp` for a tuple that has d as its department. Thus if tuple $(john, toy, 50)$ is inserted into `emp` then the test query looks for an existing tuple in `emp` that has attribute $D = toy$. □

EXAMPLE 4.3.3 Consider test condition $\mathcal{B}$ from Example 4.2.2.

$$\exists E, D, S \ \forall D', MS : [\texttt{emp}(E, D, S) \wedge ((D' = d \wedge s \geq MS) \quad \Rightarrow \quad (D' = D \wedge S \geq MS))]$$

The universally quantified variables D' and MS are eliminated from the test condition to give a condition involving only the variables of the accessible relation `emp`:

$$\mathcal{I}_B: \quad \exists E, D, S : [\texttt{emp}(E, D, S) \wedge D = d \wedge S \geq s]$$

This condition is evaluated by performing a query on the `emp` relation for a tuple that has d as its department and $S \geq s$ in the salary field. If tuple $(john, toy, 50)$ is inserted into `emp` then the test query looks for an existing tuple in `emp` that has attribute $D = toy$ and $S \geq 50$.
□

If g contains just equality predicates, then eliminating the universally quantified variables is simply a matter of propagating equalities. The cost of doing the propagation is linear in the number of equality expressions in g. With arithmetic comparisons the complexity of simplifying the implication is $O(n^3)$, where n is the number of inequalities in g [Dav87].

Approach 1 is more powerful, but it is not query-based. If we allowed the use of inequalities of the form $X \ op \ Y + c$ or $X \ op \ Y - c$ in g, then the implication is solvable in exponential time using approach 1. By contrast, a restricted form of the above sentences can be handled using approach 2 and the techniques of Appendix B. That is, universally quantified variables can be eliminated in limited cases when $X \ op \ Y + c$ or $X \ op \ Y - c$ are used. In particular, if both X and Y are not remote variables, then the method of Appendix B applies. Intuitively, such inequalities introduce limits of the form $a - c$ for remote variables, where a is a parameter. Such limits do not cause any problem in determining the parameterized bounds for remote variables. However, if both X and Y are remote variables then it is not possible to determine the tightest parameterized bounds for remote variables using the method of Appendix B. If other more sophisticated methods for eliminating universally quantified variables exist, then technique 2 will be applicable to a wider class of constraints.

In general, the applicability of test $\mathsf{TC}(C, l, \mu)$ is restricted by the evaluability of the implication based on g but the correctness of $\mathsf{TC}(C, l, \mu)$ does not depend on the structure of g. The test condition is applicable to all theories that can solve the implication condition involving g.

4.4 Generalizations and Completeness of $\mathsf{TC}(C, l, \mu)$

In this section we study three possible generalizations of the test condition $\mathsf{TC}(C, l, \mu)$ to produce stronger tests than $\mathsf{TC}(C, l, \mu)$. The generalizations are motivated by comparing the test conditions generated by the containment based algorithms of Chapter 3 with $\mathsf{TC}(C, l, \mu)$. The comparison gives the intuition for why the generalizations make sense and what the cost associated with them is. Finally, we also consider a restricted class of constraints for which $\mathsf{TC}(C, l, \mu)$ is a complete local test.

4.4.1 Change the Order of Quantified Variables $\bar{X}$, $\bar{Y}$, and $\bar{Z}$

Recall that $\mathsf{TC}(C, l, \mu)$ is of the form:

$$\mathsf{TC}(C, l, \mu):\quad \exists \bar{X}\, \forall \bar{Y}\, \forall \bar{Z} : [L(\bar{X}) \wedge (g(\mu, \bar{Y}, \bar{Z}, \bar{c}) \quad \Rightarrow \quad g(\bar{X}, \bar{Y}, \bar{Z}, \bar{c}))]$$

$\mathsf{TC}(C, l, \mu)$ looks for a single cover tuple. The single cover tuple based nature of $\mathsf{TC}(C, l, \mu)$ arises because $\exists \bar{X}$ is the outermost member of the list of quantified variables and thus forces that one instantiation of $\bar{X}$ suffice to cover μ for all values of variables $\bar{Y}$ and $\bar{Z}$. Thus, $\mathsf{TC}(C, l, \mu)$ would not always succeed in checking the forbidden interval constraint $I5$, when multiple tuples together may be needed to cover the inserted tuple.

Now, consider an alternative form of $\mathsf{TC}(C, l, \mu)$ obtained by moving $\exists \bar{X}$ to the end of the quantified list. This test can be proved correct by slightly varying Step 2 of the proof of Theorem 4.5.1.

$$\mathsf{TC}(C, l, \mu)_{\cup g}:\quad \forall \bar{Y}\, \forall \bar{Z}\, \exists \bar{X} : [L(\bar{X}) \wedge (g(\mu, \bar{Y}, \bar{Z}, \bar{c}) \quad \Rightarrow \quad g(\bar{X}, \bar{Y}, \bar{Z}, \bar{c}))]$$

Unlike test $\mathsf{TC}(C, l, \mu)$ that looks for one existing tuple that covers the inserted tuple for all values of the variables $\bar{Y}$ and $\bar{Z}$, $\mathsf{TC}(C, l, \mu)_{\cup g}$ can use a different tuple in L for every value taken by variables $\bar{Y}$ and $\bar{Z}$. Thus, $\mathsf{TC}(C, l, \mu)_{\cup g}$ uses multiple cover tuples.

The above test condition cannot be manipulated by the techniques of Appendix B. The reason is that the universally quantified variables cannot be eliminated if the universal quantifier occurs *before* the existential quantifier in the test condition. However, if $g(\mu, \bar{Y}, \bar{Z}, \bar{c})$ is restricted to an Independently Constrained Sentence (Definition 3.1.2) or a "$\leq$" sentence (Definition 3.2.2), then we can use exactly the techniques of Sections 3.1 and 3.2 to convert the implication in the above test condition to a recursive Datalog program or a union of conjunctive queries respectively.

Using the above intuition we can also explicitly factor the use of two cover tuples into test $\mathsf{TC}(C, l, \mu)$ as follows:

$$\mathsf{TC}(C, l, \mu)_2\text{:} \quad \exists \bar{X}_1 \exists \bar{X}_2 \forall \bar{Y} \forall \bar{Z} : [L(\bar{X}_1) \wedge L(\bar{X}_2) \wedge (g(\mu, \bar{Y}, \bar{Z}, \bar{c}) \;\; \Rightarrow \\ \{g(\bar{X}_1, \bar{Y}, \bar{Z}, \bar{c}) \vee g(\bar{X}_2, \bar{Y}, \bar{Z}, \bar{c})\})]$$

In general, we could write explicitly a test that uses any fixed number of cover tuples. Note, the above form involves an arithmetic implication with disjunction on its RHS similar to implication I_t discussed on Page 31. If the RHS of the test involves disjunctions then the universally quantified variables $\bar{Y}$ and $\bar{Z}$ cannot be eliminated from the test using the techniques of Appendix B. Again, if $g(\mu, \bar{Y}, \bar{Z}, \bar{c})$ is restricted to be an independently constrained sentence or a "$\leq$" sentence then the results of Sections 3.1 and 3.2 apply.

In this chapter we focus on tests that use single cover tuples, because it is simpler to understand the derivation of test conditions that use single cover tuples. For all the classes we discuss, the extension to multiple tuples (fixed or arbitrary) can be done as described above.

4.4.2 Change the Position of the Conjunct $L(\bar{X})$

Recall that $\mathsf{TC}(C, l, \mu)$ is of the form:

$$\mathsf{TC}(C, l, \mu)\text{:} \quad \exists \bar{X} \forall \bar{Y} \forall \bar{Z} : [L(\bar{X}) \wedge (g(\mu, \bar{Y}, \bar{Z}, \bar{c}) \quad \Rightarrow \quad g(\bar{X}, \bar{Y}, \bar{Z}, \bar{c}))]$$

The conjunct $L(\bar{X})$ restricts the test condition by not being a part of the implication. We highlight this point using an extension of Example 4.1.2.

EXAMPLE 4.4.1 Consider constraint $I4'$ requiring that if an employee's salary is greater than 100 then the salary should not be greater than the salary of any manager in the same department. $I4'$ is violated if the following conjunctive query derives `panic`.

$$\texttt{panic} :\!-\ \texttt{emp}(E, D, S)\ \&\ \texttt{dept}(D', MS)\ \&\ S > 100\ \&\ S > MS\ \&\ D = D'.$$

If tuple $\texttt{emp}(john, toy, 50)$ is inserted, constraint $I4'$ need not be checked because the subgoal $50 > 100$ cannot be satisfied. However, consider $\mathsf{TC}(I4', l, \mu)$ for the above constraint:

$$\exists E, D, S\ \forall D', MS : [\texttt{emp}(E, D, S) \wedge ((D' = d \wedge s \geq MS \wedge s > 100) \quad \Rightarrow \\ (D' = D \wedge S > MS \wedge S > 100))].$$

If the relation emp is empty when $\texttt{emp}(john, toy, 50)$ is inserted, then $\mathsf{TC}(I4', l, \mu)$ does not evaluate to true because of the conjunct $\texttt{emp}(E, D, S)$ that requires that there be at least one tuple in emp even though the implication is vacuously true. □

The problem arises because if $g(\mu, \bar{Y}, \bar{Z}, \bar{c})$ is false then the inserted tuple cannot violate the constraint and thus a cover tuple need not exist. However, L may be empty and thus cause the test as a whole to fail. This shortcoming can be rectified by including $L(\bar{X})$ in the LHS of the implication. In all subsequent test conditions this optimization is made. For the above example, the optimization results in the following test condition.

$$\exists E, D, S\ \forall D', MS : [(\texttt{emp}(E, D, S) \wedge D' = d \wedge s \geq MS \wedge s > 100) \Rightarrow (D' = D \wedge S > MS \wedge S > 100)].$$

We introduced $\mathsf{TC}(C, l, \mu)$ first because excluding $L(\bar{X})$ from the implication simplifies the implication condition and facilitates understanding the manipulation needed to convert the implication to a sufficient condition on the variables $\bar{X}$.[1]

4.4.3 Use Multiple Symbol Mappings

EXAMPLE 4.4.2 Consider two relations from the database for a transportation company.

$\texttt{bus\_from}(S, D)$	% *Source S to destination D are connected by a bus*
$\texttt{depot}(A)$	% *A is a depot.*

Consider an integrity constraint $I7$ on this database that asserts that if there is a bus from source S to destination D, then S and D cannot both be depots. The violation condition for this constraint can be represented by the following CQ.

$$\texttt{panic} :\!- \texttt{bus\_from}(S, D)\ \&\ \texttt{depot}(S)\ \&\ \texttt{depot}(D).$$

Let relation bus_from be local, let the parameterized tuple $\mu = (a, b)$ be inserted into the relation, and let the parameterized tuple $\alpha = (u, v)$ represent existing tuples of bus_from.

First we use the techniques of Chapter 3 to derive the parameterized local test condition. The process is as outlined on Page 31. First, we compute $\mathsf{Red}(\alpha, l, I7)$ and $\mathsf{Red}(\mu, l, I7)$.

$\mathsf{Red}(\alpha, l, I7)$: $\texttt{panic} :\!- \texttt{depot}(S)\ \&\ \texttt{depot}(D)\ \&\ S = u\ \&\ D = v.$

$\mathsf{Red}(\mu, l, I7)$: $\texttt{panic} :\!- \texttt{depot}(S)\ \&\ \texttt{depot}(D)\ \&\ S = a\ \&\ D = b.$

There are four symbol mappings from $\mathsf{Red}(\alpha, l, I7)$ to $\mathsf{Red}(\mu, l, I7)$.

$h1$: $S \to S,\ D \to D.$

$h2$: $S \to S,\ D \to S.$

[1]The optimized version of $\mathsf{TC}(C, l, \mu)$ detects irrelevant updates whereas the unoptimized version does not detect irrelevant updates. Irrelevant updates are studied in [BCL89, Elk90, LS93].

$h3$: $S \to D,\ D \to D.$
$h4$: $S \to D,\ D \to S.$

Using the above mappings, the parameterized test condition T is:

$\mathsf{LHS}(T)$: $(S = a, D = b)$.
$\mathsf{RHS}(T)$: $(S = u, D = v) \vee (S = u, D = u) \vee (S = v, D = v) \vee (S = v, D = u)$.

Now let us consider the strategy developed in this chapter and obtain $\mathsf{TC}(I7, L, \mu)$. First, we need to state constraint $I7$ as an assertion:

$$\forall S, D, A, B : [(\texttt{bus\_from}(S, D) \wedge \texttt{depot}(A) \wedge \texttt{depot}(B) \wedge A{=}S \wedge D{=}B) \ \Rightarrow\ \texttt{false}].$$

The test generated according to Definition 4.2.1 is:

(b): $\exists U, V\, \forall S, D : [\texttt{bus\_from}(U, V) \wedge ((S = a, D = b) \Rightarrow (S = U, D = V))]$.

Test (b) considers only one mapping between the two partially instantiated queries, namely the identity mapping, and thus produces a less complete but simpler test condition that has no disjuncts on the RHS.

Three other assertions can be used to represent constraint $I7$. Namely, assertions where either or both of the equality predicates $A{=}S$ and $D{=}B$ occur on the RHS of the assertion. However, these assertions are not expressible in the language we have considered. In Section 4.5.2 we generalize the assertion language to allow arithmetic conjuncts on the RHS. However, even with the more general assertions, the test condition for the above example stays the same. □

Intuitively, $\mathsf{TC}(C, l, \mu)$ is based on a single symbol mapping and is sufficient but not complete. The point to be noted here is that using just the identity mapping generates sufficient test conditions for constraints more powerful than conjunctive query constraints. For instance, for conjunctive queries that use negation and no arithmetic the existence of a symbol mapping from the positive subgoals of query $Q2$ to the positive subgoals of query $Q1$ and from the negative subgoals of query $Q2$ to the negative subgoals of query $Q1$ is a sufficient condition for $Q1 \subseteq Q2$. However, the condition is not necessary. When arithmetic inequalities are also used, the sufficient condition also needs $I(Q1) \Rightarrow I(Q2)$. The test condition stated in Definition 4.2.1 is derived using this intuition. The test condition can be proved correct for a class of queries that properly contains conjunctive queries with negation and arithmetic inequalities (for instance constraint $I1$).

Using multiple mappings we can generate test conditions that are more powerful than $\mathsf{TC}(C, l, \mu)$. Alternative mappings introduce disjunctions on the RHS of the test condition. Geometrically, the problem corresponds to determining if a union of k-dimensional spaces contains another k-dimensional space. This problem is computationally more expensive than determining if a k-dimensional space is contained in another similar space. Also, it is not possible always to eliminate universally quantified remote variables when the RHS of an implication has disjunctions. Specifically, the strategy described in Appendix B does not work. However, if there are disjunctions on the RHS of the test condition, then the discussion of Chapter 3 becomes relevant. The results of Sections 3.1 and 3.2 can be used if $g(\mu, \bar{Y}, \bar{Z}, \bar{C})$ is an independently constrained sentence or a "$\leq$" sentence.

An alternative to having disjunctions on the RHS of the test condition is to generate one test – similar to $\mathsf{TC}(C, l, \mu)$ – from each of the possible mappings. As a consequence we obtain a disjunction of implications of the form $\bigvee_{i=1}^{n}(A \Rightarrow B_i)$ instead of an implication of the form $A \Rightarrow \bigvee_{i=1}^{n}(B_i)$. Thus, for each constraint we could have several tests such that if any of these tests succeeds, then the constraint continues to hold. Geometrically, this enhanced test corresponds to determining if a k-dimensional parallelepiped is contained in one of a set of parallelepipeds and is therefore not the same as checking containment in the union of the parallelepipeds. The following example illustrates this approach.

EXAMPLE 4.4.3 Consider constraint $I7$ from Example 4.4.2.

$\texttt{panic} :- \texttt{bus\_from}(S, D)\ \&\ \texttt{depot}(S)\ \&\ \texttt{depot}(D).$

Using the four mappings described earlier, we can generate the following four tests that are each of the form of $\mathsf{TC}(C, l, \mu)$.

$$\exists U, V\, \forall S, D : [\texttt{bus\_from}(U, V) \wedge ((S = a, D = b) \Rightarrow (S = U, D = V))].$$
$$\exists U, V\, \forall S, D : [\texttt{bus\_from}(U, V) \wedge ((S = a, D = b) \Rightarrow (S = V, D = U))].$$
$$\exists U, V\, \forall S, D : [\texttt{bus\_from}(U, V) \wedge ((S = a, D = b) \Rightarrow (S = U, D = U))].$$
$$\exists U, V\, \forall S, D : [\texttt{bus\_from}(U, V) \wedge ((S = a, D = b) \Rightarrow (S = V, D = V))].$$

If any of the above tests successfully checks constraint $I7$ on insertion of tuple $\texttt{bus\_from}(a, b)$, then the constraint continues to hold. □

In the remainder of this thesis we do not consider using multiple mappings. Instead we concentrate on how to handle more expressive constraint assertions using the simpler test condition $\mathsf{TC}(C, l, \mu)$ that uses only the identity mapping. Thus, we compromise generating more powerful tests and instead generate the weaker test $\mathsf{TC}(C, l, \mu)$ for a larger class of constraints.

4.4.4 Completeness

To highlight the completeness properties of $\mathsf{TC}(C, l, \mu)$, first we consider those of the assertions described in Section 4.1 that produce violation conditions expressible as conjunctive query constraints with arithmetic inequalities. For such constraints, the previous sections illustrate that $\mathsf{TC}(C, l, \mu)$ is not complete because it considers only single cover tuples and single mappings. Now the question of interest is if there are classes of constraints for which $\mathsf{TC}(C, l, \mu)$, without generalizations, is the complete local test.

Recall the process of generating test conditions for conjunctive query constraints as described by the four step process on Page 31. The resulting test condition, I_t, was an arithmetic implication that had a disjunctive RHS. The disjunctions on the RHS arose out of three factors: multiple constraints in the set of constraints $\mathbf{C}$, multiple symbol mappings, and multiple tuples in L. Can $\mathsf{TC}(C, l, \mu)$ express I_t for any specific class of constraints? We consider each of the three factors that result in disjunctions in I_t and see how the factors affect $\mathsf{TC}(C, l, \mu)$.

Multiple constraints are not a factor in this chapter because we assume that there is only a single constraint in the set $\mathbf{C}$.

Next, we restrict the class of CQCs such that there is only one mapping from $\mathsf{Red}(\alpha, l, C)$ to $\mathsf{Red}(\mu, l, C)$. This restriction can be achieved in many ways, for instance by ensuring that no predicate is used more than once in the constraint. The restriction of a single mapping eliminates another source of disjunctions from the test condition obtained by using containment mapping. The last source of disjunctions, multiple cover tuples, can be removed by restricting the kind of arithmetic inequalities permitted in the constraint. In particular, consider the constraint classes LibCQC and RibCQC described in Section 3.2. These classes guarantee that only a single tuple is needed to check containment.

Thus, for constraints that satisfy the restrictions of the above paragraph, we can guarantee that $\mathsf{TC}(C, l, \mu)$ is indeed the complete local test when $\mathbf{C}$ has only one constraint. In fact, if repeated predicates are disallowed, then we can get complete local tests even if the conjunctive queries use negated EDB predicates. Thus, we claim that if the assertion C introduced in Section 4.1 has no repeated predicates and if g is either a "$\leq$" sentence or a "$\geq$" sentence as per Definition 3.2.2 on Page 43, then $\mathsf{TC}(C, l, \mu)$ is the complete local test for constraint C.

4.5 More General Constraints

Now we consider generating test conditions of the same form as $\mathsf{TC}(C, l, \mu)$ (without any of the proposed generalizations) for constraint assertions more powerful than the assertion language of Section 4.1. The first extension removes the restriction on the quantification of $\bar{Y}$ and $\bar{Z}$ in sentence C of Section 4.1. The second extension permits relational expressions with arithmetic inequalities on the RHS of the integrity constraint assertion. The last extension allows disjunctions of relational expressions with arithmetic to appear on the RHS of the assertion. Tests conditions for all these languages are presented. The first extension does not correspond to an intuitive class of constraints but is useful because it provides a proof for the class we considered in Section 4.1. The last two extensions do correspond to intuitive constraint assertions classes that are more powerful than those considered until now. The last two extensions are both instances of the topmost circled dot on the right branch of Figure 4.1.

4.5.1 Unrestricted Quantifiers

We consider a sentence C of the form described in Section 4.1 and remove the restrictions on the quantifiers of $\bar{Y}$ and $\bar{Z}$. We therefore consider a first order logic sentence B with the variables $\forall\bar{Y}\exists\bar{Z}$ replaced by a sequence of arbitrarily quantified variables denoted $\Theta\bar{Y}$.

$$B: \quad \forall\bar{X}\,\Theta\bar{Y} : [(L(\bar{X}) \wedge R_1(\bar{Y}_1) \wedge \ldots \wedge R_k(\bar{Y}_k) \wedge g(\bar{X},\bar{Y},\bar{c})) \;\Rightarrow\; (S_1(\bar{Y}'_1) \vee \ldots \vee S_n(\bar{Y}'_n))]$$

where:

$\Theta\bar{Y} = \alpha_1 Y_1 \alpha_2 Y_2 \ldots$, each α_i is a $\forall$ or $\exists$ quantifier, and $Y_1, Y_2, \ldots$ are variables in $\bar{Y}$.

The other terms occurring in the assertion are the same as described before. The restrictions for evaluability given in Section 4.1 also apply here.

Definition 4.5.1 ($\mathsf{TC}_g(B,l,\mu)$) Consider an integrity constraint B:

$$B: \quad \forall\bar{X}\,\Theta\bar{Y} : [(L(\bar{X}) \wedge R_1(\bar{Y}_1) \wedge \ldots \wedge R_k(\bar{Y}_k) \wedge g(\bar{X},\bar{Y},\bar{c})) \;\Rightarrow\; (S_1(\bar{Y}'_1) \vee \ldots \vee S_n(\bar{Y}'_n))]$$

Let μ represent a tuple inserted into the accessible relation L. The test condition is as follows:

$$\mathsf{TC}_g(B,l,\mu): \quad \exists\bar{X}\,\forall\bar{Y} : [g(\mu,\bar{Y},\bar{c}) \;\Rightarrow\; (L(\bar{X}) \wedge g(\bar{X},\bar{Y},\bar{c}))]$$

□

The following theorem proves the correctness of Definition 4.5.1.

Theorem 4.5.1 *Consider an integrity constraint B and a tuple μ inserted into relation L. If the database satisfies integrity constraint assertion B before adding tuple μ, and if the test condition $\mathsf{TC}_g(B,l,\mu)$ is satisfied by the accessible relation L, then the database satisfies integrity constraint assertion B after inserting tuple μ.* □

Proof: The integrity constraint assertion is assumed to be $\forall\bar{X}\,\Theta\bar{Y} : \psi(\bar{X},\bar{Y},\bar{c})$ and the test condition $\exists\bar{X}\,\forall\bar{Y} : [\psi(\bar{X},\bar{Y},\bar{c}) \Rightarrow \psi(\mu,\bar{Y},\bar{c})]$. We prove Theorem 4.5.1 in two steps. In **Step 1** we prove that the test condition along with the initial consistency assumption, $\forall\bar{X}\,\Theta\bar{Y} : \psi(\bar{X},\bar{Y},\bar{c})$, implies that the database is consistent after inserting tuple μ. In **Step 2** ψ is replaced by the logical form of integrity constraint assertions considered in Theorem 4.2.1 to generate the test condition $\mathsf{TC}_g(B,l,\mu)$.

<u>**Step 1**</u> Prove the following sentence:

$$t: \quad (\exists\bar{X}\,\forall\bar{Y} : [\psi(\bar{X},\bar{Y},\bar{c}) \Rightarrow \psi(\mu,\bar{Y},\bar{c})] \wedge \forall\bar{X}\,\Theta\bar{Y} : \psi(\bar{X},\bar{Y},\bar{c})) \;\Rightarrow\; \Theta\bar{Y} : \psi(\mu,\bar{Y},\bar{c})$$

However, we will use a different form of goal t. The LHS of the implication will be restricted and instead of using the sentence

$$a: \quad \exists\bar{X}\,\forall\bar{Y} : [\psi(\bar{X},\bar{Y},\bar{c}) \Rightarrow \psi(\mu,\bar{Y},\bar{c})] \wedge \forall\bar{X}\,\Theta\bar{Y} : \psi(\bar{X},\bar{Y},\bar{c})$$

Let $\bar{X}^o$ be a particular instantiation of $\bar{X}$ that makes (a) true. That is: sentence:

$$b: \quad \forall\bar{Y} : [\psi(\bar{X}^o,\bar{Y},\bar{c}) \Rightarrow \psi(\mu,\bar{Y},\bar{c})] \wedge \Theta\bar{Y} : \psi(\bar{X}^o,\bar{Y},\bar{c})$$

Note, (a) implies (b) (assuming that the domain of $\bar{X}$ is not empty) and therefore if we prove that (b) implies the RHS of the implication in (t), then (a) also implies RHS(t). The proof is by induction over the number of variables in $\bar{Y}$ which is the same as the number of quantifiers in Θ.

Base Case Θ is empty *i.e.* there are no variables in $\bar{Y}$. We need to prove:

$$[\psi(\bar{X}^o, \bar{c}) \Rightarrow \psi(\mu, \bar{c})] \wedge \psi(\bar{X}^o, \bar{c}) \quad \Rightarrow \quad \psi(\mu, \bar{c})$$

The truth of the base case sentence is easy to observe.

Induction Step Θ contains $n+1$ quantifiers. Assume that when Θ contains n quantifiers the following sentence is true:

$$I: \quad (\forall \bar{Y} : [\psi(\bar{X}^o, \bar{Y}, \bar{c}) \Rightarrow \psi(\mu, \bar{Y}, \bar{c})] \wedge \Theta \bar{Y} : \psi(\bar{X}^o, \bar{Y}, \bar{c})) \quad \Rightarrow \quad \Theta \bar{Y} : \psi(\mu, \bar{Y}, \bar{c})$$

- **Case 1**: The $n+1^{st}$ variable Z is universally quantified:
  Initial Consistency Assumption: $\forall Z\, \Theta \bar{Y} : \psi(\bar{X}^o, Z, \bar{Y}, \bar{c})$.
  Test Condition: $\forall Z\, \forall \bar{Y} : [\psi(\bar{X}^o, Z, \bar{Y}, \bar{c}) \Rightarrow \psi(\mu, Z, \bar{Y}, \bar{c})]$.
  Intended Conclusion: $\forall Z\, \Theta \bar{Y} : \psi(\mu, Z, \bar{Y}, \bar{c})$.
  We need to prove:

  $$(\forall Z\, \forall \bar{Y} : [\psi(\bar{X}^o, Z, \bar{Y}, \bar{c}) \Rightarrow \psi(\mu, Z, \bar{Y}, \bar{c})] \wedge \forall Z\, \Theta \bar{Y} : \psi(\bar{X}^o, Z, \bar{Y}, \bar{c})) \;\Rightarrow\; \forall Z\, \Theta \bar{Y} : \psi(\mu, Z, \bar{Y}, \bar{c})$$

  Consider z, an instance of variable Z. Because Z is universally quantified in all formulas, we can replace Z by z in all the components of the sentence to obtain:

  $$(\forall \bar{Y} : [\psi(\bar{X}^o, z, \bar{Y}, \bar{c}) \Rightarrow \psi(\mu, z, \bar{Y}, \bar{c})] \wedge \Theta \bar{Y} : \psi(\bar{X}^o, z, \bar{Y}, \bar{c})) \quad \Rightarrow \quad \Theta \bar{Y} : \psi(\mu, z, \bar{Y}, \bar{c})$$

  This sentence has only n quantifiers in Θ and the induction hypothesis (I) implies that the sentence is true for the given constant z. Because the constant z was an arbitrary constant, the sentence would be true for all variables Z proving our intended conclusion.

- **Case 2**: The $n+1^{st}$ variable Z is existentially quantified:
  Initial Consistency Assumption: $\exists Z\, \Theta \bar{Y} : \psi(\bar{X}^o, Z, \bar{Y}, \bar{c})$.
  Test Condition: $\forall Z\, \forall \bar{Y} : [\psi(\bar{X}^o, Z, \bar{Y}, \bar{c}) \Rightarrow \psi(\mu, Z, \bar{Y}, \bar{c})]$.
  Intended Conclusion: $\exists Z\, \Theta \bar{Y} : \psi(\mu, Z, \bar{Y}, \bar{c})$.
  We need to prove:

  $$(\forall Z\, \forall \bar{Y} : [\psi(\bar{X}^o, Z, \bar{Y}, \bar{c}) \Rightarrow \psi(\mu, Z, \bar{Y}, \bar{c})] \wedge \exists Z\, \Theta \bar{Y} : \psi(\bar{X}^o, Z, \bar{Y}, \bar{c})) \;\Rightarrow\; \exists Z\, \Theta \bar{Y} : \psi(\mu, Z, \bar{Y}, \bar{c})$$

  Assuming that `dom`(Z) is not empty, consider z, an instance of variable Z such that the conjunct $\exists Z\, \Theta \bar{Y} : \psi(\bar{X}^o, Z, \bar{Y}, \bar{c})$ is satisfied. The initial consistency assumption gives us that the existing inaccessible relations provide such a z. Substituting constant z for variable Z in the above sentence, we get:

  $$(\forall Z\, \forall \bar{Y} : [\psi(\bar{X}^o, Z, \bar{Y}, \bar{c}) \Rightarrow \psi(\mu, Z, \bar{Y}, \bar{c})] \wedge \Theta \bar{Y} : \psi(\bar{X}^o, z, \bar{Y}, \bar{c})) \quad \Rightarrow \quad \exists Z\, \Theta \bar{Y} : \psi(\mu, Z, \bar{Y}, \bar{c}).$$

  The first component of the conjunction holds $\forall Z$ and therefore holds for $Z = z$ resulting in:

  $$g1: \quad (\forall \bar{Y} : [\psi(\bar{X}^o, z, \bar{Y}, \bar{c}) \Rightarrow \psi(\mu, z, \bar{Y}, \bar{c})] \wedge \Theta \bar{Y} : \psi(\bar{X}^o, z, \bar{Y}, \bar{c})) \quad \Rightarrow \quad \exists Z\, \Theta \bar{Y} : \psi(\mu, Z, \bar{Y}, \bar{c})$$

  In the LHS of the implication the quantifier on predicate ψ now has length n. The induction hypothesis can therefore be used. The induction hypothesis states:

  $$I: \quad (\forall \bar{Y} : [\psi(\bar{X}^o, \bar{Y}, \bar{c}) \Rightarrow \psi(\mu, \bar{Y}, \bar{c})] \wedge \Theta \bar{Y} : \psi(\bar{X}^o, \bar{Y}, \bar{c})) \quad \Rightarrow \quad \Theta \bar{Y} : \psi(\mu, \bar{Y}, \bar{c})$$

Replacing $\psi(\bar{X}^o, \bar{Y}, \bar{c})$ with a predicate that has an additional constant argument in a fixed position, we can derive the following sentence from the induction hypothesis:

$$I2: \quad (\forall \bar{Y} : [\psi(\bar{X}^o, z, \bar{Y}, \bar{c}) \Rightarrow \psi(\mu, z, \bar{Y}, \bar{c})] \wedge \Theta\bar{Y} : \psi(\bar{X}^o, z, \bar{Y}, \bar{c})) \quad \Rightarrow \quad \Theta\bar{Y} : \psi(\mu, z, \bar{Y}, \bar{c})$$

In addition $z \in \mathtt{dom}(Z)$ and therefore:

$$I3: \quad (\Theta\bar{Y} : \psi(\mu, z, \bar{Y}, \bar{c}) \Rightarrow \exists Z \Theta\bar{Y} : \psi(\mu, Z, \bar{Y}, \bar{c}))$$

The sentences $I2$, $I3$ and modus-ponens gives us the goal $g1$ and hence the intended conclusion.

Step 2 Recall that when the integrity constraint assertion ψ is of the form:

$$(L(\bar{X}) \wedge R_1(\bar{Y}_1) \wedge \ldots \wedge R_k(\bar{Y}_k) \wedge g(\bar{X}, \bar{Y}, \bar{c})) \Rightarrow (S_1(\bar{Y}'_1) \vee \ldots \vee S_n(\bar{Y}'_n)).$$

then the test condition $\mathsf{TC}_g(B, l, \mu)$ stated in Theorem 4.5.1 is:

$$\mathsf{TC}_g(B, l, \mu): \quad \exists\bar{X}\, \forall\bar{Y} : [g(\mu, \bar{Y}, \bar{c}) \quad \Rightarrow \quad (L(\bar{X}) \wedge g(\bar{X}, \bar{Y}, \bar{c}))]$$

$\mathsf{TC}_g(B, l, \mu)$ is obtained by substituting for ψ in $\mathcal{T}$:

$$\mathcal{T}: \quad \exists\bar{X}\, \forall\bar{Y} : [\psi(\bar{X}, \bar{Y}, \bar{c}) \Rightarrow \psi(\mu, \bar{Y}, \bar{c})]$$

After substituting $(l(\bar{X}) \wedge R_1(\bar{Y}_1) \ldots \wedge g(\bar{X}, \bar{Y})) \Rightarrow (S_1(\bar{Y}'_1) \vee \ldots \vee S_n(\bar{Y}'_n))$ for ψ in $\mathcal{T}$ we get:

$$\exists\bar{X}\, \forall\bar{Y} : [\{(l(\bar{X}) \wedge R_1(\bar{Y}_1) \wedge \ldots \wedge g(\bar{X}, \bar{Y})) \Rightarrow (S_1(\bar{Y}'_1) \vee \ldots)\} \quad \Rightarrow$$
$$\{(l(\mu) \wedge R_1(\bar{Y}_1) \wedge \ldots \wedge g(\mu, \bar{Y})) \Rightarrow (S_1(\bar{Y}'_1) \vee \ldots)\}]$$

$l(\mu)$ is true because μ is inserted into relation L. Thus the above sentence can be simplified to yield:

$$\exists\bar{X}\, \forall\bar{Y} : [\{(l(\bar{X}) \wedge R_1(\bar{Y}_1) \wedge \ldots \wedge g(\bar{X}, \bar{Y})) \Rightarrow (S_1(\bar{Y}'_1) \vee \ldots)\} \quad \Rightarrow$$
$$\{(R_1(\bar{Y}_1) \wedge \ldots \wedge g(\mu, \bar{Y})) \Rightarrow (S_1(\bar{Y}_1) \vee \ldots)\}]$$

Using the valid logic sentence $(a \Rightarrow b) \equiv (\neg a \vee b)$ we obtain:

$$\exists\bar{X}\, \forall\bar{Y} : [\{\neg(l(\bar{X}) \wedge R_1(\bar{Y}_1) \wedge \ldots \wedge g(\bar{X}, \bar{Y})) \vee S_1(\bar{Y}'_1) \vee \ldots\} \quad \Rightarrow$$
$$\{\neg(R_1(\bar{Y}_1) \wedge \ldots \wedge g(\mu, \bar{Y})) \vee S_1(\bar{Y}'_1) \vee \ldots\}]$$

Using the valid logic sentence $(a \Rightarrow b) \quad \Rightarrow \quad (a \vee c \ \Rightarrow b \vee c)$ we obtain:

$$\exists\bar{X}\, \forall\bar{Y} : [\neg(l(\bar{X}) \wedge R_1(\bar{Y}_1) \wedge \ldots \wedge g(\bar{X}, \bar{Y})) \quad \Rightarrow \quad \neg(R_1(\bar{Y}_1) \wedge \ldots \wedge g(\mu, \bar{Y}))]$$

Taking the contrapositive of the above, we get

$$\exists\bar{X}\, \forall\bar{Y} : [(R_1(\bar{Y}_1) \wedge \ldots \wedge g(\mu, \bar{Y})) \quad \Rightarrow \quad (l(\bar{X}) \wedge R_1(\bar{Y}_1) \wedge \ldots \wedge g(\bar{X}, \bar{Y}))]$$

Using the valid logic sentence $(a \Rightarrow b) \quad \Rightarrow \quad (a \wedge c \ \Rightarrow b \wedge c)$ we get $\mathsf{TC}_g(B, l, \mu)$. Every step of the transformation produced either an equivalent sentence or one that implied the sentence before. Therefore, $\mathsf{TC}_g(B, l, \mu) \Rightarrow \mathcal{T}$.

If $\mathsf{TC}_g(B, l, \mu)$ is satisfiable, then the LHS of the sentence (t) (beginning of Step 1) becomes true. Because (t) has been proved valid we conclude that the RHS also has to be true. Therefore the integrity constraint assertion ψ is true in the new database state when $\mathsf{TC}_g(B, l, \mu)$ is satisfied by the accessible relation. ∎

4.5.2 Arithmetic Inequalities on the Right Hand Side

Until now we restricted the RHS of the assertion to be simple existential facts that referred to single relations. However, constraint assertions may require more involved right hand sides. Thus, the following assertion allows the RHS to be of the same form as the LHS. That is, the assertion is a first-order logic sentence of the following form:

$$A: \quad \forall \bar{X}\,\forall \bar{Y}\,\exists \bar{Z} : [(L(\bar{X}) \wedge R_1(\bar{Y}_1) \wedge \ldots \wedge R_k(\bar{Y}_k) \wedge g_1(\bar{X}, \bar{Y}, \bar{Z}, \bar{c})) \Rightarrow (S_1(\bar{Y}'_1, \bar{Z}'_1) \wedge \ldots \wedge S_n(\bar{Y}'_n, \bar{Z}'_n) \wedge g_2(\bar{X}, \bar{Y}, \bar{Z}, \bar{c}))]$$

The same restrictions for evaluability, as in Section 4.1, apply to this language. In terms of Datalog, the above integrity constraint assertion corresponds to a Datalog program that has one rule defining `panic`. This rule is a conjunctive query with arithmetic inequalities (g_1) and one negated subgoal s. The predicate in subgoal s is an IDB predicate that is defined using a single conjunctive query with arithmetic inequalities (g_2). The test condition for this more general constraint language is defined as follows.

Definition 4.5.2 ($\mathsf{TC}_a(A, l, \mu)$) For integrity constraint A:

$$A: \quad \forall \bar{X}\,\forall \bar{Y}\,\exists \bar{Z} : [(L(\bar{X}) \wedge R_1(\bar{Y}_1) \wedge \ldots \wedge R_k(\bar{Y}_k) \wedge g_1(\bar{X}, \bar{Y}, \bar{Z}, \bar{c})) \Rightarrow (S_1(\bar{Y}'_1, \bar{Z}'_1) \wedge \ldots \wedge S_n(\bar{Y}'_n, \bar{Z}'_n) \wedge g_2(\bar{X}, \bar{Y}, \bar{Z}, \bar{c}))]$$

Let μ represent a tuple inserted into the accessible relation L. The test condition is as follows:

$$\mathsf{TC}_a(A, l, \mu): \quad \exists \bar{X}\,\forall \bar{Y}\,\forall \bar{Z} : [L(\bar{X}) \wedge (g_1(\mu, \bar{Y}, \bar{Z}, \bar{c}) \Rightarrow g_1(\bar{X}, \bar{Y}, \bar{Z}, \bar{c})) \wedge (g_2(\bar{X}, \bar{Y}, \bar{Z}, \bar{c}) \Rightarrow g_2(\mu, \bar{Y}, \bar{Z}, \bar{c}))]$$

□

Theorem 4.5.2 *Consider an integrity constraint A and a tuple μ inserted into relation L. If the database satisfies integrity constraint assertion A before adding tuple μ, and if the test condition $\mathsf{TC}_a(A, l, \mu)$ is satisfied by the accessible relation L, then the database satisfies integrity constraint assertion A after inserting tuple μ.* □

Proof: Similar to the proof of Theorem 4.5.1. ∎

4.5.3 More Complex Right Hand Sides

We further extend the class of constraint assertions to allow even more complicated sentences on the RHS of the assertions. We consider a first order logic sentence of the following form:

$$\begin{aligned} A': \quad \forall \bar{X}\,\forall \bar{Y}\,\exists \bar{Z} : [(L(\bar{X}) \wedge R_1(\bar{Y}_1) \wedge \ldots \wedge R_k(\bar{Y}_k) \wedge g_0(\bar{X}, \bar{Y}, \bar{Z}, \bar{c})) \Rightarrow \\ (S_{11}(\bar{Y}'_{11}, \bar{Z}'_{11}) \wedge \ldots \wedge S_{1n_1}(\bar{Y}'_{1n_1}, \bar{Z}'_{1n_1}) \wedge g_1(\bar{X}, \bar{Y}, \bar{Z}, \bar{c})) \vee \\ (S_{21}(\bar{Y}'_{21}, \bar{Z}'_{21}) \wedge \ldots \wedge S_{2n_2}(\bar{Y}'_{2n_2}, \bar{Z}'_{2n_2}) \wedge g_2(\bar{X}, \bar{Y}, \bar{Z}, \bar{c})) \vee \\ \ldots\ldots\ldots \\ (S_{m1}(\bar{Y}'_{m1}, \bar{Z}'_{m1}) \wedge \ldots \wedge S_{mn_m}(\bar{Y}'_{mn_m}, \bar{Z}'_{mn_m}) \wedge g_m(\bar{X}, \bar{Y}, \bar{Z}, \bar{c}))] \end{aligned}$$

The same restrictions for evaluability, as in Section 4.1, apply to this language. The above integrity constraint assertion corresponds to a disjunction of m assertions, $A'_1 \vee A'_2 \vee \ldots \vee A'_m$, where A'_i is of the form:

$$A'_i\colon\ \ \forall \bar{X}\, \forall \bar{Y}\, \exists \bar{Z} : [(L(\bar{X}) \wedge R_1(\bar{Y}_1) \wedge \ldots \wedge R_k(\bar{Y}_k) \wedge g_0(\bar{X}, \bar{Y}, \bar{Z}, \bar{c})) \Rightarrow \\ (S_{i1}(\bar{Y}'_{i1}, \bar{Z}'_{i1}) \wedge \ldots \wedge S_{in_i}(\bar{Y}'_{in_i}, \bar{Z}'_{in_i}) \wedge g_i(\bar{X}, \bar{Y}, \bar{Z}, \bar{c}))]$$

The above assertion represents a Datalog program that has one rule defining `panic` such that this rule is a conjunctive query with arithmetic inequalities and an arbitrary number of negated subgoals. Each negated subgoal uses an IDB predicate that is defined using exactly one conjunctive query with arithmetic inequalities.

The test condition for A' is a disjunction of m test conditions, where the i^{th} condition checks that the assertion $\mathsf{LHS} \Rightarrow \mathsf{RHS}_i$ holds after the insertion. Each of the m test conditions is of the form $\mathsf{TC}_a(A'_i, l, \mu)$ from Definition 4.5.2. The assertion A' holds in the new database state if any of the m test conditions succeeds.

Chapter 5

Extending Local Checking

Chapter Outline In this chapter first we consider how deletions are checked locally. Then we prove the correctness of the use of multiple local checking methods in parallel by different sites to check the same constraint while treating their own updated relation as local and treating all other relations as remote. In Section 5.3 we discuss the applicability of local constraint checking when the initial consistency assumption does not hold. Finally, in Section 5.4 we describe how local checking results can be applied to materialized view maintenance.

5.1 Deletions

Until now we have considered insertions as the only update. The techniques described earlier for handling insertions can also be used to determine when deletions from relations do not violate constraints. First we give the intuition for handling deletions by using the pictorial interpretation of illegal states of the database introduced in Section 2.2. Then we describe the type of constraints for which deletions can be handled and finally give the test conditions for these constraints.

5.1.1 Intuition

Consider Figure 5.1, which is a variant of Figure 2.2 introduced earlier on Page 21. We reinterpret the above figure to illustrate the intuition for deletions. Let circle S_0 represent all possible states of relations $\bar{R}$ that *do not* violate constraint C with tuple t_0. That is, S_0 represents all the *legal* states of the inaccessible database given tuple t_0.[1] Given that the current state of the remote database does not violate C, we can infer that the current state of $\bar{R}$ lies *inside* circle S_0.

Let relation L contain tuples t_0, t_1, and t_2, and let tuple t_1 be deleted from L. Circle S_1 represents all those states of $\bar{R}$ that do not violate constraint C given tuple t_1. Given that S_1 is contained in S_0, all states of $\bar{R}$ that do not violate constraint C given tuple t_1 also do not violate

[1] As before, we are assuming that constraint C does not use multiple tuples from relation L in any derivation.

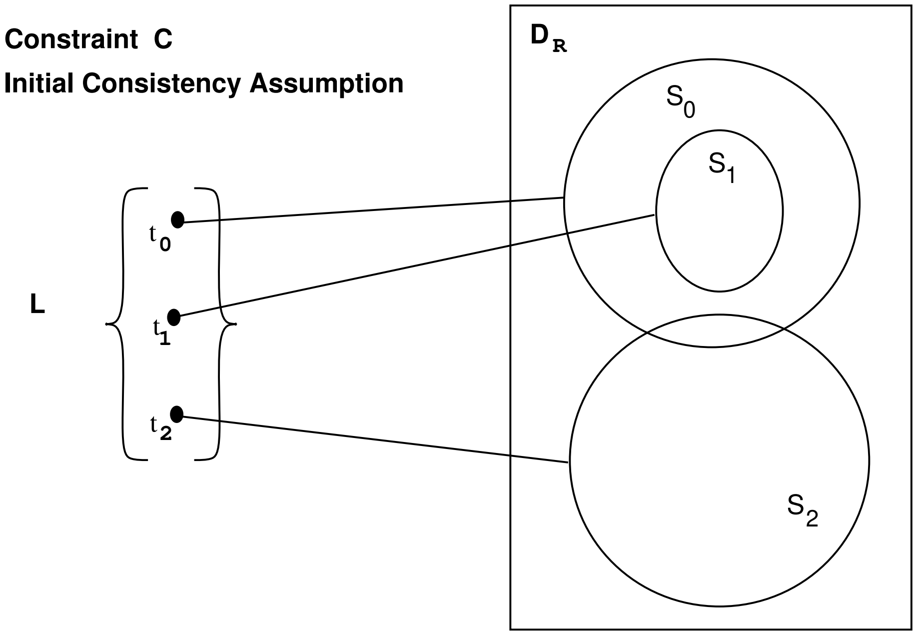

Figure 5.1: Locally Checking Constraints in Response to Deletions

C given tuple t_0. The initial consistency assumption implies that the current state of $\bar{R}$ does not violate C with t_1 and thus we infer that the current state of $\bar{R}$ lies inside circle S_1 and therefore inside circle S_0. Hence, if tuple t_1 is deleted from L then the current state of $\bar{R}$ does not violate constraint C with t_0. In this case, we say that the tuple t_0 covers the deleted tuple t_1.

Now suppose tuple t_2 is to be deleted from relation L, where L contains tuples t_0 and t_2. Circle S_2 represents all those states of $\bar{R}$ that do not violate constraint C given tuple t_2. Given that circle S_2 is not contained in the circle S_0, there could be some state of $\bar{R}$ that does not violate constraint C given tuple t_2, but that violates C given t_0. Such a state would be in S_2 but not in S_0. The current state of $\bar{R}$ could be one such state. Therefore, tuple t_0 cannot be used to conclude that the current state of $\bar{R}$ does not violate C when tuple t_2 is deleted.

Locally checking deletions involves finding a cover tuple for the deleted tuple such that the legal states of the inaccessible relations $\bar{R}$, given the deleted tuple, are a subset of the legal states of $\bar{R}$, given the cover tuple. Note, we could use a set of existing tuples to build a cover for the deleted tuple $\bar{v}$, but we do not consider multiple cover tuple techniques for deletions.

5.1.2 Language

Consider constraint assertions of the form C:

$$C: \quad \forall \bar{X}\, \forall \bar{Y}\, \exists \bar{Z}\, \exists \bar{W} : [(R_{1\ldots k}(\bar{Y}) \wedge g(\bar{X}, \bar{Y}, \bar{Z}, \bar{W}, \bar{c})) \Rightarrow (L(\bar{X}, \bar{W}) \vee S_1(\bar{Y}'_1, \bar{Z}'_1) \vee \ldots \vee S_n(\bar{Y}'_n, \bar{Z}'_n))]$$

Where $R_{1\ldots k}(\bar{Y})$ represents a conjunction of k relations, each of which uses a subset of the variables in $\bar{Y}$ as arguments. The restrictions on the relations for evaluability are exactly the same as in Section 4.1. These restrictions require that each variable in $\bar{X}$ is equated to either a variable in $\bar{Y}$ or to some constant. Note, the assertion C is similar to assertion C in Section 4.1. However, the updated relation, and thus the accessible relation, is on the RHS instead of the LHS. An assertion of the form C can become false if more tuples are added to a relation on its LHS or if tuples are removed from a relation on its RHS. This corresponds to the constraint being potentially violated when tuples are inserted into relations occurring positively or when tuples are deleted from relations occurring negatively. Thus, deletions to relation L are relevant updates.

5.1.3 Test Condition for Deletions

The test condition for locally checking constraint C is derived using the pictorial intuition given above. We need to prove that the set of legal remote database states corresponding to the deleted tuple $\bar{v}$ is contained in the set of legal remote database states corresponding to some tuple remaining in L. Note, C is of the form $\mathsf{LHS}(C) \Rightarrow \mathsf{RHS}(C)$, or equivalently, $\neg\mathsf{LHS}(C) \vee \mathsf{RHS}(C)$. By substituting $\bar{v}$ into C we get a characterization of the legal states of the remote database given $\bar{v}$. That is, states in which either $\mathsf{LHS}(C)$ is false after $\bar{v}$ has been substituted into C, or $\mathsf{RHS}(C)$ is true. That is, the legal states for $\bar{v}$ are characterized by $\neg\mathsf{LHS}(C(\bar{v})) \vee K$, where K represents $\mathsf{RHS}(C(\bar{v}))$. Similarly, the legal states for tuple t in L are characterized by $\neg\mathsf{LHS}(C(t)) \vee K'$. Thus, t covers $\bar{v}$ if the legal states of $\bar{R}$ given $\bar{v}$ are contained in the legal states of $\bar{R}$ given t, *i.e.*, $[\neg\mathsf{LHS}(C(\bar{v})) \vee K] \Rightarrow [\neg\mathsf{LHS}(C(t)) \vee K']$. A sufficient condition for the above implication to be true is $\neg\mathsf{LHS}(C(\bar{v})) \Rightarrow \neg\mathsf{LHS}(C(t))$ or equivalently, $\mathsf{LHS}(C(t)) \Rightarrow \mathsf{LHS}(C(\bar{v}))$. This implication involving the LHS of the assertion can be reduced to an implication involving just the arithmetic inequalities. Thus the test condition turns out to be an implication involving arithmetic inequalities.

Definition 5.1.1 ($\mathsf{TC}_D(C, l, \mu)$) Consider an integrity constraint assertion C:

$$C\!: \quad \forall\bar{X}\,\forall\bar{Y}\,\exists\bar{Z}\,\exists\bar{W} : [(R_{1\ldots k}(\bar{Y}) \wedge g(\bar{X}, \bar{Y}, \bar{Z}, \bar{W}, \bar{c})) \Rightarrow (L(\bar{X}, \bar{W}) \vee S_1(\bar{Y}'_1, \bar{Z}'_1) \vee \ldots \vee S_n(\bar{Y}'_n, \bar{Z}'_n))]$$

Let $\bar{v}$ be a tuple deleted from the accessible relation L. The test condition is as follows:

$$\mathsf{TC}_D(C, l, \mu)\!: \quad \exists\bar{W}\,\exists\bar{X}\,\forall\bar{Y}\,\forall\bar{Z} : [L(\bar{X}, \bar{W}) \wedge (g(\bar{W}, \bar{X}, \bar{Y}, \bar{Z}, \bar{c}) \;\Rightarrow\; g(\bar{v}_x, \bar{v}_w, \bar{Y}, \bar{Z}, \bar{c}))]$$

where $\bar{v}_w$ and $\bar{v}_x$ are the projections of tuple $\bar{v}$ onto variables $\bar{W}$ and $\bar{X}$ respectively. □

Theorem 5.1.1 *Consider an integrity constraint C and a tuple $\bar{v}$ deleted from relation L. If the database satisfies integrity constraint assertion C before deleting tuple $\bar{v}$, and if the test condition $\mathsf{TC}(C, l, \mu)$ is satisfied by the accessible relation L, then the database satisfies integrity constraint assertion C after deleting tuple $\bar{v}$.* □

Proof: Identical to the proof of Theorem 4.5.1 stated in Section 4.5 on page 61. ∎

The techniques to evaluate $\mathsf{TC}(C, l, \mu)$, discussed in the previous chapter, can be used to evaluate $\mathsf{TC}_D(C, l, \mu)$. However, in both cases there is a difference. In $\mathsf{TC}(C, l, \mu)$ the parameters corresponding to the inserted tuple appear on the LHS of the implication, whereas in $\mathsf{TC}_D(C, l, \mu)$ the parameters corresponding to the deleted tuple appear on the RHS of the implication. The first technique, where each existing tuple in L is substituted into the test condition and the resulting implications are evaluated, can be used as before. The second technique, where universally quantified variables are eliminated from the test condition to give a selection condition on L needs to be changed as described in Appendix B.

5.1.4 Modifications

We treat tuple modifications as a deletion followed by an insertion (in that order). We discuss local tests for constraints in which the modified relation occurs positively. An interesting point to note here is that the deleted tuples can be stored temporarily and used in the local tests for better performance. The following example illustrates this idea.

EXAMPLE 5.1.1 Consider constraint $I4$ from Example 4.1.2. Constraint $I4$ is violated if there is at least one employee whose salary is greater than the salary of some manager in the same department, *i.e.*, when the following query derives `panic`.

$\texttt{panic} \coloneq \texttt{emp}(E, D, S)\ \&\ \texttt{dept}(D, MS)\ \&\ S > MS.$

Let the tuple $\texttt{emp}(john, toy, 100)$ be modified to $\texttt{emp}(john, toy, 90)$. Given that constraint $I4$ was not violated before the update, we can use $\texttt{emp}(john, toy, 100)$ as a cover for tuple $\texttt{emp}(john, toy, 90)$. Instead, if the update was treated as a deletion+insertion and the deleted tuple was not used in the test, then the local test would have failed to use the old value of the tuple to infer that the new value does not violate the constraint. □

Formally, the test condition for locally checking modifications is exactly the same as in the case of insertions, *i.e.*, as described in Sections 4.2 and 4.5. The difference is that the test is executed on the old relation that includes the old value of the modified tuple. Thus, the logic is the same as in the case of insertions but the implementation needs to be slightly different.

In case the modified relation occurs negatively, then we can model a modification as an insertion followed by a deletion and use the going-to-be deleted tuple in the local check. The process is symmetric to that for locally checking modifications in case the modified relation occurred positively.

5.2 Multiple Tests in Parallel

Until now we have discussed local constraint checking assuming that only one relation is updated. Consider the motivating scenario for our local checking methods: a constraint spans multiple relations that are not all part of the same database. In such a scenario, different sites may update their relations independently, and each of these sites may use a local constraint checking method to check if the constraint holds after the update. Thus, there is no coordination between the sites even though the sites are checking the same constraint. The question that arises in such a situation is whether this strategy may conclude that the constraint holds when actually the constraint is violated.

Therefore, in this section we consider what happens when local checking is done independently for multiple updated relations that are involved in the same constraint. That is, multiple relations are updated, and for each update a different test condition is executed that treats only the updated relation as accessible and other relations as inaccessible. We prove that if multiple tests are run in parallel then a constraint is inferred to hold only if it indeed does hold, and thus the independent tests cannot result in undetected violations.

We use the notation for constraints introduced in Section 2.4. Thus, a constraint is represented by its violation condition. Recall, $\mathbf{C} = \{C_0, C_1, \ldots, C_m\}$ is a set of constraints. Each $C_i \in \mathbf{C}$ uses some subset of the relations $\{L, R_1, \ldots, R_n\}$. Local checking uses the initial consistency assumption for $\mathbf{C}$, the update to relation L, and the contents of relation L, to check each constraint $C_i \in \mathbf{C}$. The algorithm for locally checking the constraints in $\mathbf{C}$, originally stated in Figure 2.3, is restated below.

---

(1) for $i \in 0 \ldots m$ do
(2) if $\forall \bar{R} : C_i(L \cup \{t\}, \bar{R}) \Rightarrow \mathbf{C}(L, \bar{R})$ then okay$(C_i) \leftarrow$ true **else** okay$(C_i) \leftarrow$ false

---

The results in Chapters 3 and 4 correctly implement the above algorithm for particular classes of constraints.

If two or more relations are updated then each occurrence of an updated relation is treated as an accessible relation and tests are run in parallel for each of these occurrences, to check for constraint violations. If a relation occurs multiple times in a constraint, then each occurrence is treated as an independent relation and a local check is executed for each occurrence. Thus, for the following discussion, we assume that no relation is repeated. We prove that if the tests are all successful, *i.e.* all tests conclude that no constraint in $\mathbf{C}$ is violated, then indeed the updates do not violate any constraint in $\mathbf{C}$. This result is crucial to the applicability of the test conditions described in this thesis.

Theorem 5.2.1 *Let* **C** *be a set of constraints that use relations* $\{R_1, \ldots, R_n\}$ *such that no constraint in* **C** *is violated in the current database state DB. Let relations* $R_1, \ldots, R_k$, $k \geq 1$, *in C be updated to give a new database state* DB'. *Let test* $\mathcal{T}_j$ *be the execution of the algorithm above assuming that relation* R_j *is accessible and all other relations are inaccessible. If each of the k test conditions* $\{\mathcal{T}_1, \ldots, \mathcal{T}_k\}$ *independently infers that no constraint in* **C** *is violated, then indeed no constraint in* **C** *is violated in the new database state* DB'. □

Proof: We represent the set of constraints **C** as a function $\bar{C}(R_1, \ldots, R_n)$ that computes `panic`, or is true, if any constraint in **C** is violated. Without loss of generality, say that relations R_1 and R_2 are updated. We want to prove that if the independent tests, one accessing just relation R_1 and the other accessing just relation R_2, infer that no constraint in **C** is violated, then indeed no constraint in **C** is violated in the new database state obtained by incorporating both updates (assuming initial consistency.)

Let relation R_i after update be referred to by R_i^ν. If the test for relation R_1 infers that no constraint in **C** is violated by changing R_1 to R_1^ν, then we can infer that the implication in line 3 of the algorithm above holds.

$\mathcal{A}$: $\forall X_2, \ldots, X_n : [\bar{C}(R_1^\nu, X_2, \ldots, X_n) \Rightarrow \bar{C}(R_1, X_2, \ldots, X_n)]$

where X_i represents an instance of relation R_i. Similarly, if the test condition accessing relation R_2 infers that no constraint in **C** is violated by changing R_2 to R_2^ν, then we can infer that:

$\mathcal{B}$: $\forall X_1, X_3, \ldots, X_n : [\bar{C}(X_1, R_2^\nu, X_3, \ldots, X_n) \Rightarrow \bar{C}(X_1, R_2, X_3, \ldots, X_n)]$

Suppose however, that some constraint in **C** is violated in the new database state *i.e.* $\bar{C}(R_1^\nu, R_2^\nu, \ldots, R_n)$ is true. Using $\mathcal{A}$ we can conclude therefore that $\bar{C}(R_1, R_2^\nu, \ldots, R_n)$ is true. Similarly, using this new conclusion and $\mathcal{B}$ we obtain that $\bar{C}(R_1, R_2, \ldots, R_n)$ is true. However, the initial consistency assumption tells us that $\bar{C}(R_1, R_2, \ldots, R_n)$ is false. Therefore, the assumption that $\bar{C}(R_1^\nu, R_2^\nu, R_3, \ldots, R_n)$ is true leads to a contradiction. ∎

As a result of the above theorem, constraints involving data on multiple sites can be checked on each of these sites using local checking without fear of incorrectness arising from interference from the other sites. If any one of these tests fails then alternative ways of checking constraints is needed, as described in the introduction.

5.3 Inferring Violations

Until now we have used the initial consistency assumption to develop local checks for constraints. However, suppose a constraint does not hold in a database. Can local checking be used in any reasonable manner? The following example illustrates some issues in using local checking when the initial consistency assumption does not hold.

EXAMPLE 5.3.1 Consider constraint $I4$ from Example 5.1.1. Let relation `emp` contain two tuples $(john, toy, 50)$ and $(mary, toy, 150)$ and let relation `dept` contain only tuple $(toy, 100)$. For this database, we can infer that $john$ does not violate constraint $I4$ and $mary$ does violate the constraint. Suppose we flag a tuple in `emp` if that tuple violates constraint $I4$. So $(mary, toy, 150)$ is flagged and $(john, toy, 50)$ is not flagged.

Now, we insert a tuple `emp`$(bob, toy, 175)$ into the database and we also assume that only relation `emp` is available for constraint checking. Using the flags on the tuples we can infer that `emp`$(bob, toy, 175)$ also violates the constraint. The reasoning is as follows. If $mary$ violates $I4$ then department toy has a manager whose salary is less than 150 and therefore less than 175. In fact, the reasoning is exactly as in the case of local checking to check if the inserted tuple does not violate the constraint.

As before, if we insert tuple `emp`$(sara, toy, 25)$ into the database then we can use $(john, toy, 50)$ to infer that $sara$ does not violate constraint $I4$ since $(john, toy, 50)$ is not flagged.

Note, if we insert tuple `emp`$(cary, toy, 75)$ into the database then we cannot use the existing tuples in `emp` in order to determine whether or not the inserted tuple violates the constraint. Such an insertion "falls through the cracks" with respect to local checking. □

Flagging tuples, as in the above example, partitions the tuples in the local relation into two groups, one set that violates the constraint and the other set that does not violate the constraint. We refer to these two sets as the OK and NOTOK sets. When the initial consistency assumption was true, then the entire relation was OK.

Let us see how flagged tuples can be used to do local checking. Let tuple μ be inserted into local relation L. If tuple μ is covered by a tuple in the set OK, then μ is also in set OK. If tuple μ *covers* a tuple in the set NOTOK, then μ is also in set NOTOK. If no tuple in OK covers μ and μ does not cover any tuple in NOTOK, then we need an alternate scheme for determining the effect of inserting μ on constraint C.

Flagging can be seen as another kind of partial information. However, it is not clear always what it means to say that "a tuple violates a constraint." For instance, consider the above example and treat relation `dept` as local. It is not clear if the tuple `dept`$(toy, 100)$ should be flagged NOTOK. This decision depends on the semantics of the particular constraint being considered. In [GT94, Tiw94] there is a detailed discussion that highlights the fact that the responsibility of constraint violations is usually shared asymmetrically by the participating relations, and thus often only one of the participating relations needs to be flagged. In addition, if there are multiple constraints in the system then tuples may be flagged with respect to some constraints and not flagged with respect to other constraints. Thus, the cost of flagging tuples is proportional to the number of constraints.

5.4 View Maintenance

In this section we describe how to adapt the results of the previous chapters for updating materialized views using only the view definition, the contents of the updated relation, and the update to the database.

EXAMPLE 5.4.1 Consider the following schema for an employee-department database which is an extension of the schema described in the introduction.

`emp`(E, D, S)	% *employee number E in department D has salary S*
`dept`(M, D, MS)	% *manager M in department D has salary MS*

Let view `bad_dept`(D, M) be defined to have all departments D that have a manager M and an employee whose salary is greater than the salary of manager M.

`bad_dept`(D, M) :- `emp`(E, D, S) & `dept`(M, D, MS) & $S > MS$.

Let tuple $(john, toy, 50)$ be inserted into relation `emp`. If only the view definition and the update are available then it is not possible to infer the effect of the insertion on the view, *i.e.*, the update cannot be inferred to be irrelevant with respect to the view [BCL89].

However, if we know that the updated relation `emp` already contains tuple $(mary, toy, 65)$ then we can infer that view `bad_dept` will not change after the insertion (assuming set semantics). This inference can be made by observing that if `emp`$(john, toy, 50)$ would cause the insertion of tuple $(toy, henry)$ in `bad_dept`, then the salary of manager *henry* must be < 50. Thus, the salary of manager *henry* would also be < 65, and $(toy, henry)$ would be in `bad_dept` even before the insertion, by virtue of tuple `emp`$(mary, toy, 65)$. This inference is made using the contents of the updated relation `emp`. Relation `dept` and view `bad_dept` are not used. Thus the results of Chapters 2, 3, and 4 would seem applicable to this problem. □

The above example illustrates that partial information can be used to determine that an update to some base relation does not cause an update to the view. In this section we discuss how to apply the results of the previous chapters to determine that an update does not affect a view, using only the view definition, the update, and the updated relation.

However, what about the case where a base relation update *does* result in an update to the view? For the case when only the updated relation is available, this problem is much more difficult to solve. Intuitively, it is difficult to infer the contribution of an update using only the updated relation because the contribution may involve tuples from other relations. We prove that this problem cannot be solved if the view uses even one more relation other than the updated relation. Note, however, that if the contents of the view are added to the partial information, then more inferences may be made because the contents of the view reveal parts of tuples that exist in the remote relation. Flagging tuples can also give information about which tuples contribute to the view. However, in this thesis we consider the problem of determining if an update contributes to the view using only the updated relation, the update, and the view definition. We do not use the contents of the view itself or any additional information such as flags.

5.4.1 Update Does Not Contribute to View

Example 5.4.1 illustrates this case. We refer to updates that are inferred to not contribute to the view, using the contents of the local relation, as *locally-irrelevant updates*. Local constraint checking methods extend naturally to detecting locally-irrelevant updates. Just as for constraints, first we define the notion of a complete local test for identifying locally-irrelevant updates. A local test for determining that an update does not affect a view is complete if whenever the test fails, then there is a state of the inaccessible relations that along with the updated accessible relation contributes a tuple t to the view such that t was not in the view before the update.

First, we consider views that are defined using conjunctive queries with arithmetic inequalities and we consider only insertions of single tuples into the accessible relation. As in the previous sections, we assume that views have set semantics. We extend the query containment based results of Chapters 2 and 3 to generate complete tests for detecting locally-irrelevant insertions. Insertions are the only relevant update to conjunctive query views. Then we consider techniques that are not complete but apply to a larger class of views. We show the results of Chapter 4 can be used to detect locally-irrelevant insertions and deletions. These tests are not complete.

Complete Local Tests for Detecting Locally-Irrelevant Insertions

EXAMPLE 5.4.2 Consider a variant of view `bad_dept` defined in Example 5.4.1.

$\texttt{bad\_dept}(E, D) \coloneq \texttt{emp}(E, D, S)\ \&\ \texttt{dept}(D, M, MS)\ \&\ S > MS.$

Let relation `emp` be accessible and tuple $\mu = (e, d, s)$ be inserted into `emp`. Following the line of argument in Example 5.4.1 we observe that an `emp` tuple t covers another `emp` tuple μ only if t and μ agree on attributes E and D and if $t.S \geq \mu.S$.

In terms of conjunctive query containment, we can look upon the problem as follows. If we substitute the tuples $\mu = (e, d, s)$ and $t = (e', d', s')$ into the statement of the view, then we get the following two partially instantiated conjunctive queries A_μ and A_t.

A_μ: $\texttt{bad\_dept}(E, D) \coloneq \texttt{dept}(D, M, MS)\ \&\ D = d\ \&\ s > MS\ \&\ E = e.$
A_t: $\texttt{bad\_dept}(E, D) \coloneq \texttt{dept}(D, M, MS)\ \&\ D = d'\ \&\ s' > MS\ \&\ E = e'.$

Tuple t covers tuple μ if $A_t \supseteq A_\mu$. Using the results of Appendix A we see that the above containment reduces to the condition:

$(D = d' \wedge s' > MS \wedge E = e') \Rightarrow (D = d \wedge s > MS \wedge E = e).$

That is, $e = e'$, $d = d'$, and $s \geq s'$.

If attribute E was not distinguished, then the equality on E would not have been forced. However, the equality on D would still have been forced because of the join involving D. Intuitively, all distinguished attributes that appear locally participate in an equality because a cover tuple should contribute the same tuple to the view as the tuple being covered. □

The framework developed in Chapter 2 using query containment is applicable in a straightforward manner to views. In fact, in Chapter 2 we restricted the containment results of Appendix A to 0-ary views to solve the constraint checking problem. Here we consider the more general form as defined in the appendix on Page 98.

$$V: \quad s(\bar{X}) \text{ :- } l(\bar{Y}_0) \ \& \ r_1(\bar{Y}_1) \ \& \ \ldots \ \& \ r_n(\bar{Y}_n) \ \& \ c_1(\bar{Z}_1) \ \& \ \ldots \ \& \ c_k(\bar{Z}_k).$$

For such views, we can extend the containment based result for locally checking constraints stated in Theorem 2.7.3. However, before stating the extension to the theorem we redefine the function Red that was originally defined on Page 29. The redefinition takes into account that $\mathsf{Red}(t, l, V)$ retains all the constants introduced by tuple t that are relevant to the view even though some of the constants may not participate in any built-in comparison.

Definition 5.4.1 ($\mathsf{Red}(t, l, V)$) Consider a conjunctive query (CQ) view V as defined above and let t be a tuple in relation L for predicate l. The *reduction* of V by tuple t, $\mathsf{Red}(t, l, V)$, is the partially instantiated CQ obtained by substituting tuple t for predicate l in C and eliminating l. If an attribute A of l is distinguished then we substitute in the head of the query the constant assigned to A by tuple t. □

Using this new definition of Red, Theorem 2.7.3 generalizes to:

Theorem 5.4.1 *Let V be a view defined using a conjunctive query with arithmetic inequalities and let t be a tuple inserted into relation L for predicate l. Assume V is correctly materialized before the update. View V stays unchanged after inserting t into L if and only if* $\mathsf{Red}(t, l, V)$ *is contained in* $\bigcup_{t_i \ in \ L} \mathsf{Red}(t_i, l, V)$. □

The evaluation issues for detecting locally-irrelevant updates to views are the same as for locally checking constraints, because the condition produced by Theorem 5.4.1 is not affected by the arity of the head of the the rules and thus is similar to the condition produced by Theorem 2.7.3. The only difference from the case of constraints is that more equalities are introduced on the LHS and RHS of the implication condition I_t because now some variables are distinguished.

EXAMPLE 5.4.3 Consider view `bad_dept` defined in Example 5.4.2.

$\texttt{bad\_dept}(E, D)$:– $\texttt{emp}(E, D, S)$ & $\texttt{dept}(D, M, MS)$ & $S > MS$.

Let relation `emp` be accessible and tuple $\mu = (e, d, s)$ be inserted into `emp`. $\mathsf{Red}(\mu, l, V)$ is:

$\texttt{bad\_dept}(e, d)$:– $\texttt{dept}(D, M, MS)$ & $s > MS$ & $D = d$.

If Theorem 5.4.1 is used to derive the test condition for detecting a locally-irrelevant insertion, then we need to check if $\mathsf{Red}(\mu, l, V)$ is contained in $\mathsf{Red}(\alpha, l, V)$ where parameter α represents existing tuples in L. This containment condition is checked using the steps outlined on Page 2.7.3, just as the containment was checked for constraints. □

Detecting Locally-Irrelevant Updates Using Local Tests that are not Complete

We reduce the problem of detecting a locally-irrelevant update to the problem of locally checking an integrity constraint and then the results developed in Chapter 4 can be used.

EXAMPLE 5.4.4 Consider view `bad_dept` defined in Example 5.4.1.

$\texttt{bad\_dept}(D, M)$:– $\texttt{emp}(E, D, S)$ & $\texttt{dept}(D, M, MS)$ & $S > MS$.

If view `bad_dept` is consistently materialized, *i.e.*, correctly computed, then the following constraint assertion holds:

$\mathcal{A}_{ok}$: $\forall E,D,S,M,MS : [(\texttt{emp}(E, D, S) \wedge \texttt{dept}(M, D, MS) \wedge S > MS) \Rightarrow \texttt{bad\_dept}(D, M)]$

Let view `bad_dept` be consistent before an update to relation `emp`, *i.e.*, assertion $\mathcal{A}_{ok}$ holds. If the update to `emp` can be inferred not to affect the truth value of assertion $\mathcal{A}_{ok}$ then we can conclude that the contents of view `bad_dept` do not need to be updated. Local constraint checking methods can be used to check the truth value of assertion $\mathcal{A}_{ok}$ using only the contents of updated relation `emp`, and the update to `emp`. □

In general, consider the view definition V for relation `head`:

V: `head` :– `body`.

For a given database DB, if the view materialization is consistent then the following integrity constraint assertion holds:

$\mathcal{A}$: `body` $\Rightarrow$ `head`.

Let some relation L in `body` be updated. The view relation `head` remains unchanged in the new database state if and only if the assertion $\mathcal{A}$ holds in the new database. Chapters 2, 3, and 4 describe how to use only relation L to check that the update does not violate the constraint $\mathcal{A}$. If the methods determine that the constraint $\mathcal{A}$ continues to hold after the update, then we can infer that the corresponding view does not need to be updated and thus the update is locally-irrelevant.

EXAMPLE 5.4.5 Continuing with Example 5.4.4, assume that only relation `emp` is available for view maintenance. Let tuple (e, d, s) be inserted into `emp`. The update does not contribute to the view `bad_dept` if and only if the update does not violate assertion $\mathcal{A}_{ok}$. That is, if the following query on `emp` derives fact `no_change`:

$$\texttt{no\_change} :- \texttt{emp}(E, D, S) \ \& \ D = d \ \& \ S \geq s \ .$$

When tuple $(john, toy, 50)$ was inserted into relation `emp` then the above query would be satisfied by existing tuple $\texttt{emp}(mary, toy, 65)$ (validating the intuition of Example 5.4.1). □

Locally-irrelevant updates can be detected for all those views that can be reduced to a constraint that is solvable using local constraint checking techniques.

5.4.2 Update Definitely Contributes to View

We prove that it is not possible to infer that an update *definitely* contributes to a view if only a subset of participating base relations is available. Note, this result applies even when relations other than the modified relation are accessible.

Theorem 5.4.2 *Let view V be defined by a single rule r in a Datalog program $\mathcal{D}$ that may use recursion, arithmetic inequalities, and stratified negation. Let program $\mathcal{D}$ use relations $\{L_1, \ldots, L_m, R_1, \ldots, R_n\}, m \geq 1, n \geq 1$ where no R_i is the same as any L_j. Also, rule r uses at least one of the relations $R_1, \ldots, R_n$. Let relation L_u be updated by update $\mathcal{U}$. It is not possible to infer that $\mathcal{U}$ definitely inserts a tuple into view V by any partial-information-based technique that uses only $\mathcal{D}$, $\{L_1, \ldots, L_m\}$, and $\mathcal{U}$.* □

Proof: (By contradiction) We assume that update $\mathcal{U}$ definitely contribute tuple t to the view V and then we prove a contradiction. Consider the following two cases:

1. Some inaccessible relation R_i occurs positively in rule r.

   Consider a database in which relation $R_i = \emptyset$. Thus, $\mathcal{D}$ computes an empty view independent of the accessible relations. Thus, altering some L_j cannot add a tuple to view V.

2. All inaccessible relations that occur in R, occur negatively, and R_i is one of them.

   Consider a database in which relation R_i has all possible tuples that can be obtained using the set of constants in all the accessible relations and the updates. Thus, the subgoal that uses the negated occurrence of R_i in rule r will always derive false, thereby causing $\mathcal{D}$ to compute an empty view independent of the accessible relations. Thus, altering some L_j cannot add a tuple to view V.

Thus, assuming that the update contributes tuple t to the view V leads to a contradiction in either case. ∎

If the contents of the view also can be used as a part of the partial information, then it may be possible to infer the contribution of an update to the view. We do not discuss this problem in this thesis, but we consider it in [GB94]. We illustrate it with an example:

EXAMPLE 5.4.6 Consider the view `bad_dept` defined in Example 5.4.4 and let the view contain tuple $(toy, henry)$. That is, *henry* is a manager in the *toy* department and some employee in the same department has salary greater than *henry*'s salary. Let the relation `dept` contain the tuple $(henry, toy, 500)$. Now, insert tuple $(diana, toy, 400)$ into relation `dept`. We can infer that the tuple $(toy, diana)$ needs to be inserted into the view because the inserted `dept` tuple covers an existing `dept` tuple that contributed to the view. □

Chapter 6

Constraint Checking with No Data

Two other instances of constraint checking with partial information are discussed in this thesis. Both of these instances use less information than local checking.

Chapter Outline

First, in Section 6.1 we discuss constraint subsumption, *i.e.*, how to check one constraint using a set of other constraint specifications and no data. In Section 6.2 we consider irrelevant updates, that is checking constraints using only the database update and a set of other constraints. Finally, Section 6.3 describes the relationship of our work with existing work on constraint checking and view maintenance in relational databases.

6.1 Constraint Subsumption

Constraint subsumption involves checking a constraint using only the constraint specifications and not even the update. This problem is not applicable if the database has only a single constraint because we cannot check if an update violates a constraint without at least examining the update. However, in a database with multiple constraints, the validity of one constraint C_1 may imply the validity of another constraint C_2; in this case, C_2 need not be checked when it is known that C_1 is not violated. Since we assume that all constraints must hold after all updates, if C_1 implies (or *subsumes*) C_2 then C_2 can be ignored altogether. Furthermore, since subsumption of C_2 by C_1 is independent of data or database updates, the subsumption can be inferred or checked once, at constraint-definition time.

Definition 6.1.1 (**Constraint Subsumption**) Let C be a constraint query and $\mathbf{C} = \{C_1, \ldots, C_m\}$ be a set of constraint queries. We say that $\mathbf{C}$ *subsumes* C if whenever C is violated in a database D then some C_i in $\mathbf{C}$ is also violated in D. □

Since constraint queries only produce $\{\texttt{panic}\}$ or $\emptyset$ as a result, subsumption is a special case of containment of programs. Then the following is obvious:

Theorem 6.1.1 *Constraint set* $\mathbf{C} = \{C_1, \ldots, C_m\}$ *subsumes constraint* C *if and only if, viewed as programs,* $C \subseteq C_1 \cup \cdots \cup C_m$. □

Proof: Suppose $\mathbf{C}$ subsumes C. We need to show that $C \subseteq \mathbf{C}$, *i.e.*, we must show that for any database, anything in the result of C is in the result of $\mathbf{C}$. The result of the set of programs $\mathbf{C}$ is the union of the results derived by the programs in the set. Each of these programs derives either empty or **panic**. Thus the result of $\mathbf{C}$ also is either empty or **panic**. By Definition 6.1.1, if the result of C is `panic` then the result of $\mathbf{C}$ also is `panic`.

Conversely, suppose $\mathbf{C} \subseteq C$. We need to show C subsumes $\mathbf{C}$, *i.e.*, we must show that for any database, if $\mathbf{C}$ satisfies `panic` then C also satisfies `panic`. This follows directly from $\mathbf{C} \subseteq C$. ∎

There are many known results about program containment that apply directly to constraint subsumption. For example, if the constraints are CQ's or unions of CQ's, the problem is NP-complete [Sar90]. If the CQ's use arithmetic inequalities as subgoals or if the subsuming constraint is a recursive Datalog program, then the problem is still solvable in exponential time. The former case is Π_2^p-complete [Klu88, Mey92] and the latter is exponential-time-complete [CLM81, Sag88].

If we allow the subsumed constraint to be a recursive Datalog program, while the subsuming constraints are nonrecursive Datalog, the problem remains decidable [Cou91]. The complexity of this problem was resolved in [Cha92], who showed it is complete for triply exponential time, with some less complex special cases. On the other hand, when both the subsuming and subsumed constraints are recursive Datalog, the problem becomes undecidable [Shm87].

Let us pictorially discuss subsumption using Figure 2.1 introduced in Chapter 2. If the constraint queries are all the information that is available, then the circle of illegal states S for C represents all the possible database instances for which the constraint-violation condition C derives `panic`. Similarly, for each constraint C_i in $\mathbf{C}$ there is a circle S_i representing the databases for which C_i derives `panic`, as illustrated in Figure 6.1. Thus, if we can infer that the union of circles $S_1 \cup \ldots \cup S_m$ contains circle S, then we are guaranteed that every database that violates C also violates some constraint in $\mathbf{C}$. Thus $\mathbf{C}$ subsumes C and C need not be checked if the constraints in $\mathbf{C}$ are enforced.

6.1.1 Containment Versus Constraint Subsumption

Since constraint queries have a 0-ary goal predicate the question that arises is whether checking constraint subsumption is easier than checking query containment. It turns out that for a large class of queries the two problems are equally hard, that is, constraint subsumption is just as hard as the corresponding query containment problem.

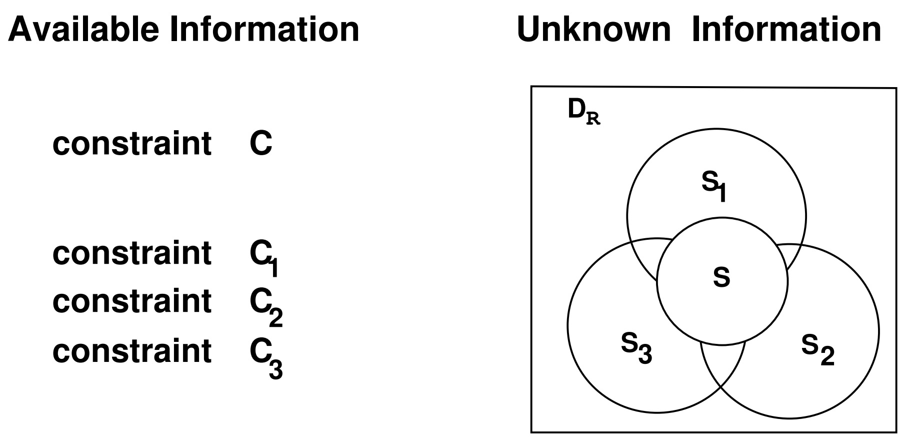

Figure 6.1: Pictorial Representation of Subsumption

First let us consider only conjunctive queries. The NP-completeness of containment for CQ's with 0-ary heads was proved by Chandra and Merlin in [CM77]. The containment of conjunctive queries is also NP-complete. In general, we can reduce CQ containment to constraint subsumption in a very robust way. If Q is a CQ of the form h `:-` B, we rename the predicate of the head h if it appears in the body B. We then "move" the head into the body, creating the CQ Q' that is

`panic :-` h & B.

If Q and R are two CQ's, it is easy to check that $Q \subseteq R$ if and only if $Q' \subseteq R'$. Thus, we can claim the following:

Theorem 6.1.2 *For any class of CQ's that is closed under the addition of a subgoal that is of the ordinary type (uninterpreted predicate with arguments, not negated), the containment problem reduces to the corresponding constraint subsumption problem in time proportional to the size of the query.* □

Proof: By the construction given above. ∎

Thus for conjunctive query constraints, checking subsumption and checking containment are equally hard problems. How about generalizations of CQ's? For slightly more powerful query languages, like Datalog without negation, we do not know if subsumption and containment are equally hard. For query languages that are even more powerful, where intermediate predicates are allowed, and negation may be used (e.g., nonrecursive Datalog with stratified negation), it is easy to reduce query containment to the corresponding constraint subsumption problem by adding rules, thus providing a lower bound on the complexity of constraint subsumption for these classes. We have not explored the boundaries where query containment stops being equivalent to subsumption, if that is indeed the case, or the point where the equivalence resumes.

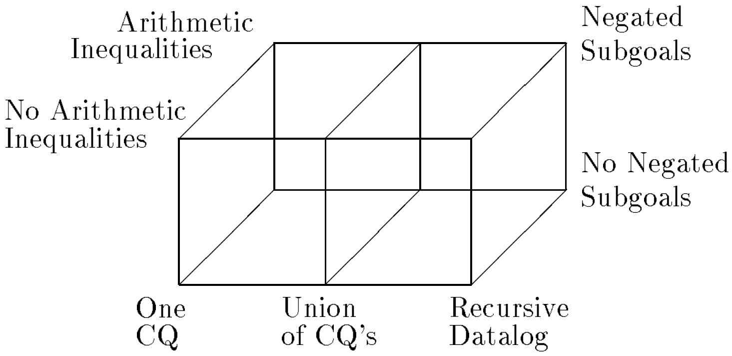

Figure 6.2: Classes of logical languages

6.2 Using the Update

The second problem we consider is when we are allowed to use a collection of constraints **C** and an update u to determine that another constraint C is not violated. This problem has been studied in [BC79, TB88, Elk90, LS93] as the "query independent of update problem." Only the update and the expression of the query are needed to make this decision. The papers describe how to decide if an update could affect the answer computed by a query, for queries expressed as CQs, CQs with arithmetic, and CQs with negated EDB subgoals. The papers also consider updates that change not only the underlying database, but the query itself. The focus of the papers is to develop algorithms for detecting irrelevant updates for progressively more expressive query languages.

In this section, we study a particular way of detecting irrelevant updates, originally introduced in [LS93]. The basic strategy for identifying an irrelevant update is to first incorporate explicitly update u into the constraint program C to yield a program C^u that holds before the update if and only if C holds after the update. The test for whether C holds after the update, given that it and perhaps some other constraints $C_1, \ldots, C_m$ held before the update, is to see whether C^u is contained in $C \cup C_1 \cup \cdots \cup C_m$. If constraints $C, C_1, \ldots, C_m$ held before the update, then we can infer that $C \cup C_1 \cup \cdots \cup C_m$ is empty. If C^u is contained in $C \cup C_1 \cup \cdots \cup C_m$, then we conclude that C^u also does not derive any fact, in particular `panic`. Thus, C^u holds before the update and therefore C continues to hold after the update.

When we construct C^u from C, it may not be guaranteed that C^u is expressible in the same constraint language as C. We consider the twelve combinations of features for conjunctive query constraints illustrated in Figure 6.2 to study the relationship between C and C^u. For these classes first we study how incorporating insertions takes a constraint from one of the above twelve classes into some other class. Then, in Section 6.2.2 we study the effect of incorporating deletions. Insertions and deletions are considered separately because insertions are incorporated into C differently from deletions and as a consequence it may be the case that the two different kinds of updates affect the class of C^u differently. We do not consider the problem of checking the containment of C^u in $C \cup C_1 \cup \cdots \cup C_m$. That problem is discussed in [LS93, ZO93] and in Appendix A.

6.2.1 Incorporating Insertions into Constraints

The following example illustrates how constraints can be modified to account for insertions and also that different languages may be needed to express C and C^u.

EXAMPLE 6.2.1 Consider the employee relation `emp` and consider the relation $\texttt{tall}(E)$ where $\texttt{tall}(e)$ means that employee e is tall. Now consider the following constraint:

A: $\texttt{panic} \text{ :- } \texttt{emp}(E, D, S)\ \&\ \texttt{tall}(E)$.

That is, if some employee in relation `emp` is tall, then constraint A is violated. This constraint is a conjunctive query that does not use arithmetic or negation.

Let a tuple $(jones, shoe, 500)$ be inserted into `emp`. Constraint A can be rewritten as A^u after incorporating the inserted tuple into A.

A^u: $\texttt{panic} \text{ :- } \texttt{emp1}(E, D, S)\ \&\ \texttt{tall}(E)$
$\texttt{emp1}(E, D, S) \text{ :- } \texttt{emp}(E, D, S)$
$\texttt{emp1}(jones, shoe, 500)$.

A^u derives `panic` with a database DB if and only if A derives `panic` with $DB \cup \{\texttt{emp}(jones, shoe, 500)\}$. Note, even though A was in the language of conjunctive queries, A^u is expressed as a union of conjunctive queries. □

The above example shows how to incorporate an insertion to a positive subgoal in a constraint query. Intuitively, an additional fact contributes an alternative way of satisfying subgoals and thus leads to disjunctions in rules. These disjunctions can be factored out using additional rules. The example also illustrates that the language needed to express the resulting constraint may be more expressive than the original language. Thus, the question arises as to whether C^u has to necessarily be in a more expressive language than C. The following theorem proves that for a large class of constraints, we cannot avoid going to a more expressive language.

Theorem 6.2.1 *Consider a constraint A defined using a single conjunctive query (possibly using arithmetic and negation) such that the predicate p occurs positively and only once in A, and the number of p tuples that can satisfy A is not bounded a priori. Let t represent a tuple inserted into relation P such that t is not irrelevant with respect to A. Let A^u be the constraint A with inserted tuple t incorporated into A as shown above. Hence, A^u derives* `panic` *with DB if and only if A derives* `panic` *with $DB \cup \{p(t)\}$. Constraint A^u cannot be expressed as a single conjunctive query even if the query uses arithmetic inequalities and negation.* □

Proof: In Appendix D. ∎

The above theorem proves that for each of the four uncircled classes some constraints necessarily move to one of the circled classes after incorporating the insertion into the constraint. Thus none of the four uncircled classes of constraints in Figure 6.3 is preserved in the presence of insertions.

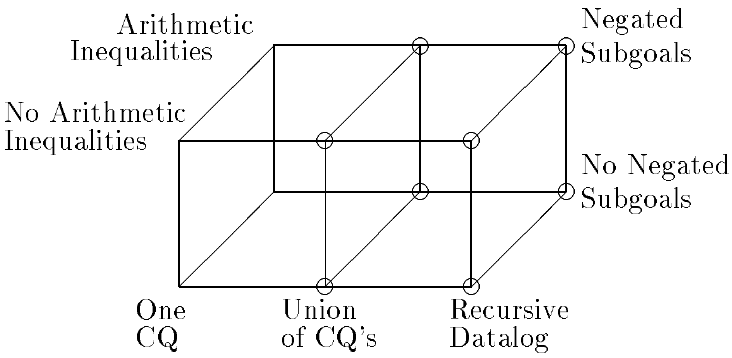

Figure 6.3: Classes preserved under insertion

The above theorem considers insertions only to the positive subgoal. How about insertions to negatively occurring subgoals?

EXAMPLE 6.2.2 Consider relation `emp` defined in Chapter 1, and a relation `insured`(E) such that `insured`(e) says that employee e has health insurance. Consider a constraint $I8$ that is violated if any employee in relation `emp` is not in relation `insured`. Constraint $I8$ is represented by the following rule:

`panic` :– `emp`(E, D, S) & *not* `insured`(E).

We assume constraint $I8$ holds before the update. Suppose there is an update in which *tom* is added to the set of insured employees. We can define a constraint that represents $I8$ after the update as

`panic` :– `emp`(E, D, S) & *not* `insured1`(E).
`insured1`(E) :– `insured`(E).
`insured1`(tom).

Call this constraint $I9$. $I9$ is in the language of nonrecursive Datalog with negation, even though the constraint $I8$ from which it was derived is in the narrower class of conjunctive queries with negation. Then in order to be sure that $I8$ has not become violated by the update we need to check $I9 \subseteq I8$. This happens to be the case and we can conclude that $I8$ is not violated by the update.

Note, $I9$ is in the language of "union of CQ's with negation but no arithmetic inequalities." Another way to express $I9$ is by the single rule

$\texttt{panic} :- \texttt{emp}(E, D, S)$ & *not* $\texttt{insured}(E)$ & $E \neq tom$.

Now, the constraint is in the language of "CQ's with both negation and arithmetic inequalities." □

For the above example we prove that it is not possible to express the resulting constraint in a simpler language that those used above.

Theorem 6.2.2 *Constraint I9, stating that after insertion of "tom" into relation* `insured` *there is no employee in a department that does not appear in* `insured`*, cannot be expressed as a single CQ (over the predicates* `emp` *and* `insured` *denoting their values before insertion) without arithmetic inequalities, even if negation is allowed.* □

Proof: Appears in Appendix D. ∎

Now, let us consider the eight circled classes in Figure 6.3. The technique used to incorporate an insertion into a relation as illustrated in Example 6.2.1 works for each of these classes. The technique involved converting an EDB relation into an IDB using extra rules. Any language that allows us to add rules, even nonrecursive ones and rules without negation or arithmetic inequalities, allows us to express a constraint after an insertion in the same language. We thus claim

Theorem 6.2.3 *The eight circled classes in Figure 6.3 are preserved by insertions; that is, a constraint in the class after an insertion can be expressed in the same language.* □

Proof: Let tuple $r(\bar{u})$ be inserted into relation R that occurs in constraint C. Constraint C^u is obtained by replacing all occurrences of the predicate r in C by a new predicate $r1$ and defining $r1$ as follows:

$r1(\bar{X}) :- r(\bar{X})$.
$r1(\bar{X}) :- r(\bar{u})$.

The only change to the original constraint is the additional of a rule. This change keeps C^u in the same class as C for all the circled classes in Figure 6.3. ∎

6.2.2 Incorporating Deletions into Constraints

In this section we study the effect of deletions on the expressive power of the language needed to express a constraint after the deletion has been incorporated. The following example illustrates the technique.

EXAMPLE 6.2.3 Continuing with Example 6.2.2, say we delete tuple $(jones, shoe, 500)$ from the `emp` relation. Then we need to construct a new predicate `emp1` that reflects the deletion of this tuple. Here is one way to do so.

$\texttt{emp1}(E, D, S) :- \texttt{emp}(E, D, S)$ & $E \neq jones$.
$\texttt{emp1}(E, D, S) :- \texttt{emp}(E, D, S)$ & $D \neq shoe$.

emp1(E, D, S) :– emp(E, D, S) & $S \neq 500$.

The predicate emp1 can substitute for emp in constraint $I8$ to create a new constraint $I8$' that reflects the situation after this deletion. Note that in this construction, conjunctive queries are brought into the class of nonrecursive Datalog with arithmetic inequalities. □

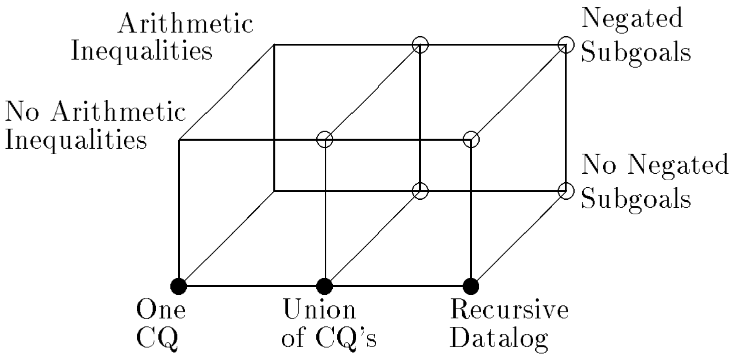

Figure 6.4: Classes Preserved Under Deletion

Alternatively, we could use negated subgoals instead of arithmetic inequalities in each of the above three rules. For instance, we could replace the subgoal $E \neq jones$ in the first rule of Example 6.2.3 by *not* $isJones(E)$, where predicate $isJones$ is defined by

$isJones(jones)$.

It does not appear to be possible to avoid using one of negation and arithmetic inequalities. Note, if two tuples are deleted from emp instead of one tuple, then two defined intermediate predicates, emp1 and emp2, are needed to capture the two deletions.

Now we consider different constraint classes and study the effect of deletions on the language needed to express the resulting constraint.

For Figure 6.4, we illustrate via examples that the three uncircled classes and the classes indicated by the solid disks may not be preserved when deletions are incorporated into the constraint. First, we consider CQs, union of CQs, and recursive Datalog constraints that do not use negation or arithmetic. We prove that deletions may take such constraints out of their original class. These classes are indicated by the solid disks. Consider constraint A introduced in Example 6.2.1:

A: panic :– emp(E, D, S) & tall(E).

If tuple tall$(mary)$ is deleted from relation tall then constraint A can be rewritten as:

A^u: panic :– emp(E, D, S) & tall1(E)
tall1(E) :– tall(E) & $E \neq mary$.

We prove that negation or arithmetic inequalities have to be used to express constraint A^u.

Theorem 6.2.4 *Consider constraint A^u that states that no employee in* `emp` *is in* `tall` *after "mary" is deleted from* `tall`. *There is no constraint C that is equivalent to A^u such that C uses neither negation nor arithmetic.* □

Proof: Appears in Appendix D. ∎

The above example can be modified to prove that union of CQs and recursive Datalog programs need negation or inequalities when incorporating deletion. To build the required examples, we replace the EDB relation `emp` by an IDB relation that is defined using unions of CQs and recursive Datalog, respectively. The proof of Theorem 6.2.4 does not depend on the definition of `emp` thereby allowing us to claim that constraints defined using union of CQs and recursive Datalog programs may need negation or arithmetic when deletions are incorporated into the constraint. Thus, we have shown that there exist instances of the three classes indicated by the solid discs that are not preserved by deletions.

Next we consider constraints that use negation. Constraint $I8$ is an instance of such a constraint.

`panic` :– `emp`(E, D, S) & *not* `insured`(E).

Deleting a tuple from a negatively occurring predicate is similar to inserting a tuple in a positively occurring predicate. Thus, if tuple `insured`$(john)$ is deleted then a union of CQs should be required for expressing $I8$ after the deletion. The intuitive argument for this claim is that given the deleted tuple, $I8$ is violated if $john$ is in relation `emp` or if some other employee is not insured. This new constraint is therefore a union of the old constraint and another condition, and should need two rules. Thus, we claim:

Theorem 6.2.5 *Constraint I8, after deleting tuple "john" from relation* `insured`, *cannot be expressed as a single CQ constraint that uses negation.* □

Proof: In Appendix D. ∎

Thus, of the three unmarked classes of Figure 6.4, only two remain to be considered. The remaining classes are CQ constraints that use arithmetic inequalities. For these classes, if deletions are made only to monadic predicates, *i.e.* predicates of arity one, we can prove that the class of CQ constraints with arithmetic inequalities is preserved under deletions.

Theorem 6.2.6 *Let C be a CQ constraint that may use arithmetic inequalities such that deletions are made only to monadic predicates. The constraint representing C after a tuple has been deleted from any participating relation, can be expressed using a single CQ constraint with arithmetic inequalities.* □

Proof: (By Construction) Consider a monadic predicate p in constraint C and let tuple $p(a)$ be deleted from relation P. From the statement of the theorem we know that C does not involve negation and thus p

occurs positively. Replacing "$p(X)$" by "$p(X)$ & $X \neq a$" incorporates the deletion of $p(a)$. The constraint is in the class of CQs with arithmetic inequalities. ∎

If the deletions are made to predicates of arity more than one, then we conjecture, without proof, that even simple CQ constraints with arithmetic need unions of CQs to express the constraints after deletions.

After considering classes that are *not* preserved under deletions, we now state a theorem that addresses the six circled classes of Figure 6.4 that are indeed preserved.

Theorem 6.2.7 *The six classes circled in Figure 6.4 can express constraints that result from a deletion.* □

Proof: Just as for Theorem 6.2.3, the construction of C^u uses only the features in the class of C. For deletions, negation and inequalities are the only features that are needed to build C^u from C as illustrated in Example 6.2.3. All the circled classes in Figure 6.4 include one of these two features. ∎

6.3 Related Work

Many different types of partial information based view maintenance techniques have been previously studied. Some types of partial information have not been considered in this thesis, for instance, aggregate information about relations. The techniques have been proposed for view maintenance and therefore apply to constraint checking because constraints can be expressed as views. In this section, first we compare these different techniques with our work. In each case, we assume that the view definition and the update are used in addition to the information listed explicitly in each item. After we describe the existing results, we discuss original contributions of our work.

- *No additional partial information is used.* This problem has been called the "query independent of update" or "irrelevant update" problem in [BC79, TB88, Elk90, LS93]. This approach infers when a view remains unchanged after an update, independent of the underlying database. This approach is potentially the most economical way of checking if an update affects a view and thus could be the first step in a system for view maintenance. Note, the results apply to general views and thus also carry over to constraints. The above cited papers explore various classes of views for which irrelevant updates can be detected.

  In Section 6.2 we studied an orthogonal aspect of this problem restricting ourselves only to constraints. We use a reduction of the irrelevant update problem to the query containment problem to study the relationship between the class of the original constraint and the class of the resulting query for which containment needs to be checked. Our results also apply to general n-ary view definitions.

- *The partial information consists of derived aggregate information about the base relations.* This problem is considered in [BBC80]. [BGM92] stores extra information about relations that is not aggregate information, but is derived from the relation by a user. However, we refer to it as aggregate information for the purpose of the following discussion. In this thesis we do not consider aggregate information as a kind of partial information. Such techniques are useful when the extra information is easily maintained and saves accesses to more expensive information. The motivation is exactly the same as the work done in this thesis, and the results complement our work.

  Specifically, [BGM92] considers the distributed database scenario and considers only constraints on atomic variables. For such constraints [BGM92] stores extra information about remote variables and uses this information to infer that changes to the local variables do not affect the constraint. [BBC80] uses extra aggregate information (maxima and minima) about the database to produce sufficient tests for constraints that are expressed by tuple calculus formulas involving two tuple variables. Aggregates are computed and stored for both the relations involved in a view or constraint. Thus, it may be possible to check a constraint using the stored aggregate information about the remote relation even though it may not be possible to check the constraint using only the local modified relation. The techniques developed in [BBC80] apply to a very limited language, that using only two relations. Our techniques generalize the results of [BBC80] to identify the aggregate functions that need to be computed and stored, for a much larger classes of views than those considered in [BBC80].

- *The partial information consists of only the old contents of the materialized view.* This problem was first introduced in [TB88] and the paper considered views defined using conjunctive queries with arithmetic inequalities. However, the algorithm in [TB88] to update a view using only the original contents of the view is erroneous. [GB94] reconsiders the problem for the same class of views. The paper also proposes a way to use an arbitrary subset of the base relations in addition to the old contents of the materialized view ([TB88] does not consider using any base relations.)

  In the context of constraints, local checking uses the contents of the view by using the initial consistency assumption that implies that the original view is empty. That is, local checking corresponds to using the available local relation in addition to the contents of the view. Thus, the generalizations discussed in [GB94] that update views using the old materialized view and an arbitrary subset of the base relations should apply to local checking also. [GB94] compares the approach developed in this thesis with the approach used in [GB94] for conjunctive query constraints that use arithmetic inequalities.

- *The partial information consists of all the base relations and the old contents of the materialized view.* The available information is no longer "partial" but indeed the information is sufficient to compute the required answer. This problem is known as the "incremental view maintenance" problem and has been studied from the perspective of accessing as little of the database as possible. These techniques use the heuristic of inertia, *i.e.* only a part of the view changes in response to changes in the base relations. Therefore, often it is cheaper to compute only the changes in the view instead of recomputing the view from scratch by using the update to prune the amount of "relevant" information. Thus, the issue is to use this information in as clever a fashion as possible by avoiding redundant derivations and unnecessary database accesses. The problem has been studied in [BB82, BCL89, BLT86, Bla81, BMM92, CW90, CW91, DS92, GMS93, HD92, KSS87, Kuc91, NY83, QW91, SI84, UO92, WDSY91].

In the context of integrity constraints, [BMM92, KSS87, LST87, Nic82] discuss ways to incorporate the update made to the database into the constraint to make the checking phase more efficient. Just as for view maintenance, the idea is to reduce the amount of data read from the database during checking. They use the initial consistency assumption to simplify the expression that needs to be evaluated. That is, the techniques use the fact that the original view is empty to simplify the checks that need to be executed. These papers can be considered to provide ways of simplifying the incremental view maintenance problem for the case when the views are 0-ary and when the initial view is empty. Several of these papers also state general view maintenance algorithms on the way to developing more efficient algorithms for constraint checking.

Solutions to constraint checking discussed in the above cited papers correspond to the "brute-force" approach in our framework, where all the underlying relations are used. The results from the afore mentioned papers can be used for constraint checking when a potentially cheaper partial information based technique fails to compute the answer.

The idea of using only the update, or only the contents of the view, for view maintenance and constraint checking has been considered before. This thesis introduces the notion that these, and other, kinds of partial information are actually part of a spectrum of options for developing efficient ways to do view maintenance and constraint checking. We introduce a yet unconsidered instance of the partial information, namely the updated relation, and explore its application for constraint checking. For some classes of constraints we develop complete local constraint checking methods, while for other classes we develop only sufficient local methods. We also extend these results to detecting locally irrelevant updates.

Chapter 7

Conclusions

In this chapter we briefly summarize the contributions of this thesis and discuss the extensions and other resulting ideas that we are currently exploring. We also describe the prototype Distributed Constraint Management System (DCMS) we have participated in implementing.

7.1 Contributions

In this thesis we described strategies for checking integrity constraints while restricting the amount of information used for the checking process. Such an approach is especially advantageous to avoid using information that is expensive to access or unavailable. Different combinations of available information can be used for checking constraints depending on the particular constraint and the update. We considered three instances of partial information and discussed how to check different classes of constraints using the information.

Local constraint checking That is, checking a constraint using the update to the database, the contents of the updated relation, the constraint specification, and the initial consistency assumption. Figure 1.1 outlines our approach to investigating local checking.

- We considered conjunctive query constraints that use arithmetic inequalities (CQCs) and showed how to reduce local checking to a conjunctive query containment problem. (Chapter 2)
- We identified subclasses of CQCs for which constraints can be checked locally by posing a Datalog query to the updated relation. The Datalog query is a complete local check. (Chapter 3)
- We identified subclasses of CQCs for which the complete local checks are expressible using unions of conjunctive queries. (Chapter 3)

- We studied local checking methods that are not complete. We tradeoff completeness in favor of deriving sufficient local tests for a larger class of constraints than conjunctive query constraints with arithmetic inequalities, for instance, constraints that use restricted negation. For this larger class we developed algorithms that generate sufficient test conditions on the updated local relation. (Chapter 4)

- We considered properties and extensions of local checking methods. (Chapter 5)

  - We discussed how to locally check constraints when tuples are deleted from the local relation. The constraint language of Chapter 4 is used in the discussion because that language has negation and thus deletions become relevant.
  - We discussed how to efficiently locally check modifications to tuples.
  - We discussed the correctness of applying several local tests in parallel when the tests check the same constraint but treat different relations as local.
  - We discussed how to extend local checking developed in the context of constraints, to updating materialized views using only the updated relation, the update to the database, and the view definition.

Subsumption and **Irrelevant Updates** We considered two other instances of partial information. (Chapter 6)

- Subsumption uses a set of constraints $\mathbf{C}$ to check another constraint C. If the set $\mathbf{C}$ subsumes constraint C, then checking $\mathbf{C}$ obviates the need to check C.

- Irrelevant updates [BC79, TB88, Elk90, LS93] use only the update and the constraint C to check C. We discussed how to incorporate the update into constraint C and then check if the resulting constraint is contained in the original constraint. We identified how the expressive power of the language needed to express the resulting constraint changes when the update is incorporated.

7.2 Architecture of DCMS

In this section we briefly discuss the architecture of the *Distributed Constraint Management System* (DCMS) we have built, shown in Figure 7.1. [Tiw94] contains a detailed discussion of the design and implementation of the system. DCMS was developed as a part of the "Collaborative Environment for Designing Buildings" project. An outline of the project appears in [HKG+94]. In this section, we will briefly touch upon the DCMS system and its components.

Interacting with the DCMS are applications that do domain-specific reasoning and analysis and in the process alter the database objects comprising the design. The *design cache manager* at

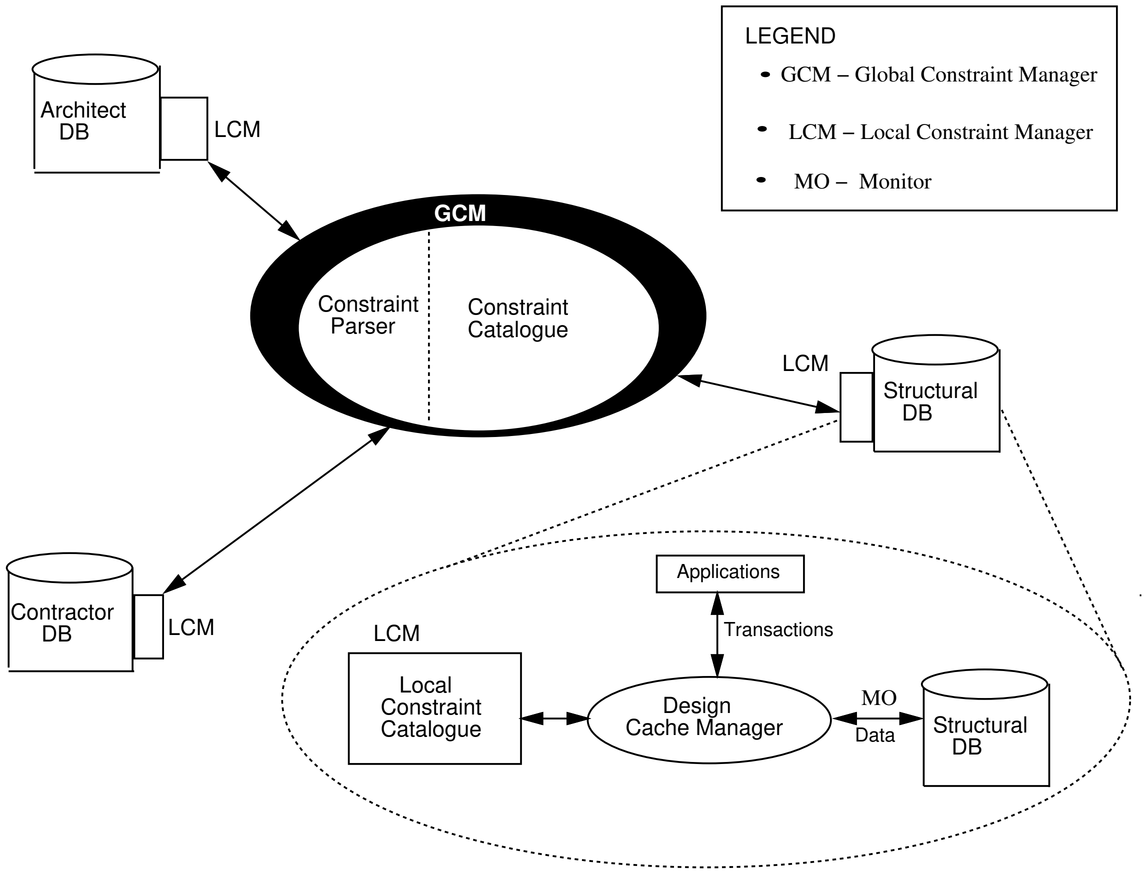

Figure 7.1: The Distributed Constraint Management System

each site supports interaction of the applications with design databases. The design cache manager allows checkin-checkout transactions and keeps track of the changes made to the design [KL94]. Our prototype is built assuming that the databases are relational.

Our focus is on managing the constraints that regulate the design changes made by applications. In our system, constraints are specified in an SQL-like language that is a variant of the language proposed in [CW90], extended to express constraints on multiple autonomous databases. Constraints are specified as inconsistent design states, *i.e.* as constraint-violation conditions. The following example illustrates a constraint specified in our system.

EXAMPLE 7.2.1 Consider a construction scenario and a relation from the contractor's database:

`crane`$(Crane\_id, Floor\_id, Capacity)$ *% location and lifting capacity of cranes.*

Now consider a relation in the structural designer's database:

`column`$(Column\_id, Floor\_id, Weight)$ *% location, weight of concrete support columns.*

Consider constraint C_g that requires that every floor that appears in relation `column` should have at least one crane that has the capacity to lift the heaviest column on that floor. The violation condition for C_g is expressed as follows. Structural::`Columns` refers to the `Columns` relation on the Structural Designer's site.

Structural::`Column`.*Weight* $\geq$ all (**select** `Crane`.*Capacity*
from Contractor::`Crane`
where `Cranes`.*Floor_Id*$=$`Column`.*Floor_Id*)

actions: Notify(Structural Designer, Contractor, Project Manager);

The above specification says that constraint C_g is violated if there is a column whose weight is greater than the capacity of every crane on that floor. If such a column exists then the structural designer, contractor, and project manager should be notified. □

The constraints may span multiple participant databases and are specified at, preprocessed by, and stored by the *Global Constraint Manager* (GCM). From the high-level specification of constraints, the GCM extracts at compile time information needed for constraint checking. The information derived at compile time includes:

- *Potentially invalidating operations, i.e., operations on the database objects that might violate a constraint [CW90].* For the example constraint, the set of potentially invalidating operations are:

  Designer Site: Insert(Designer::`column`),
  update(`column`.*Weight*), update(`column`.*Floor_Id*).
  Contractor Site: Delete(Contractor::`cranes`),
  update(`cranes`.*Capacity*), update(`cranes`.*Floor_Id*).

- *Local tests that can check a constraint for some potentially invalidating operations.* For instance, consider constraint C_g and suppose the designer adds a new column col_1 of weight 10 tons on floor fl_1. We need to ensure that there is a crane on floor fl_1 that can lift column col_1. Suppose floor fl_1 already had a column weighing 12 tons. Assuming that C_g was satisfied before adding col_1, we can infer that floor fl_1 has a crane with capacity of at least 12 tons and thus adding col_1 of weight 10 tons will not violate C_g. Note, the remote relation *crane* on the contractors site was not read for the above check.

- *The global query that checks a constraint when the local tests fail.* For the above constraint, the query evaluates the violation condition.

  select *
  from Structural::`Columns`
  where *Weight* $\geq$ all (**select** `Crane`.*Capacity*
  from Contractor::`Cranes`
  where `Cranes`.*Floor_Id*$=$`Columns`.*Floor_Id*)

Currently the system has the following functionality. We emulate multiple sites using multiple table spaces in a single site database; henceforth "sites" refer to "table spaces." Thus, there is a single user who forces constraint checking after making some changes. We envision that eventually the constraint checking will be initiated by the design cache manager. The GCM derives optimization information, and distributes this derived information to the participating sites. The derived information currently includes local tests, potentially invalidating operations, and the information to send to the GCM in case the local check fails. At individual sites the *Local Constraint Managers* (LCM) are responsible for constraint checking. The LCMs contain most of the run-time constraint checking machinery and store the site specific information derived by the GCM at compile-time. The information derived by the GCM is stored by the GCM and the LCMs in *persistent catalogs* (implemented as database tables). Catalogs support efficient constraint checking by providing the GCM and LCMs fast access at run-time to the information obtained at compile-time. Catalogs also provide a facility for querying constraints at the local sites. We allow queries like "which constraints may be violated if a column is inserted?" Currently, we *Monitor* the database using triggers in Oracle.

We have designed, but not implemented, fragmentation methods for fragmenting global constraints into database specific local components that can be evaluated efficiently locally at the participating sites. For instance, if a constraint refers to two relation from site 1, then it may be economical to compute the join of the relations on site 1 and shipping the result to the GCM instead of shipping the two relations separately. Distributed query optimizer research addresses these issues. Fragmentation may reduce the amount of information that local sites need to communicate to the GCM for constraint checking.

Now we briefly discuss the behavior of the system at runtime. Given an update, the LCM uses the local test stored in its constraint catalog to check if the update violates any constraint. If the local tests fail, then the GCM is informed about the updates. A global query – stored in the global constraint catalog – is initiated to check the constraint globally. In the current system we always run the local tests assuming that the database is initially consistent. This assumption may not be true always in a real database system. In case the initial consistency assumption does not hold, we could either use the idea of flagging tuples as outlined in Section 5.3 or not use local checking for constraints that are known to not hold.

Notification is the final phase of constraint management, activated only if a constraint is violated. Often notifications need to be issued to select participants who are responsible for resolving the design inconsistency arising from the constraint violation. Thus all the participants in a constraint are not "equal." This asymmetric participation of sites is modeled by listing the participants in order of their responsibility in the constraint specification. For instance, the notification list of constraint C_g may have only the contractor in which case only the contractor is informed of violations. Conversely, the notification list may have more participants listed than are involved in the constraint. If a notification list is not provided then notifications are broadcast to all the sites referred by the constraint. In the current implementation, notifications consist of messages that contain the ID of the violated constraint.

As another part of the same project, there is work being done on efficient change management and the design cache manager [KL94]. We briefly discuss some of the issues being addressed in this effort. Note, in Figure 7.1 the cache manager monitors the database relations to detect the updates that potentially violate any constraint. The cache manager should compute net changes made to the database and thereby avoids redundant checks; for example, a deleted object that is reinserted should not be considered as having changed. Only the relevant changes should be sent to the LCM. Also, the notion of transactions is not well defined in the design domain. Therefore, the research is also addressing the issue of when constraint checking should be initiated.

7.3 Future Work

We outline three main lines of future research. Two stem from the idea of using partial information based techniques. The first direction to pursue is to enhance the kinds of partial information that can be used and to do so for more expressive constraints and views than those considered in this thesis. The second direction is to apply the local checking techniques developed in this thesis to other problems. Finally, we point out some problems that came up while building the DCMS system built to demonstrate the usefulness of constraints in the engineering design domain.

Using Different Kinds of Partial Information

May different kinds of partial information can be used for updating materialized views and checking constraints, for instance, multiple base relations, the old contents of a view, functional dependencies, and aggregate information. In [GB94] we discuss how to use multiple base relations and the old contents of a view. The paper addresses incremental view maintenance of views defined using conjunctive queries and arithmetic inequalities when the partial information consists of the old contents of the materialized view and an arbitrary subset of the underlying base relations. The paper also extends the techniques to constraint checking with the same amount of information. However, there remain many extensions like using functional dependencies, aggregate information, flags on tuples, *etc.*

Another extension is to determine test conditions for more expressive constraint and view languages. We are currently considering constraints that use functions like volume, distance, and other complicated mathematical functions [HG95]. In this context we study the problem of determining when a tuple covers another tuple with respect to a constraint that uses mathematical functions.

Recall, the cover tuple property allows us to locally check an inserted tuple if there is an existing tuple that covers the inserted tuple. One of the interesting aspects of the study is identifying the conditions under which the complete covering condition involves the inserted tuple and just one tuple from the available relation. This case is interesting because such cases are much more efficient to check than if an arbitrary number of existing tuples could together cover the inserted tuple. The issue of single versus multiple cover tuples is covered in more detail in Section 3.3 and is illustrated using the forbidden interval example.

Using Local Checking for Inferring Representative Relations

The following example illustrates application of local checking techniques to other problems.

EXAMPLE 7.3.1 Consider a multisite database with relations `emp` and `dept` as described in Example 1.0.1. Suppose relation `emp` is on site 1 and `dept` is on site 2. Let site 2 use view `good_dept` defined as follows:

$$C\text{:}\quad \texttt{good\_dept}(D) \text{ :- } \texttt{emp}(E, D, S) \;\&\; \texttt{dept}(D, MS) \;\&\; S \geq MS.$$

Suppose site 2 replicates the relation `emp` for the purpose of answering queries for view `good_dept`. If the relation `emp` has tuples `emp`$(john, toy, 100)$ and `emp`$(mary, toy, 50)$ then both tuples need not be cached in order to answer queries on view `good_dept`. Tuple `emp`$(mary, toy, 50)$ is covered by the other tuple with respect to view `good_dept`. Thus, covered tuples can be deleted from the replica of relation `emp` that is stored on site 2 resulting in a smaller replica. In addition, the replica can also be maintained more efficiently. Thus, if a covered tuple is inserted into relation `emp` on site 1, the replica on site 2 need not be updated.

That subset of relation `emp` that contains only those tuples that have no cover in `emp` constitute the "representative relation" of `emp` with respect to the view `good_dept`. □

The intuition for building representative relations is that not all the tuples in a base relation provide new information with regard to a view, query, or constraint. Some tuples provide no additional information in the presence of other, covering, tuples. Therefore, the information contained in a base relation often can be summarized into smaller representative relations. All those tuples that are not covered by any other tuple constitute the representative relation. That is, covering with respect to a view definition defines a partial order on the tuples in any relation. The representative relation consists of the largest elements of this partial order. The smaller elements can be pruned.

The idea of pruning the redundant portions of a relation with respect to a particular query is the basis of most conventional query optimization techniques. The trick is in correctly and efficiently identifying the redundant portions of relations. Thus, we propose using the idea of pruning a part of a relation based on the remaining contents of the relation. We use this idea to develop the notion of "Generalized Projections" that are useful for query optimization as discussed in [HG94].

Constraint Management Systems

Many interesting directions have emerged from the design and implementation of DCMS. One issue is to keep track of design conflicts (constraint violations) that are detected and further track those that have been resolved or are in the process of being resolved. Thus conflicts that do not receive any attention for a given time period can be detected and reminders issued to participants. An alternative approach to violations is to try to fix them automatically (constraint enforcement). We believe that constraint enforcement is not a realistic approach to constraint management in engineering design databases because not all constraints can be enforced automatically. However, we need to be able to enforce *some* constraints and therefore we need to build constraint enforcing machinery in the system even though automatic fixing may not be done always. In addition, we need to better determine how best to use local checking and other partial-information-based methods in the presence of violations.

There is also a need for allowing *what-if* changes to be made – a sort of pseudo commit process – which will be very useful for iterative design. Thus, a designer can make changes that the designer would like to either commit to the database or reconsider, depending on the actual effect of the changes on the constraints in the system. The ability to make what-if changes provides designers with the ability to view the effect of potential changes without actually making them. Another area that needs work is the development of graphical constraint specification languages. Most constraints are specified by design engineers who are not well versed in programming languages and thus may prefer to use other more intuitive tools for constraint specification.

Appendix A

Extending Conjunctive Query Containment

In this appendix we consider the containment problem for conjunctive queries extended with interpreted predicates. The problem was first considered by Klug in [Klu88] using arithmetic inequalities as the interpreted predicates. We present a more general approach for solving the problem for arbitrary interpreted predicates using the idea of "containment mappings" [CM77]. The same approach has been used independently by [ZO93] to derive the results described in this appendix.

The results of this appendix are useful for building complete methods for locally checking conjunctive query constraints that use arithmetic inequalities. We also discuss a restricted class of conjunctive queries with arithmetic inequalities for which a stronger containment result can be proved than for the general class.

A.1 Preliminary Definitions and Examples

We consider conjunctive queries of the form:

$$C:\quad s(\bar{X}) \coloneq r_1(\bar{Y}_1)\ \&\ \ldots\ \&\ r_n(\bar{Y}_n)\ \&\ c_1(\bar{Z}_1)\ \&\ \ldots\ \&\ c_k(\bar{Z}_k)$$

where $r_1, \ldots, r_n$ are *ordinary* (uninterpreted) subgoals and $c_1, \ldots, c_k$ are interpreted subgoals. The ordinary subgoals represent relations that participate in query C and the interpreted subgoals could be function computations. The conjunction of the interpreted subgoals of C is referred to as $I(C)$ and the conjunction of the ordinary subgoals is referred to as $O(C)$. The following restrictions are imposed on C:

1. Every variable that appears in an interpreted predicate must also appear in some ordinary predicate, *i.e.* $\cup_{i=1}^{k}(\bar{Z}_i) \subseteq \cup_{j=1}^{n}(\bar{Y}_j)$.

2. $\bar{X} \subseteq \cup_{j=1}^{n}(\bar{Y}_j)$, *i.e.*, variables of the head also appear in an ordinary subgoal.

3. Constant arguments of ordinary subgoals and joins between ordinary subgoals are represented explicitly by using equality in the c_i's. Therefore, no two ordinary subgoals share a variable or have a constant argument.

The interpreted predicates could be predicates like arithmetic comparison operators. Arithmetic comparison operators are the predicates of most interest to us in the context of the discussion in Chapter 2. However, we develop containment results for arbitrary interpreted predicates by not considering the specific interpretation of the subgoals $c_1, \ldots, c_k$.

First we extend the definition of symbol mappings for conjunctive query constraints – as stated by Definition 2.7.2 on Page 29 – to conjunctive queries with interpreted predicates.

Definition A.1.1 (**Symbol Mapping**) A symbol mapping h is a function from a set of symbols S to another set of symbols T; *i.e.*, for each symbol $a \in S$, $h(a)$ is a symbol in T. Consider two conjunctive queries Q_1 and Q_2 of the form C defined above.

Q_1: P :- J_1 & ... & J_{n_1} & K_1 & ... & K_{k_1}
Q_2: R :- G_1 & ... & G_{n_2} & F_1 & ... & F_{k_2}

where the J's and G's are ordinary subgoals and the K's and F's are interpreted subgoals. Let $\bar{V}$ be the set of variables in Q_2 and let h be a symbol mapping on $\bar{V}$ where $h(v), v \in \bar{V}$ can be an arbitrary term and h is the identity mapping on predicate names and function symbols. h is a symbol mapping from Q_2 to Q_1 if h turns R into P and every ordinary subgoal G_i of Q_2 to some ordinary subgoal J_j of Q_1. □

We illustrate the use of symbol mappings to determine containment for conjunctive queries with no interpreted predicates.

EXAMPLE A.1.1 Consider the conjunctive queries Q_1 and Q_2:

Q_1: $p(X,Y)$:- $q(X,Y)$ & $r(U,V)$ & $r(V,U)$.
Q_2: $p(A,B)$:- $q(A,B)$ & $r(W,Z)$.

It is straightforward to observe that $Q_1 \subseteq Q_2$. The inference can be made using symbol mappings. Consider the following symbol mappings defined on the variables that occur in query Q_2:

1. $h(A) = X;\ h(B) = Y;\ h(W) = U;\ h(Z) = V.$
2. $g(A) = X;\ g(B) = Y;\ g(W) = V;\ g(Z) = U.$

Each symbol mapping converts the head of Q_2 to the head of Q_1 and converts every subgoal in the body of Q_2 to some subgoal in the body of Q_1. Therefore, every variable assignment ρ for the query Q_1 in database D can be converted into a variable assignment for query Q_2 in D by considering either $\rho \circ h$ or $\rho \circ g$. Therefore, every fact derived by Q_1 will also be derived by Q_2.

The symbol mappings h and g are called containment mappings [CM77] because the existence of a symbol mapping from Q_2 to Q_1 that satisfies the requirements of Definition A.1.1 is necessary and sufficient to infer that $Q_1 \subseteq Q_2$ if both Q_1 and Q_2 are conjunctive queries. □

However, the existence of a symbol mapping is *not* necessary and sufficient for containment of conjunctive queries with interpreted predicates. An extension of the above example illustrates this point when the interpreted predicate is $\leq$.

EXAMPLE A.1.2 This is Example 14.7 from [Ull89].

Q_1: $p(X,Y)$:- $q(X,Y)$ & $r(U,V)$ & $r(V,U)$.
Q_2: $p(X,Y)$:- $q(X,Y)$ & $r(U,V)$ & $U \leq V$.

The last predicate of Q_2 is interpreted in the obvious way. If "$\leq$" is treated as an ordinary (uninterpreted) predicate, there is no symbol mapping from Q_2 to Q_1 and therefore $Q_1 \subseteq Q_2$ cannot be inferred.

However, if the data domain of r is totally ordered then it is indeed the case that $Q_1 \subseteq Q_2$. Intuitively, the containment can be observed as follows. Consider a database D in which Q_1 derives the fact $p(x,y)$ using facts $q(x,y)$, $r(u,v)$, and $r(v,u)$. In addition, one of u and v has to be as large as the other. Without loss of generality, let $u \leq v$. Therefore, $r(u,v)$ & $u \leq v$ would also be true and thus query Q_2 will also derive fact $p(x,y)$ in D. Thus, we infer $Q_1 \subseteq Q_2$. □

Intuition for Our Approach

Symbol Mappings encode the relationship between ordinary predicates of the two queries in question. The algorithm described in this appendix breaks the question of containment into two steps. The first step uses the mappings from the ordinary predicates of Q_2 to the ordinary predicates of

Q_1 to produce a condition on the interpreted predicates in the two queries. This step does not depend on the interpretation of the subgoals. The condition derived by the first step is evaluated using the appropriate theory for the intended interpretation. For instance, if the interpreted predicates were arithmetic comparison operators with a dense domain, then the theory of ordered real numbers would be used to evaluate the condition produced by the algorithm. The following example illustrates this intuition.

EXAMPLE A.1.3 Consider queries Q_1 and Q_2 from Example A.1.2 assuming that the domain of the arguments of relation R is ordered. The mappings from the ordinary subgoals of Q_2 to the ordinary subgoals of Q_1 are as illustrated in Example A.1.1.

1. $h(A) = X;\ h(B) = Y;\ h(W) = U;\ h(Z) = V.$
2. $g(A) = X;\ g(B) = Y;\ g(W) = V;\ g(Z) = U.$

The mappings h and g give the two ways of transforming the ordinary subgoals of Q_2 to the ordinary subgoals of Q_1. Mappings h and g transform the interpreted predicates of Q_2 to $U \leq V$ and $V \leq U$ respectively. In order to determine if $Q_1 \subseteq Q_2$, we need to ensure that whenever the interpreted subgoals of Q_1 are true, the transformed interpreted subgoals of Q_2 are also true. Given that Q_1 has no interpreted subgoals, *i.e.* the interpreted subgoals of Q_1 are always *true*, we need to check that

$$true \ \Rightarrow\ U \leq V \texttt{ or } V \leq U.$$

The above statement is true given that U and V come from an ordered domain. Therefore, we can infer that $Q_1 \subseteq Q_2$. □

The algorithm is therefore applicable for all theories in which the resulting implication condition can be evaluated. For many theories, specialized algorithms and logical simplifications techniques evaluate the condition efficiently. For other theories, the condition may be undecidable but sufficient conditions could exist for these cases. For such theories, we can often infer containment of one query in another even though the necessary and sufficient condition is not evaluable.

A.2 Algorithm for Conjunctive Query Containment

Recall, ordinary subgoals do not share any variables in their arguments. We now state the results for containment of a conjunctive query Q_1 in a conjunctive query Q_2. We use the queries Q_1 and Q_2 from Definition A.1.1.

Theorem A.2.1 *Suppose Q_1 and Q_2 are two conjunctive queries with interpreted predicates (as in Definition A.1.1). Let $\mathcal{M}$ be the set of symbol mappings from conjunctive query Q_2 to query Q_1. $Q_1 \subseteq Q_2$ if and only if $\forall \bar{X}\ \exists h_{\ in\ \mathcal{M}} : [I(Q_1) \Rightarrow h(I(Q_2))]$ ($\bar{X}$ is the set of variables in Q_1).* □

Proof: $\boxed{\text{If}}$ Consider a variable assignment ρ that derives the fact f from query Q_1 in database D. Assuming that the conditions of the theorem hold, we construct a variable assignment for Q_2 that derives fact f from query Q_2.

If the conditions of the theorem hold then $\rho(I(Q_1)) \Rightarrow \rho(h(I(Q_2)))$ for some mapping $h \in \mathcal{M}$. Given that $\rho(I(Q_1))$ is true, we can infer that $\rho(h(I(Q_2)))$ is also true. That is, $I(Q_2)$ is satisfied by the variable assignment $\rho \circ h$. In addition, h maps every subgoal $G_i \in O(Q_2)$ to some subgoal $J_j \in O(Q_1)$, *i.e.*, $h(G_i) = J_j$. Given that $\rho(O(Q_1))$ is true in D, we can infer that $\rho(O(Q_2))$ is also true in D. Finally, $h(R) = P$ and therefore, $\rho(h(R)) = \rho(P) = f$.

$\boxed{\text{Only If}}$ Now suppose that $I(Q_1)$ does not imply $\vee_{h \ in \ \mathcal{M}} I(Q_2)$. Then there must be an instantiation ρ for the variables of Q_1 such that $\rho(I(Q_1))$ is true, yet for no h in $\mathcal{M}$ is $\rho(h(I(Q_2)))$ true.

Let D be the database consisting of exactly those tuples that are obtained by applying ρ to the ordinary subgoals of Q_1. Let f be the fact obtained by instantiating the head of Q_1 by ρ. Suppose Q_2 also produces f on D. Then there has to be some instantiation μ of the variables of Q_2 that makes the ordinary subgoals of Q_2 become tuples of D and such that $\mu(I(Q_2))$ is true. Because variables of Q_1 and Q_2 appear only once in ordinary subgoals, it is possible to write $\mu = \rho \circ h$, where h is a containment mapping from $O(Q_2)$ to $O(Q_1)$.

We know that $\rho(h(I(Q_2)))$ is false. That is, $\mu(I(Q_2))$ is false, contradicting the assumption that μ makes $I(Q_2)$ true. We conclude that Q_2 does not derive f on database D. Since Q_1 does, we have shown that when the condition of Theorem A.2.1 does not hold, Q_1 is not contained in Q_2. ∎

Example A.1.3 illustrates an application of Theorem A.2.1. Note that the implication condition involving the interpreted subgoals has to be checked separately and depends on the theory that deals with the particular interpretation of the subgoals. The above theorem can be extended in the obvious way to obtain a condition for the containment of a conjunctive query in a set of conjunctive queries:

Theorem A.2.2 *Suppose $\{Q_1, Q_2, \ldots, Q_n\}$ are conjunctive queries that use interpreted predicates. Let Q be another conjunctive query of the same form. Let $\mathcal{M}_i$ be the set of symbol mappings from conjunctive query Q_i to query Q. $Q \subseteq \{Q_1 \cup Q_2 \cup \ldots \cup Q_n\}$ if and only if $\forall \bar{X} \ \exists Q_i \ \exists h \ _{in \ \mathcal{M}_i} :$ $[I(Q) \Rightarrow h(I(Q_i))]$.* □

Proof: The proof is the similar to the proof of Theorem A.2.1. ∎

A.3 Special Classes of Conjunctive Queries

This section considers conjunctive queries where the interpreted predicates are the arithmetic comparison operators $<, >, \leq, \geq, =$ over a totally ordered dense domain. In particular, we describe two subclasses for which the containment condition derived by Theorem A.2.1 can be simplified. The results of this section are used to identify subclasses of conjunctive query constraints for which complete local checking can be done using first-order test conditions as discussed in Section 3.2. The two subclasses we consider are defined below:

Definition A.3.1 (left-semiinterval conjunctive query (LSCQ)) A conjunctive query Q is left-semiinterval if the arithmetic comparison subgoals in the query are either of the form $X < c$ or $X \leq c$ or $X = Y$ where X, Y are variables and c is a constant. A right-semiinterval query (RSCQ) is defined using $(>, \geq)$ in place of $(<, \leq)$. □

Recall from Theorem A.2.1 that the condition for query Q_1 to be contained in query Q_2 required computing all possible mappings from Q_2 to Q_1 and then checking an implication condition involving the interpreted subgoals as modified by *each* of these mappings. For LSCQs and RSCQs the necessary and sufficient condition involves finding just *one* mapping from Q_2 to Q_1 that satisfies an implication similar to that used before.

EXAMPLE A.3.1 Consider the two left-semiinterval conjunctive queries Q_1 and Q_2 where we want to check if $Q_1 \subseteq Q_2$:

Q_1: $p(X)$:- $q(X, Y)$ & $e(Y_1, A)$ & $e(Y_2, B)$ & $A \leq 5$ & $B \leq 15$ & $Y = Y_1$ & $Y = Y_2$.
Q_2: $p(W)$:- $q(W, Z)$ & $e(Z_1, C)$ & $e(Z_2, D)$ & $C \leq 10$ & $D \leq 20$ & $Z = Z_1$ & $Z = Z_2$.

The set of mappings from Q_2 to Q_1 are:

1. $h_1(W) = X$; $h_1(Z) = Y$; $h_1(Z_1) = Y_1$; $h_1(C) = A$; $h_1(Z_2) = Y_2$; $h_1(D) = B$.
2. $h_2(W) = X$; $h_2(Z) = Y$; $h_2(Z_1) = Y_1$; $h_2(C) = A$; $h_2(Z_2) = Y_1$; $h_2(D) = A$.
3. $h_3(W) = X$; $h_3(Z) = Y$; $h_3(Z_1) = Y_2$; $h_3(C) = B$; $h_3(Z_2) = Y_1$; $h_3(D) = A$.
4. $h_4(W) = X$; $h_4(Z) = Y$; $h_4(Z_1) = Y_2$; $h_4(C) = B$; $h_4(Z_2) = Y_2$; $h_4(D) = B$.

Consider mapping h_1 and the implication $I(Q_1) \Rightarrow h_1(I(Q_2))$.

$(A \leq 5$ & $B \leq 15$ & $Y = Y_1$ & $Y = Y_2) \Rightarrow h_1(C \leq 10$ & $D \leq 20$ & $Z = Z_1$ & $Z = Z_2)$.
$(A \leq 5$ & $B \leq 15$ & $Y = Y_1$ & $Y = Y_2) \Rightarrow (A \leq 10$ & $B \leq 20$ & $Y = Y_1$ & $Y = Y_2)$.

The above statement is true and therefore the containment holds. The same conclusion can be reached by using only mapping h_2. □

In general, for LSCQs and RSCQs one containment mapping is necessary and sufficient to infer containment. The following theorems formalize this result.

Lemma A.3.1 *Let each of $D_0, D_1, \ldots, D_n$ be a conjunction of arithmetic constraints of the form $X < a$ or $X \leq a$. $D_0 \Rightarrow (D_1 \vee D_2 \vee \ldots \vee D_n)$ if and only if $D_0 \Rightarrow D_i$ for some $i, 1 \leq i \leq n$.* □

Proof: We give the intuition for the proof by considering the case when each of $D_0, D_1, \ldots, D_n$ involves a single variable X and only $\leq$ is used. Let D_i be $X \leq a_i$. The values of X that are a solution to D_i lie in the interval $(-\infty, a_i]$. Similarly for each of $D_0, D_1, \ldots, D_n$ the solution space for X is an interval from $-\infty$ to some constant as the right limit. If $D_0 \Rightarrow (D_1 \vee D_2 \vee \ldots \vee D_n)$ then the interval $(-\infty, a_0]$ is contained in the union of the solution intervals $(-\infty, a_1], \ldots, (-\infty, a_n]$. Let $a_k = \mathbf{max}(a_1, \ldots, a_n)$. Therefore, $(-\infty, a_k]$ properly contains all the other solution intervals. Thus, $D_0 \Rightarrow D_k$.

In general, let each of the conjunctions $D_0, D_1, \ldots, D_n$ use m variables $X_1, \ldots, X_m$.

$\boxed{\text{If}}$ Follows from: $[A \Rightarrow B] \Rightarrow [A \Rightarrow (B \vee C \vee \ldots \vee K)]$.

$\boxed{\text{Only If}}$ (By contradiction) Assume, $D_0 \Rightarrow (D_1 \vee D_2 \vee \ldots \vee D_n)$ holds but $D_0 \Rightarrow D_j$ does not hold for any conjunction $D_j, 1 \leq j \leq n$. Therefore, for each $D_j, 1 \leq j \leq n$ there is a vector v_j of constants $a_1, \ldots, a_m$ that when assigned to variables $X_1, \ldots, X_m$ satisfy D_0 but do not satisfy D_j; we say that v_j *differentiates* the pair (D_0, D_j). Given that D_0 does not imply any D_j, there is at least one differentiating vector for each of the pairs $(D_0, D_j), 1 \leq j \leq n$.

Let $\mathcal{V}$ be such a set of differentiating vectors. Note, every vector in $\mathcal{V}$ satisfies constraint D_0. From the set $\mathcal{V}$ we then construct a new vector v' such that $v'(i) = \mathtt{max}\{v(i) | v \in \mathcal{V}\}$ *i.e.* $v'(i)$ is the maximum value assigned to variable X_i by any vector in $\mathcal{V}$.

Because variables $X_1, \ldots, X_m$ are independent of each other in each of the conjunctions, v' satisfies the constraint D_0. However, v' differentiates every constraint from D_0. This contradicts the assumption that $D_0 \Rightarrow (D_1 \vee D_2 \vee \ldots \vee D_n)$. ∎

Lemma A.3.1 assumes that comparisons of the form $X{=}Y$ did not appear in the conjunctions. The lemma also holds when such equalities are used in the conjunctions. If $X{=}Y$ occurs in the conjunction D_0, all occurrences of Y can be replaced by X in $D_0, D_1, \ldots, D_n$ without affecting the truth of the implication. All equalities in D_0 can be eliminated in this manner. In case some equality remains in one of the D_i's, say $A{=}B$ in D_k, D_k can be replaced by $D_{k_a} \vee D_{k_b}$ where D_{k_a} is D_k with B replaced by A. It is easy to observe that the truth of the implication is not affected in this case either. Note, comparisons of the form $X{=}c$ are not allowed. Example A.3.2 illustrates why.

Theorem A.3.1 *LSCQ Q_1 is contained in LSCQ Q_2 if and only if $I(Q_1) \Rightarrow h(I(Q_2))$ for some mapping h from Q_2 to Q_1.* □

Proof: By Theorem A.2.1, $Q_1 \subseteq Q_2$ if and only if $\exists h$ $_{in\ \mathcal{M}}$: $[I(Q_1) \Rightarrow h(I(Q_2))]$, where $\mathcal{M}$ is the set of mappings from Q_2 to Q_1. The condition can be rewritten as $I(Q_1)$ implies $\bigvee_{h \in \mathcal{M}} h(I(Q_2))$. If Q_1 and Q_2 are LSCQs, then $I(Q_1)$ and $I(Q_2)$ are conjunctions of left-semiinterval constraints. Lemma A.3.1 can be used to infer that $I(Q_1)$ implies $\bigvee_{h\ in\ \mathcal{M}} h(I(Q_2))$ if and only if $I(Q_1) \Rightarrow h(I(Q_2))$ for some $h \in \mathcal{M}$. ∎

Theorem A.3.2 *, LSCQ query q is contained in a union of LSCQs $\{Q_1 \cup \ldots \cup Q_m\}$ if and only if q is contained in some $Q_i, 1 \leq i \leq m$.* □

Proof: Similar to the proof of Theorem A.2.1. ∎

Theorems A.3.1 and A.3.2 apply to RSCQs also. The following example illustrates that if the arithmetic comparison operators include inequalities of the form $X = c$ in addition to $X < c$ and $X \leq c$, then the above theorems do not hold.

EXAMPLE A.3.2 (This example was originally suggested by Surajit Chaudhuri.) Consider the queries Q, Q_1, and Q_2; we want to check if $Q \subseteq (Q_1 \cup Q_2)$.

Q: $s(X)$:- $t(X)$ & $X \leq 7$.

Q_1: $s(X)$:- $t(X)$ & $X < 7$.

Q_2: $s(X)$:- $t(X)$ & $X = 7$.

It is simple to observe that $Q \subseteq (Q_1 \cup Q_2)$ but $Q \not\subseteq Q_1$ and $Q \not\subseteq Q_2$. The example uses the fact that the interval $(-\infty, 7]$ can be broken up into the intervals $(-\infty, 7)$ and $[7, 7]$. $\square$

Appendix B

Algorithm to Eliminate Remote Variables from Local Tests

This appendix considers the problem of eliminating the universally quantified variables from $\mathsf{TC}(C, l, \mu)$ which is of the form:

$$\mathsf{TC}(C,l,\mu):\quad \exists\bar{X}\,\forall\bar{Y}\,\forall\bar{Z} : [L(\bar{X}) \wedge (g(\mu,\bar{Y},\bar{Z},\bar{c}) \quad\Rightarrow\quad g(\bar{X},\bar{Y},\bar{Z},\bar{c}))]$$

The intent is to eliminate the universally quantified variables from the above test condition to obtain a condition $\mathcal{I}$ involving only variables in $\bar{X}$, the parameter μ, and the constants in $\bar{c}$, such that:

$$\exists\bar{X} : [\mathcal{I} \quad\Rightarrow\quad \forall\bar{Y}\forall\bar{Z} : (g(\mu,\bar{Y},\bar{Z},\bar{c}) \;\Rightarrow\; g(\bar{X},\bar{Y},\bar{Z},\bar{c}))]$$

The condition $\mathcal{I}$ on $\bar{X}$ can be posed as a query on the accessible relation and if the query has a nonempty answer then we conclude that test $\mathsf{TC}(C, l, \mu)$ is true. We would like to derive $\mathcal{I}$ using only the parameter μ and not the actual inserted tuple. Thus, $\mathcal{I}$ can be derived at compile-time and then evaluated when the inserted tuple is available. Also, we use $\bar{Y}$ to represent both the sets $\bar{Y}$ and $\bar{Z}$ because both sets are treated similarly for elimination purposes.

EXAMPLE B.0.3 Consider the following arithmetic implication where the letters a, b represent the parameter μ, the variable A corresponds to the universally quantified variables in $\bar{Y}$, and V, W represent the existentially quantified variables in $\bar{X}$.

$$\exists V, W\,\forall A : [A \geq a \wedge A \geq b \quad\Rightarrow\quad A \geq V \wedge A \geq W].$$

We want to eliminate variable A in order to get a condition $\mathcal{I}$ on variables V and W such if $\mathcal{I}$ holds, then so does the above implication. We can replace the above implication by:

$$\exists V, W\,\forall A : [A \geq \mathbf{max}(a,b) \quad\Rightarrow\quad A \geq V \wedge A \geq W].$$

If we can find V and W that are $\leq \mathbf{max}(a, b)$ then the above implication is guaranteed to hold. This intuition is used to develop the algorithm for eliminating universally quantified variables without using the inserted tuple.

Now consider a more complicated implication:

$$\forall A, B : [A \geq a \wedge B \geq b \wedge A \geq B \quad \Rightarrow \quad A \geq V \wedge B \geq W \wedge A \geq B].$$

The LHS of the above sentence implies that $A \geq \mathbf{max}(a, b)$. This inference can be made using one transitive step. Thus, we can infer that if $V \leq \mathbf{max}(a, b)$ and $W \leq b$, then the above implication holds. □

$\mathcal{I}$ is computed from $\mathsf{TC}(C, l, \mu)$ in two phases.

1. From the LHS of the implication $g(\mu, \bar{Y}, \bar{c}) \Rightarrow g(\bar{X}, \bar{Y}, \bar{c})$, *i.e.*, from $g(\mu, \bar{Y}, \bar{c})$, compute the tightest parametric bounds on the variables $\bar{Y}$.
2. Propagate the bounds for $\bar{Y}$ computed in Phase 1 to the variables $\bar{X}$ thereby obtaining a query on relation L.

Phase 1: Generating Parametric Bounds on Variables in $\bar{Y}$

We describe how to infer the tightest parametric bounds on the remote variables using $g(\mu, \bar{Y}, \bar{c})$. For the sake of simplicity, we assume that comparisons $<$ and $>$ are not permitted. Shortly, we relax this restriction.

- Build a graph that has one node for each remote variable.
- Draw a directed edge from node Y to node Z if $Y \leq Z$ is in $g(\mu, \bar{Y}, \bar{c})$ (or if $Y < Z$ is in $g(\mu, \bar{Y}, \bar{c})$).
- Each node Y is labelled with its lower and upper bounds y_l and y_h. Initially all nodes are labelled $(-\infty, +\infty)$.
- Given the inequalities $Y \geq a_1, Y \geq a_2, \ldots, Y \geq a_k$ in $g(\mu, \bar{Y}, \bar{c})$, node Y is labelled $(\mathbf{max}(-\infty, a_1, a_2, \ldots, a_k), \infty$
- Given the inequalities $Y \leq b_1, Y \leq b_2, \ldots, Y \leq b_k$ in $g(\mu, \bar{Y}, \bar{c})$, node Y is labelled $(y_l, \mathbf{min}(\infty, b_1, b_2, \ldots, b_k))$, where y_l is as computed by the previous item.
- If the equality $Y = a$ occurs in $g(\mu, \bar{Y}, \bar{c})$ then y_l is changed to $\mathbf{max}(a, y_l)$ and y_h is changed to $\mathbf{min}(a, y_h)$.
- If there is an edge from node Y to node Z, and assuming that the label on node Y is $[y_l, y_h]$, and the label on node Z is $[z_l, z_h]$, then:
  - Change y_h to $\mathbf{min}(y_h, z_h)$.
  - Change z_l to $\mathbf{max}(y_l, z_l)$.

The above operation eventually reaches fixed point. That is, all labels in the graph remain unchanged on subsequent applications of the above operation.

Note, y_h and z_h are defined using, possibly many occurrences of, **min**. If y_h and z_h do use **min** and further we need to compute $\mathbf{min}(y_h, z_h)$, then the inner application of **min** can be folded into the outer application. That is, say y_h was defined as $\mathbf{min}(a, b, c)$. Then $\mathbf{min}(y_h, z_h)$ is the same as $\mathbf{min}(a, b, c, z_h)$.

We can accomodate inequalities of the form $Y > a$, and $Y > Z$ by storing an extra bit with each limit. Thus, each of the limits y_l and y_h for Y are ordered pairs of the kind (a, b) where the first element indicates the limit and the second element indicates if the comparison is strict or not. Thus, if y_l is $(a, 1)$ then the comparison is of the form $Y > a$, and if y_l is $(a, 0)$ then the comparison is of the form $Y \geq a$. **min** and **max** are defined over ordered pairs. That is, $(a, 1) > (a, 0)$. Thus, combining inequalities $Y > 2 \wedge Y \geq 2$ yield the lower limit on Y as $\mathbf{max}((2, 1), (2, 0))$ which is $(2, 1)$ and corresponds to $Y > 2$ which is the more strict condition as we desired.

Phase 2: Propagating Bounds on $\bar{Y}$ to $\bar{X}$

Phase 2 uses the bounds derived in the Phase 1 and $g(\bar{X}, \bar{Y}, \bar{c})$, the RHS of the arithmetic implication in $\mathsf{TC}(C, l, \mu)$, to derive condition $\mathcal{I}$ on the variables in $\bar{X}$. The strategy is as follows:

- For each remote variable Y in $\bar{Y}$ create the condition $y_l \leq y_h$. When the value of the parameter μ is available, then these conditions can be evaluated. In case $y_l \leq y_h$ evaluates to false for any variable Y, then we can infer that the inserted tuple is such that it contradicts the arithmetic comparisons of g. Thus, the integrity constraint assertion cannot be falsified because g appears as the antecedent of the assertion. In other words, the antecedent of the implication in $\mathsf{TC}(C, l, \mu)$ becomes false and thus the implication evaluates to true. To capture the requirement that the lower limit of each variable should be at most as big as the upper limit of the variable, we add the condition *not* $(y_l \leq y_h)$ as a *disjunct* to $\mathcal{I}$.

- If $g(\bar{X}, \bar{Y}, \bar{c})$ has an inequality of the form $X \leq Y$ or $X < Y$, then add the condition $X \leq y_l$ to $\mathcal{I}$ as a conjunct.

- If $g(\bar{X}, \bar{Y}, \bar{c})$ has an inequality of the form $X \geq Y$ or $X > Y$, then add the condition $X \geq y_h$ to $\mathcal{I}$ as a conjunct.

- If $g(\bar{X}, \bar{Y}, \bar{c})$ has an inequality of the form $X = Y$ then add the condition $X = y_h$ to $\mathcal{I}$ as a conjunct. Both y_h and y_l are forced to be equal because if $X = Y$ occurs in $g(\bar{X}, \bar{Y}, \bar{c})$ then there has to be an equality of the form $c = Y$ in $g(\mu, \bar{Y}, \bar{c})$ which will force the upper and lower limits of Y to become the same.

B.1 Eliminating Remote Variables from $\mathsf{TC}_D(C, l, \mu)$

This section outlines the changes needed to change the strategy for determining $\mathcal{I}$ for $\mathsf{TC}(C, l, \mu)$ such that $\mathcal{I}$ can be computed for $\mathsf{TC}_D(C, l, \mu)$. The difference arises because in $\mathsf{TC}(C, l, \mu)$ the parameterized component of the implication appears on the LHS of the implication whereas in $\mathsf{TC}_D(C, l, \mu)$ the parameterized component is on the RHS. The following example highlights the difference.

EXAMPLE B.1.1 Consider the implication from Example B.0.3 with the LHS and RHS reversed.

$$\forall A : [A \geq X \wedge A \geq Y \quad \Rightarrow \quad A \geq a \wedge A \geq b].$$

The above implication holds for a particular X and Y if the variables are $\geq \mathbf{max}(a, b)$. □

Therefore, the bounds computed on remote variables need to be propagated to the local variables differently. Thus, Phase 1 remains the same as before. Phase 2 changes as follows.

- If $g(\bar{X}, \bar{Y}, \bar{c})$ has an inequality of the form $X \leq Y$ or $X < Y$, then add the condition $X \geq y_h$ to $\mathcal{I}$ as a conjunct. Note, in the case of $\mathsf{TC}(C, l, \mu)$ we added $X \leq y_l$.
- If $g(\bar{X}, \bar{Y}, \bar{c})$ has an inequality of the form $X \geq Y$ or $X > Y$, then add the condition $X \leq y_l$ to $\mathcal{I}$ as a conjunct. Note, in the case of $\mathsf{TC}(C, l, \mu)$ we added $X \geq y_h$.

Appendix C

Formalizing Containment of Rectangles

In Section 3.1.2 of Chapter 3 on Page 39 we discuss how to use Datalog rules to instantiate and solve the parameterized test condition (T) using relation L and the inserted tuple t when the constraints of interest are independently constrained queries (IQCs) (Definition 3.1.3). On Page 40 we give a five step process for generating a Datalog program that is sufficient to express (T) for IQCs.

The Datalog program was derived using the intuition that inferring the truth of (T) for a given L and t was the same as determining if a k-dimensional parallelepiped is contained in a union of other k-dimensional parallelepipeds. We derived the Datalog rules based on a fragment-combine algorithm for rectangles described pictorially in Figure 3.2.

In this appendix we prove the correctness of the algorithm for checking if a rectangle is contained in a union of other rectangles. We also prove that the Datalog rules that we generated in Section 3.1.2 do indeed check the containment property that we are interested in. Finally, we also prove the correctness of the Datalog rules generalized to k-dimensional parallelepipeds.

We start with proving some properties of arithmetic inequalities. Then we discuss the containment problem for 1-dimensional spaces, *i.e.*, intervals on the number line. Then we generalize the solutions to k-dimensions.

C.1 Characterizing Conjunctions of Arithmetic Inequalities

In this section we discuss some properties of conjunctions of arithmetic inequalities and show how Datalog predicates relate to the conjunctions. Note, the conjunctions discussed are the type that occur in IQCs.

Observation 1 *Consider a conjunction of arithmetic inequalities:*

$X \; \texttt{op} \; a_1, X \; \texttt{op} \; a_2, \ldots, X \; \texttt{op} \; a_n.$

where "," represents logical "and" and each occurrence of `op` *can be any of* $>, <, =, \neq, \geq, \leq$. *The above conjunction can be equivalently represented as a disjunction of conjunctions where each conjunction uses only the operators* $>, <, =$. *The transformation can be effected by the following substitutions:*

$\neq$ *by* $< \vee >$
$\geq$ *by* $= \vee >$
$\leq$ *by* $= \vee >$

and then representing the resulting expression in disjunctive normal form. □

Lemma C.1.1 *Consider the conjunction:*

$X > a_1, X > a_2, \ldots, X > a_n.$

where each a_i *is a constant or a parameter. The values of variable* X *for which the conjunction is true are the same as the values of* X *for which the following inequality is true.*

$X > \mathbf{max}(a_1, \ldots, a_n).$

The values can be represented by the open interval $(\mathbf{max}(a_1, \ldots, a_n), \infty)$. □

The proof of the above lemma is simple to observe. Henceforth, we also use $\mathbf{max}_{i=1}^{n}(a_n)$ in place of $\mathbf{max}(a_1, \ldots, a_n)$.

Observation 2 *If the possible values of a variable* X *lie in an open interval* (a, b), *a 2-ary predicate can be used to represent the solution space for* X. *Let one such predicate be* `soln`. *The first argument of* `soln` *represents the lower limit for* X *and the second argument represents the upper limit for* X. *By definition, we assume that if the only solution to variable* X *is equal to a constant* c, *the predicate* `soln` *has* c *as its first argument and* $NULL$ *as the second argument.* □

Lemma C.1.2 *The maximum element of a set* s *of* n *elements can be computed by a set of* n *Datalog rules that use the order predicate* $\geq$. □

Proof: The rules to compute the maximum element of the set s are:

$\texttt{max}(X_1, \ldots, X_n, X_1) :- s(X_1, \ldots, X_n) \;\&\; X_1 \geq X_2 \;\&\; X_1 \geq X_3 \;\&\; \ldots \;\&\; X_1 \geq X_n.$
$\texttt{max}(X_1, \ldots, X_n, X_2) :- s(X_1, \ldots, X_n) \;\&\; X_2 \geq X_1 \;\&\; X_2 \geq X_3 \;\&\; \ldots \;\&\; X_2 \geq X_n.$
....
$\texttt{max}(X_1, \ldots, X_n, X_n) :- s(X_1, \ldots, X_n) \;\&\; X_n \geq X_1 \;\&\; X_n \geq X_2 \;\&\; \ldots \;\&\; X_n \geq X_{n-1}.$

The first n arguments of predicate `max` are elements of the set, and the last argument is the maximum element. ∎

Note, only the size of the set needs to be known, not the actual elements themselves. A similar set of rules can be generated to compute the minimum element of a set of known size. Also, we use **max** to represent the built-in function that computes the maximum of a set of arguments and `max` to represent the predicate defined above that takes $n+1$ arguments and assigns the maximum of the first n arguments to the $n+1^{st}$ argument.

Lemma C.1.3 *Consider the conjunction:*

$X > a_1, X > a_2, \ldots, X > a_n.$

where each a_i is a constant or a parameter. The solutions for variable X can be computed by a set of Datalog rules. The solution is represented as a tuple in a 2-ary relation, say `soln`*, as specified in Observation 2. The constant ∞ is used in the rules given that $\infty > c$ is true for all constants $c \neq \infty$.* □

Proof: By construction. The maximum element of the set of constants $a_1, \ldots, a_n$ can be computed using Datalog as described in Lemma C.1.2. Let the maximum be defined by the predicate *max_of_set*. The solutions for X are defined by the rule:

$\texttt{soln}(X_l, +\infty) \iff max\_of\_set(X_l).$ ∎

Just as Lemmas C.1.1 and C.1.3 hold for conjunctions that use the arithmetic predicate $>$, similar lemmas hold for conjunctions that use the predicate $<$. The constant $-\infty$ is introduced for handling $<$. We now consider a sentence that uses all the three arithmetic predicates that we allow: $<, >, =$.

Lemma C.1.4 *Consider a conjunction of inequalities of the form:*

$\mathcal{V}$: $X > a_1, \ldots, X > a_n, X < b_1, \ldots, X < b_m, X = c_1, \ldots, X = c_k.$

where each a_i, b_j, c_l is a constant or a parameter. The values of X for which the conjunction $\mathcal{V}$ is true is represented by one of the following four cases.

1. $X = c_1$
   if $c_1 = c_2 = \ldots = c_n$ *and* $\mathbf{max}_{i=1}^n(a_i) < c_1 < \mathbf{min}_{i=1}^m(b_i)$.

2. $(\mathbf{max}_{i=1}^n(a_i), \mathbf{min}_{i=1}^m(b_i))$
   if $k=0$ *i.e., X is not equated to any constant, and* $\mathbf{max}_{i=1}^n(a_i) < \mathbf{min}_{i=1}^m(b_i)$.

3. $(-\infty, \mathbf{min}_{i=1}^m(b_i))$
   if $k=0$ *and* $n=0$ *i.e., X is not equated to nor greater-than any constant.*

4. $(\mathbf{max}_{i=1}^n(a_i), \infty)$
   if $k=0$ *and* $m=0$ *i.e., X is not equated to nor less-than any constant.*

□

Proof: The conjunction $X > a_1, \ldots, X > a_n$ can be replaced by $X > \mathbf{max}_{i=1}^{n}(a_i)$ (Lemma C.1.1). Similarly $X < b_1, \ldots, X < b_m$ can be replaced by $X < \mathbf{min}_{i=1}^{m}(b_i)$. Finally, $X = c_1, \ldots, X = c_k$ can be replaced by $X = c_1$ if all the C_is are equal (else the conjunction is inconsistent and has no solution). The possible solutions for the resulting conjunction:

$\mathcal{V}$: $X > \mathbf{max}(a_i), X = c_1, X < \mathbf{min}(b_i)$

are exactly as listed as the four cases. ∎

Lemma C.1.5 *Given a conjunction of the form $\mathcal{V}$, it is possible to write Datalog rules defining predicate* `soln` *such that if* $\texttt{soln}(a, b)$ *is derivable by the rules and $b \neq NULL$, then $a < X < b$ is the solution space for the variable X in $\mathcal{V}$. If $b = NULL$, then $X = a$ is the only solution for X in $\mathcal{V}$.* □

Proof: A different set of Datalog rules is defined for each case listed in Lemma C.1.4. We discuss only one case in detail.

$\boxed{\text{Case 1}}$ The rule defining `soln` is:

$$\begin{aligned}\texttt{soln}(c_1, NULL) \coloneq\ & c_1 = c_2 = \ldots = c_{k-1} = c_k \ \& \\ & \texttt{max}(a_1, \ldots, a_n, A) \ \& \ A < c_1 \ \& \\ & \texttt{min}(b_1, \ldots, b_m, B) \ \& \ B > c_1.\end{aligned}$$

Rules for `max` and `min` are as defined by Observation C.1.2. We argue that the above rule derives $\texttt{soln}(c_1, NULL)$ if and only if the solution of $\mathcal{V}$ is $X = c_1$.

If: Note, the above rule repeats the conditions of Case 1. Therefore, if the conditions of Case 1 are true, then this rule will also be true.

Only If: If the rule is not true then Case 1 cannot be true either.

The arguments for the other cases are similar and simple to see. ∎

C.1.1 Manipulating (T) into a Desired Form

Recall the form of the local test (T) generated by the steps described on Page 31.

$A' \Rightarrow B'_1 \vee \ldots \vee B'_n$

where A' is $I(\mathsf{Red}(\mu, l, C))$ and B'_i is $I(\mathsf{Red}(\alpha, l, C))$. Let's assume that the local test involves only one variable X. Thus, $A', B'_1, \ldots, B'_n$ are conjunctions of arithmetic order predicates involving $>, <, \leq, \geq, =$ where each arithmetic predicate uses either X and a (parameterized) constant or two (parameterized) constants. Using the transformation of Observation 1, the predicates $\leq, \geq$ can be eliminated such that (T) is transformed to:

$(A_1 \vee \ldots \vee A_m) \Rightarrow (B_1 \vee \ldots \vee B_k)$

Using the logical equivalence of $(a \vee b) \Rightarrow c$ and $(a \Rightarrow c) \wedge (b \Rightarrow c)$, the above local test can be rewritten as the following conjunction of m tests:

$$(T):\quad \wedge_{i=1}^{m}(A_i \Rightarrow B_1 \vee \ldots \vee B_k) \qquad = \qquad T_1 \wedge T_2 \wedge \ldots \wedge T_m$$

Each of $A_1, \ldots, A_m, B_1, \ldots, B_k$ is of the form $\mathcal{V}$ given in Lemma C.1.4. Using Observation 2 we observe that the solution to each A_i can be represented as a tuple of a 2-ary relation that is defined by a rule that uses the parameter μ and constants in A_i. Let the predicate `ins` represent the solutions to the A_is. Similarly, the solution to each B_j can be represented as a tuple of the predicate `soln` and can be computed by rules that use the parameter α and the constants in B_j.

Note, each conjunct T_i of test (T) can be instantiated to obtain an arithmetic instantiation I_i using the inserted tuple t and relation L just as (T) is instantiated to obtain I_t The LHS of I_i is obtained by instantiating A_i with tuple t, and the RHS is obtained by instantiating each B_j with a tuple in L and adding the resulting sentence as a disjunct in the RHS. The result of the instantiation is the conjunction $I_1 \wedge \ldots \wedge I_m$. Note, the RHS of all $I_1 \wedge \ldots \wedge I_m$ is the same.

Alternatively, each of $A_i, B_1, \ldots, B_k$ in T_i is of the form $\mathcal{V}$ and thus the solutions to the variables in each component can be represented by Datalog rules as described in Lemma C.1.5. The tuples of `ins`, corresponding to the solutions of the LHS of I_i, A_i, are computed using the tuple t to instantiate parameter μ. The rule that defines `ins`, and uses the parameter μ in its body, has the additional subgoal `ins`(μ) to instantiate μ with the inserted tuple t. Similarly, the `soln` tuple that represents the solutions to the variables that occur in the disjuncts in the RHS of I_i can be computed using the tuples of L to instantiate parameter α. Every rule that defines `soln`, and uses parameter α in its body, has the additional subgoal $l(\alpha)$ such that the tuples in L provide the alternative instantiations for α. The relations `soln` and `ins` can be defined by Datalog rules derived using Lemma C.1.5 for each $I_i, 1 \leq i \leq m$.

Thus, after having seen how to generate Datalog rules for the solutions for both the RHS and LHS of I_i as generated above, we now describe how to determine the truth of I_i.

C.1.2 Determining containment of 1-dimensional spaces

First we consider conjuncts that use only one variable. That is, we consider I_i that uses only one variable. Subsequently we extend the results to multivariable cases.

Lemma C.1.6 (splitting-lemma): *Consider the sentence*

$D:\quad a < X < c$

where $a < c$. Consider constant b such that $a < b < c$. X satisfies $a < X < c$ if and only if X satisfies:

$D':\quad a < X < b \quad \vee \quad X = b \quad \vee \quad b < X < c.$

□

Proof: Simple to observe. ∎

Lemma C.1.7 *Let* soln(a, c) *represent the solution to* $a < X < c$. *Let constant* b, $a < b < c$, *be used to partition* $a < X < c$ *as in Lemma C.1.6. The solution facts that represent the solutions to the resulting disjunction, are defined by the following rules:*

soln(X, Y) :– soln(X, Z) & constant(Y) & $X < Y < Z$.
soln$(Y, NULL)$:– soln(X, Z) & constant(Y) & $X < Y < Z$.
soln(Y, Z) :– soln(X, Z) & constant(Y) & $X < Y < Z$.

where constant *is some base relation that is used to instantiate* Y. □

Definition C.1.1 (cons) Consider a sentence S that uses the comparison operators $<, >, =$ and the logical connectives $\vee, \wedge$. cons(S) refers to the set of constants that appear in S. □

Observation 3 *Consider a disjunction* Θ *of arithmetic predicates where each disjunct is of the form* $a < X < b$ *or* $X = c$. *Consider the corresponding* soln *facts that represent the solutions to* Θ. *The set of constants* cons(Θ) *can be generated by the following rule:*

cons(X) :– $soln(X, Y)$.
cons(Y) :– $soln(X, Y)$.

□

Definition C.1.2 (**Minimizing a disjunction of arithmetic inequalities**) Consider a disjunction of sentences Θ where each disjunct is of the form $a < X < b$ or $X = c$. Also, consider a set of constants C and a particular constant $c \in C$ such that there is a disjunct $a < X < b$ in Θ and $a < c < b$. Using Lemma C.1.6, $a < X < b$ can be replaced in Θ by a disjunction of three sentences using c as the splitting point. Θ is said to be *minimized* with respect to C if no disjunct in Θ can be split any further with respect to any constant in C and all repeated disjuncts have been eliminated. □

Given the predicate soln that represents solutions for a sentence of the form $a < X < b$, and a predicate constant that represents a set of constants, the soln facts corresponding to the minimization of $a < X < b$ are generated by the recursive rules of Lemma C.1.7. However, minimization required *eliminating* a disjunct D after D had been split to avoid having non-minimal disjuncts in **min**(O). This elimination process is replicated in Datalog by first identifying all the soln tuples that are not minimal, and then eliminating these from the set of all soln tuples generated by the splitting rules.

not_min_soln(X, Y) :– soln(X, Y) & constant(Z) & $X < Z < Y$.
min_soln(X, Y) :– soln(X, Y) & $\neg$not_min_soln(X, Y).

Lemma C.1.8 *Consider a disjunction Θ of arithmetic inequalities each of the form $a < X < b$ or $X = c$. Let $\mathsf{min}(\Theta)$ be the minimization of Θ with respect to $\mathsf{cons}(\Theta)$. The solutions for every pair of disjuncts in $\mathsf{min}(\Theta)$ are disjoint.* □

Proof: Simple to observe. ∎

Definition C.1.3 (Index) The index of the arithmetic inequality $X = c$ is $(c, =)$ and of the inequality $a < X < b$ is $(a, <)$. The ordering relationship $<_x$ between indexes is defined as follows:

$$(i, op_i) <_x (j, op_j) \text{ if } \begin{cases} i < j \text{ or} \\ i = j \text{ and } op_i \text{ is } = \text{ and } op_j \text{ is } < \end{cases}$$

If op_i is $(c, <)$ and op_j is $(c, <)$, then the order predicate $<_x$ is not defined on the two indices. □

Lemma C.1.9 (Ordering-lemma): *Consider a set of inequalities D that is minimized with respect to the constants in $\mathsf{cons}(D)$. The disjuncts in D can be ordered totally using the $<_x$ relationship defined in Definition C.1.3.* □

Proof: We prove the above lemma by contradiction. Let there be two distinct disjuncts D_1 and D_2 that cannot be ordered with respect to each other. Therefore, the index of D_1 is $(c, <)$ and the index of D_2 is $(c, <)$. By Definition C.1.3, D_1 and D_2 are of the form $c < X < b_1$ and $c < X < b_2$ respectively. In addition, $b_1 \neq b_2$ because D_1 and D_2 are distinct. Without loss of generality assume that $b_1 < b_2$. Then $c < b_1 < b_2$ and $b_1 \in \mathsf{cons}(D)$ and therefore D_2 is not minimal. ∎

Consider $I_1 \wedge \ldots \wedge I_m$ obtained by instantiating the local test (T) from Section C.1.1 by t and L. Let the set of constants involved in the RHS of any of the I_is be $\mathcal{K}$. Consider a particular I_i and minimize both the RHS and the LHS of I_i with respect to $\mathcal{K}$. Minimization replaces each sentence by disjunctions of sentences. The disjunctions introduced in the LHS of I_i can be eliminated by replacing I_i by a conjunction $I_i^1 \wedge \ldots \wedge I_i^p$ where each I_i^j is of the same form as I_i before splitting (using a process similar to the one used in Section C.1.1 to eliminate $\leq, \geq$ from (T)). Therefore, minimization of $I_1 \wedge \ldots \wedge I_m$ with respect to $\mathcal{K}$ results in a sentence of the same form as $I_1 \wedge \ldots \wedge I_m$ except that the number of conjuncts is larger and every arithmetic inequality in the sentence has been minimized wrt $\mathcal{K}$.

Now we discuss how to solve one of the implications I_j.

Lemma C.1.10 *Consider the implication:*

I: $A \Rightarrow B_1 \vee \ldots \vee B_k$.

where A and each B_j is minimized with respect to the constants occuring in $B_1 \vee \ldots \vee B_k$. I is true if and only if there exists some i such that $A \Rightarrow B_i$. □

Proof: Simple to observe. Intuitively, if A is not contained in some B_i but is contained in a disjunction of two or more B_is, then A could be further minimized. ∎

We now discuss how to use Datalog rules to determine if a sentence A of the form $X = c$ or $a < X < b$ implies (is contained in) another inequality B of the same form. Recall that the solutions to inequalities of these two forms can be represented as facts of a 2-ary predicate.

Observation 4 *Let inequalities A and B be of the form $X = c$ or $a < X < b$. Let the corresponding solution facts be* `ins`(a_l, a_h) *and* `soln`(b_l, b_h)*. $A \Rightarrow B$ if and only if one of the following rules derives* `OK`*.*

`OK` :– `ins`(a_l, a_h) & `soln`(b_l, b_h) & $b_l \leq a_l$ & $b_h \geq a_h$.
`OK` :– `ins`$(a_l, NULL)$ & `soln`(b_l, b_h) & $b_l \leq a_l \leq b_h$.
`OK` :– `ins`$(a_l, NULL)$ & `soln`$(b_l, NULL)$ & $b_l = a_l$.

□

Recall that predicate `ins` represents the solutions for the LHS of $I_1 \wedge \ldots \wedge I_m$ and `soln` represents the solutions for the RHS. Observation 3 says that we can get the constants in the RHS, cons(RHS), using Datalog rules. In addition, Datalog rules can be used to minimize the LHS and RHS to give predicates `min_ins` and `min_soln`. Lemma C.1.10 implies that if each `min_ins` is "contained" in some `min_soln` fact, then the local test condition $I_1 \wedge \ldots \wedge I_m$ is true. This check can also be run as a Datalog program with negation (similar to the minimization process).

C.1.3 Eliminating Negation from the Datalog Rules

We now prove that it is possible to eliminate negation from the Datalog rules that check (T) given t and L. First consider the converse of Lemma C.1.6.

Lemma C.1.11 (Combining Lemma): *Consider the disjunction D:*

D: $a < X < b \ \vee \ X = b \ \vee \ b < X < c$.

where $a < b$ and $b < c$. The set of values of X for which D is true is the same as the set of values for which D' is true, where D' is:

D': $a < X < c$.

□

Proof: Simple to observe. ∎

Lemma C.1.12 *Consider the disjunction D:*

D: $(a < X < b \ \vee \ c < X < d) \ \wedge \ b > c$.

where $a < b$ and $c < d$. The set of values of X for which D is true is the same as the set of values for which D' is true, where D' is:

D': $a < X < d$.

□

Lemma C.1.13 *Let* $\texttt{soln}(a,b)$*,* $\texttt{soln}(b, NULL)$*,* $\texttt{soln}(b,c)$ *represent the solutions to sentences* $a < X < b$*,* $X = b$*,* $b < X < c$ *respectively. The solution for the sentence* $a < X < b \vee X = b \vee b < X < c$*,* $\texttt{soln}(a,c)$*, is computed from the original* soln *facts by the following rule:*

P: $\texttt{soln}(X,Y)$:– $\texttt{soln}(X,Z)$ & $\texttt{soln}(Z, NULL)$ & $\texttt{soln}(Z,Y)$.

□

Proof: The above lemma is the same as Lemma C.1.11, stated in Datalog terms. ∎

Lemma C.1.14 *Let* $\texttt{soln}(a,b)$*, and* $\texttt{soln}(c,d)$ *represent the solutions to sentences* $a < X < b$*, and* $c < X < d$ *respectively and let* $b > c$*. The solution for the sentence* $(a < X < b \ \vee \ c < X < d) \ \wedge \ b > c$*,* $\texttt{soln}(a,d)$ *is computed from the original* soln *facts by the following rule:*

P: $\texttt{soln}(X,Y)$:– $\texttt{soln}(X,Z)$ & $\texttt{soln}(W,Y)$ & $Z > W$.

□

Proof: The above lemma is the same as Lemma C.1.12, stated in Datalog terms. ∎

Lemma C.1.15 *Consider two sentences* D_1 *and* D_2 *of the form* $a < X < b$ *and* $c < X < d$ *respectively. Let arithmetic inequalities* D_1^c *and* D_2^c *be such that* $D_1 \Rightarrow D_1^c$ *and* $D_2 \Rightarrow D_2^c$*. If* $b > c$ *then* D_1 *and* D_2 *can be combined to yield* $D_3 : a < X < d$*. In this case,* D_1^c *and* D_2^c *can also be combined to yield* D_3^c *such that* $D_3 \Rightarrow D_3^c$*.* □

Proof: If $D_1 \Rightarrow D_1^c$ then D_1^c is of the form $e < X < f$ where $e \leq a$ and $f \geq b$. Similarly, if $D_2 \Rightarrow D_2^c$ then D_2^c is of the form $g < X < h$ where $g \leq c$ and $h \geq d$. Given that $f \geq b$, $b > c$, $c \geq g$ we can conclude that $f > g$. Therefore, D_1^c and D_2^c can be combined to yield $e < X < h$ where $e \leq a$ and $h \geq d$ and therefore $a < X < d \Rightarrow e < X < h$, *i.e.*, $D_3 \Rightarrow D_3^c$. ∎

Lemma C.1.16 *Consider three arithmetic inequalities* D_1*,* D_2*, and* D_3 *of the form* $a < X < b$*,* $X = b$*, and* $b < X < d$ *respectively. Let arithmetic inequalities* D_1^c*,* D_2^c*, and* D_3^c *be such that* $D_1 \Rightarrow D_1^c$*,* $D_2 \Rightarrow D_2^c$*, and* $D_3 \Rightarrow D_3^c$*. Given that* D_1*,* D_2*, and* D_3 *can be combined to yield* D_4*, then* D_1^c*,* D_2^c*, and* D_3^c *can also be combined to yield* D_4^c *such that* $D_4 \Rightarrow D_4^c$*.* □

We now discuss the impact of the above lemmas on the Datalog rules that derived and evaluated I.

Consider the tests $I_1, \ldots, I_m$ before minimization. Consider the test I_1. After it has been minimized with respect to $\mathcal{K}$, I_1 is replaced by a conjunction of tests $I_1^1, \ldots, I_1^l$ of the same form as itself except that each I_1^i is minimized wrt $\mathcal{K}$. If I_1^i is true, then there is some sentence $S_i \in \mathbf{min}(\mathsf{RHS}(I_1))$

such that $\mathsf{LHS}(I_1^i) \Rightarrow S_i$. Note, $\mathsf{LHS}(I_1)$ can be obtained by combining $\mathsf{LHS}(I_1^1), \ldots, \mathsf{LHS}(I_1^l)$ using lemmas C.1.12 and C.1.11. Therefore, $S_1, \ldots, S_l$ can also be combined to yield S such that $\mathsf{LHS}(I_1) \Rightarrow S$ (Lemmas C.1.15 and C.1.16).

The consequence of the above inference is that only the RHS of $I_1, \ldots, I_m$ need be minimized. The resulting minimized sentences, of the form $X = c$ and $a < X < b$, can be combined in all possible ways using Lemmas C.1.12 and C.1.11 to generate a set of sentences $\mathcal{S}$. Furthermore, it can be guaranteed that test I_i holds if and only if there is some sentence S in $\mathcal{S}$ such that $\mathsf{LHS}(I_i) \Rightarrow S$. Given that the sentences in the minimized RHS will be recombined in any case, it is not necessary to isolate the minimized set. Therefore, the negation that was introduced for this isolation can be eliminated.

The second negation step was required because minimizing the RHS of the tests $I_1, \ldots, I_m$ could result in an arbitrary number of tests. However, now that the RHS need not be minimized, we know that the number of tests remains m. This number is known at compile time. Therefore, we can generate m tests, one corresponding to each I_i.

In conclusion, for integrity constraints that are IQCs and have only one remote variable, it is possible to write Datalog rules using test (T) such that given an inserted tuple t and a relation L, the Datalog rules derive `OK` if and only if the test (T) is true with the given input. In general, we could consider a set of inserted tuples instead of just one tuple.

C.1.4 Multiple Variables

Until now we considered constraints where test (T) involved a single variable. However, in general the remote inaccessible relation(s) could have multiple attributes, resulting in multiple variables in T. This section shows that the results and techniques of the previous section extend to the n-variable case too.

In this section we consider IQCs, *i.e.*, $I(\mathsf{Red}(\mu, l, C))$ is an independently constrained sentence. As before, the transformation of Observation 1 can be carried out on a sentence that uses $<, >, \leq, \geq, =, \neq$ to yield a disjunction of sentences Θ that use $<, >, =$. Consider one disjunct $D \in \Theta$. Let D involve n variables $\{X_1, \ldots, X_n\}$, each of which is independently constrained and participates in $>, <, =$ comparisons with constants and parameters. Consider variable X_i. The conjunction of the inequalities involving X_i is of the form $\mathcal{V}$ (Page 112). Now, let's order the n variables in some arbitrary total order. The solutions to the variables in Θ can therefore be represented as tuples of a Datalog predicate, say `soln`, that has $2n$ attributes; 2 for each variable. Note, each $D \in \Theta$ may not have all n variables. In fact some D may not have any variables, just constants and parameters. However, `soln` is defined to have the same number of attributes for each D. Attribute $2i-1$ $(2i)$ represents the lower (upper) limit of the solution space for the i^{th} variable. If D does not have an arithmetic inequality involving the i^{th} variable X_i, then the upper limit for X_i is ∞ and the lower limit is $-\infty$. As before, if the upper limit is NULL, then the solution for the attribute is the lower limit.

Using a Lemma similar to Lemma C.1.5, we can define a `soln` fact representing the solution for every sentence $D \in \Theta$. The solution for Θ is a union of the `soln` facts for the disjuncts in Θ. We now define a variant of the splitting-lemma.

Lemma C.1.17 (splitting-lemma): *Consider the sentence*

D: $a < X < c \quad \wedge \quad C.$

where $a < c$ and C does not involve the variable X. Let constant b be such that $a < b < c$. X satisfies $a < X < c \quad \wedge \quad C$ if and only if X satisfies:

D': $(a < X < b \quad \wedge \quad C) \qquad \vee \qquad (X = b \quad \wedge \quad C) \qquad \vee \qquad (b < X < c \quad \wedge \quad C).$

$\square$

In particular, the above lemma implies that a statement of type D can be split up into a disjunction of 3 sentences by using one of the variables in D as the splitting-variable. The corresponding Datalog rules need to be defined for splitting using every possible variable. Therefore, we need $3n$ Datalog rules for a `soln` predicate that represents the solutions to n variables. The rules for the first variable are as follows:

$\texttt{soln}(X_1^l, Y, \ldots, X_n^l, X_n^h) \text{ :- } \texttt{soln}(X_1^l, X_1^h, \ldots, X_n^l, X_n^h) \;\&\; \texttt{constant}(Y) \;\&\; X_1^l < Y < X_1^h.$
$\texttt{soln}(Y, \mathit{NULL}, \ldots, X_n^l, X_n^h) \text{ :- } \texttt{soln}(X_1^l, X_1^h, \ldots, X_n^l, X_n^h) \;\&\; \texttt{constant}(Y) \;\&\; X_1^l < Y < X_1^h.$
$\texttt{soln}(Y, X_1^h, \ldots, X_n^l, X_n^h) \text{ :- } \texttt{soln}(X_1^l, X_1^h, \ldots, X_n^l, X_n^h) \;\&\; \texttt{constant}(Y) \;\&\; X_1^l < Y < X_1^h.$

We now need $2n$ rules to compute the constants involved in set Θ; the i^{th} rule captures the constants that appear in the i^{th} position of `soln` (in the spirit of Observation 5).

Sentences in Θ are minimized by splitting along every variable. That is, one variable is used to split sentences in Θ at one time as presrcibed by the modified splitting-lemma (Lemma C.1.17). The minimization of a set Θ, $\texttt{min}(\Theta)$, is a disjunction of sentences where no sentence $D \in \Theta$ can be split along any variable using a constant from $\texttt{cons}(\Theta)$. Datalog rules for defining the solutions for $\texttt{min}(\Theta)$ are similar to the rules in the 1-variable case.

The index for an n-variable arithmetic inequality of the form D is also redefined. Let the arbitrary total order of the n variables be $X_1, \ldots, X_n$.

Definition C.1.4 (index) Consider a 1-variable sentence B of the form $X = c$ or $a < X < b$. The index of B is redefined as follows:

$$\text{index } B = \begin{cases} (c, NULL, =) & \text{if } B \text{ is } X = c \\ (a, b, <) & \text{if } B \text{ is } a < X < b \end{cases}$$

Ordering between pairs of distinct indexes is defined as follows:

$$(i_1, i_2, op_i) <_x (j_1, j_2, op_j) \text{ if } \begin{cases} i_1 < j_1 \text{ or} \\ i_1 = j_1 \text{ and } op_i \text{ is } = \text{ and } op_j \text{ is } < \end{cases}$$

If two indexes are identical, they are said to be equal to each other and satisfy the $=_x$ relationship. That is:

$$(i_1, i_2, op_i) = (j_1, j_2, op_j) \text{ if } \begin{cases} i_1 = j_1 \text{ and } i_2 = j_2 \text{ and } op_i \text{ is } < \text{ and } op_j \text{ is } < \text{ or} \\ i_1 = j_1 \text{ and } i_2 = j_2 \text{ and } op_i \text{ is } = \text{ and } op_j \text{ is } = \end{cases}$$

In all other cases, the relationship between indexes is undefined. □

Definition C.1.5 (Composite Index (Cindex)) The composite index is defined for an inequality of the form D. Cindex(D) is a list of n indexes, where the i^{th} index corresponds to the index for the inequality involving the i^{th} variable in D. The i^{th} index within Cindex(D) is referred to as Cindex(D)(i).

Cindex(D_1) is $<_x$ Cindex(D_2) if $\exists k$ such that Cindex(D_1)(k) $<_x$ Cindex(D_2)(k) and $\forall 1 \leq i \leq k-1$ Cindex(D_1)(i) $=_x$ Cindex(D_2)(i).

No other relationship is defined between Cindexes. □

The ordering-lemma for the single variable case can be extended to the n-variable case as follows.

Lemma C.1.18 (Ordering-lemma): *Consider a set of inequalities Θ that is minimized with respect to the constants in* cons(Θ). *The disjuncts in Θ can be ordered totally using the $<_x$ relationship between the Cindexes as defined in Definition C.1.5.* □

Proof: We prove the above lemma by contradiction. Let there be two distinct disjuncts D_1 and D_2 that cannot be ordered with respect to each other. There are two cases to consider:

1. $\exists k$ such that $\forall 1 \leq i \leq k-1$ Cindex(D_1)(i) $=_x$ Cindex(D_2)(i) and the relationship between Cindex(D_1)(k) and Cindex(D_2)(k) is not defined.
   The relationship between two indexes (i_1, i_2, op_i) and (j_1, j_2, op_j) is not defined only if $i_1 = j_1$ and $i_2 < j_2$ and $op_i = op_j = "<"$. Note, $i_1 < i_2$ because (i_1, i_2, op_i) corresponds to a consistent inequality $i_1 < X < i_2$. Similarly $j_1 < j_2$. Therefore, we get that $j_1 < i_2 < j_2$. However, i_2 is in cons(Θ) and therefore can be used to split D_2. This would contradict the fact that Θ is minimized with respect to cons(Θ).
2. $\forall 1 \leq i \leq n$ Cindex(D_1)(i) $=_x$ Cindex(D_2)(i)
   This case violates the uniqueness of the members of minimized set of disjuncts.

■

Lemma C.1.19 *Consider the implication:*

I: $A \Rightarrow B_1 \vee \ldots \vee B_k$.

where A and each B_j is of the form D and has been minimized with respect to cons(RHS). *I is true if and only if there exists some i such that $A \Rightarrow B_i$.* □

Proof: By induction on n, the number of variables involved in the implication.

$\boxed{n=1}$ Lemma C.1.10.

$\boxed{n=m+1}$ Say the lemma holds for m variables. Without loss of generality, assume than the new variable is X_1. Each of $A, B_1, \ldots, B_k$ is of the form $S(X_1) \wedge S(X_2) \wedge \ldots \wedge S(X_{m+1})$. Recall that $S(X)$ is either of the form $X=c$ or $a<X<b$. We now do induction on k, the number of disjuncts on the RHS.

$\underline{k=2}$ Therefore, there exists two vectors $V_1 :< x_1, \ldots, x_{m+1} >$ and $V_2 :< y_1, \ldots, y_{m+1} >$ that are solutions to A and V_1 is a solution to B_1 but not to B_2; and V_2 is a solution to B_2 but not to B_1.

Consider two cases:

1. In A, $S(X_1) : X_1 = c$

   In this case, $x_1 = y_1 = c$ because both V_1 and V_2 are solutions to A. This contradicts the induction hypothesis as follows: Consider the implication:

   $A \Rightarrow B_1 \vee B_2.$

   where each of A, B_1, B_2 is of the form $S(X_2) \wedge \ldots \wedge S(X_{m+1})$, *i.e.*, involves m variables. The vector $< x_2, \ldots, x_{m+1} >$ satisfies A and B_1 but not B_2; and $< y_2, \ldots, y_{m+1} >$ satisfies A and B_2 but not B_1.

2. In A, $S(X_1) : a < X_1 < b$

   (a) In B_1, $X_1 = c$

   Given that B_1 has exactly one solution, x_1 must be c. If c is a solution to A, then $a < c < b$. However, $c \in \mathsf{cons}(\mathsf{RHS})$ and A is minimal with respect to $\mathsf{cons}(\mathsf{RHS})$. Therefore, if $a<c<b$ then A can be split with respect to c and the minimality would be violated.

   (b) In B_1, $c<X<d$; $B_2 : e<X<f$

   Without loss of generality, let $d < e$.

$c<x_1<d$ and $a<x_1<b$	x_1 is a solution for A and B_1	Therefore: $a<d$.
$e<y_1<f$ and $a<y_1<b$	y_1 is a solution for A and B_2	Therefore: $e<b$.

   $a<d$ and $d<e$ and $e<b$ implies that $a<d<b$.

   However, d is in $\mathsf{cons}(\mathsf{RHS})$ violating the minimality of A with respect to $\mathsf{cons}(\mathsf{RHS})$.

$\underline{k=l+1}$ The argument is exactly as in Lemma C.1.10. ∎

The combining lemmas and all the subsequent results of the single variable case carry over exactly to the multivariable case.

Appendix D

Proofs for Theorems in Chapter 6

In this appendix we restate and prove some theorems from Section 6.2.

D.1 Insertions

Theorem 6.2.1: *Consider a constraint A defined using a single conjunctive query (possibly using arithmetic and negation) such that the predicate p occurs positively and only once in A and the number of p tuples that can satisfy A is not bounded a priori. Let t represent a tuple inserted into relation P such that t is not irrelevant with respect to A. Let A^u be the constraint A with inserted tuple t incorporated into A. A^u derives* `panic` *with DB if and only if A derives* `panic` *with $DB \cup \{p(t)\}$. Constraint A^u cannot be expressed as a single conjunctive query even if the query uses arithmetic inequalities and negation.*

Proof: (By Contradiction) Let C be a conjunctive query, possibly using arithmetic inequalities and negation, that is equivalent to A^u. We prove a contradiction arises with this assumption. We assume that p has arity three.

<u>Claim 1</u> C derives `panic` for all databases with which A derives `panic`,*i.e.*, $C \supseteq A$. This claim is proved as follows. Let DB be a database such that A derives `panic` with DB. Given that A is monotonic with respect to relation p, A derives `panic` with $DB \cup \{p(t)\}$. Thus, $C \equiv A^u$ also derives `panic` with database DB.

<u>Claim 2</u> C has no negated occurrences of subgoal $p(X, Y, Z)$. We prove this claim by contradiction. Let C have a negated subgoal *not* $p(X, Y, Z)$. Consider a database DB such that A derives `panic` with $DB \cup \{p(t)\}$. Thus, C derives `panic` with DB. Consider all instantiations of C in DB that derive `panic`. Let ρ be one such instantiation, *i.e.*, $\rho(C)$ derives `panic` in DB. If the fact $\rho(p(X, Y, Z))$ is added to DB, to get a new database DB^+, instantiation ρ does not derive `panic` using C in DB^+. Database $DB^+ \cup \{p(t)\}$ derives `panic` using A because A is monotonic in p. Thus, A^u derives `panic` with DB^+ contradicting the claim that $C \equiv A^u$.

<u>Claim 3</u> C has at least one occurrence of the subgoal $p(X, Y, Z)$. We prove this claim by contradiction. Let C have no occurrence of the subgoal $p(X, Y, Z)$. Thus, C is independent of p.

Consider a database DB that satisfies A and is such that if all the p facts are deleted from DB, then DB no longer satisfies A. Such a database exists because p occurs only positively in A and is not a redundant subgoal. Let the database resulting from the deletions be DB^-. In addition, consider an inserted tuple t such that $DB^- \cup \{p(t)\}$ does not satisfy A. Such a tuple has to exist because BD^- is finite and therefore we can choose the constants in t appropriately.

Consider a database DB as defined above. A derives `panic` with DB and hence with $DB \cup \{p(t)\}$. Thus, A^u derives `panic` with DB, implying that C derives `panic` with DB. Now, C is independent of p and thus C derives `panic` even if all p tuples are removed from DB. That is, C derives `panic` with DB^-. However A does not derive `panic` with $DB^- \cup \{p(t)\}$ leading to a contradiction.

The above three claims contradict the initial assumption that C derives `panic` with DB if and only if A derives `panic` with $DB \cup \{p(t)\}$. Consider the following database: $P = \{\}$ and the remaining relations involved in A are such that they satisfy A when $P = \{t\}$. C does not derive `panic` with this database (Claim 4) but A does derive panic with $\mathcal{P} = \{t\}$. ∎

Theorem 6.2.2: *Constraint $I9$, stating that after insertion of "tom" into relation* `insured` *there is no employee in a department that does not appear in* `insured`*, cannot be expressed as a single CQ (over the predicates* `emp` *and* `insured` *denoting their values before insertion) without arithmetic inequalities, even if negation is allowed.*

Proof: (By Contradiction) Recall that constraint $I9$ is:

`panic` :- `emp`(E, D, S) & *not* `insured1`(E).
`insured1`(E) :- `insured`(E).
`insured1`(tom).

Let C be a CQ that does not use arithmetic inequalities and expresses $I9$.

Claim 1 C cannot have an unnegated subgoal with predicate `insured`. We prove this claim by contradiction. Say C does have an unnegated subgoal with predicate `insured`. Then C would not produce `panic` whenever `insured` is empty and `emp` has tuple (tom, a, b), thereby contradicting the claim that C is equivalent to $I9$.

Claim 2 C cannot have a subgoal *not* `insured`(d) where d is any constant such as tom. We prove this claim by contradiction. Say C does have a subgoal *not* `insured`(d) for some constant d. In this case, consider the database:

`emp`$(mary, shoe, 50)$, `insured`(d).

where $d \neq mary$. C fails to cause `panic` with the above database, where $I9$ does derive `panic`.

Claim 3 The only `insured` subgoals in C are of the form *not* `insured`(E) for some variable E. Follows from Claims 1 and 2.

Consider the database with tuples

`emp`$(mary, shoe, 50)$, `emp`$(tom, shoe, 50)$.

and no other tuple for either `emp` or `insured`. $I9$ produces `panic` because in the above database there are two employees in `emp` but relation `insured` is empty. Given that C is equivalent to *I9*, C must also derive `panic`. Consider an instantiation of C's variables that satisfies the body of C. If any of the subgoals of the form *not* `insured`(E) instantiates to *not* `insured`$(mary)$, we claim we can replace these by *not* `insured`(tom) and still produce `panic`. Why? If E appears nowhere else, surely this change can be made. If E appears elsewhere, it can only be in another *not* `insured`(E) subgoal, which presents no problem, or in some subgoal of the form `emp`(E, D, S). In the latter case, the instantiation of the subgoal must have been either

1. `emp`$(mary, shoe, 50)$, in which case we can legally instantiate it instead to `emp`$(tom, shoe, 50)$, or

2. *not* `emp`(a, b, c) for some constants a, b, and c, at least one of which (corresponding to variable E) is $mary$. In this case, we can again replace $mary$ by tom, and the negated subgoal will continue to be true.

Now, consider the database with the same two tuples, `emp`$(mary, shoe, 50)$ and `emp`$(tom, shoe, 50)$ for `emp`, but with additional tuple `insured`$(mary)$. Constraint *I9* does not produce `panic` with this database. However, we established in the paragraph above that there is an instantiation of C that does not use $mary$ as an argument of an `insured` subgoal, the same instantiation produces `panic` on this database. Thus, we contradict the assumption that C is an equivalent of *I9*. Therefore, there is no way to express *I9* as a single CQ without arithmetic inequalities. ∎

D.2 Deletions

Theorem 6.2.4: *Consider constraint A^u that states that no employee in* `emp` *is in* `tall` *after "mary" is deleted from* `tall`.

$$
\begin{aligned}
A^u\!:\quad & \texttt{panic} \coloneq \texttt{emp}(E, D, S)\ \&\ \texttt{tall1}(E) \\
& \texttt{tall1}(E) \coloneq \texttt{tall}(E)\ \&\ \ E \neq mary.
\end{aligned}
$$

A^u cannot be expressed by an equivalent Datalog program C that uses neither negation nor arithmetic.

Proof: (By Contradiction) Let C be such an equivalent constraint expressed as a Datalog program that uses neither negation nor arithmetic. Let $DB1$ be the following database that causes A^u, and hence C, to derive `panic`:

$$\texttt{emp} = \{(john, d, s)\},\quad \texttt{tall} = \{(john)\}.$$

Given that C uses neither negation nor arithmetic, we can infer that C produces a derivation tree for `panic` such that the tree uses only equijoins. We claim base facts `emp`$(john, d, s)$ and `tall`$(john)$ appear in the tree (one or more times). For if one of the base facts does not appear in the tree, say `tall`$(john)$, then we can contradict the claim that C is equivalent to A^u by using an empty `tall` relation in $DB1$.

Equijoins are preserved if all the participating arguments are transformed by a 1-1 function. Thus replacing "*john*" in `emp` and `tall` by "*mary*" will not change the derivation tree for C. Thus C will still derive `panic` while A^u will not, thereby violating the equivalence claim. ∎

Theorem 6.2.5: *Constraint I8, after deleting tuple "john" from relation* `insured`, *cannot be expressed as a single CQ constraint that uses negation.*

Proof: Let A^u represent the rewritten version of constraint $I8$ after the deletion:

A^u: `panic` :- `emp`(E, D, S) & *not* `insured1`(E)
`insured1`(E) :- `insured`(E) & $E \neq john$.

We argue that A^u cannot be rewritten as a single conjunctive query C that may use negation and/or arithmetic. We prove the theorem by contradiction.

Claim 1 $C \supseteq I8$. Observe that $A^u \supseteq I8$, and C is equivalent to A^u.

Claim 2 C has no positive occurrences of subgoal `insured`(E). We prove this claim by contradiction. If there is a subgoal `insured`(E) then a database DB such that A^u derives `panic` in DB. Given that A^u is anti-monotonic in `insured`, we can delete all `insured` facts from DB and still have A^u derive `panic`. However, then the positive subgoal in C would be false, thereby contradicting the equivalence of C and A^u.

Claim 3 C has no occurrence of the subgoal *not* `insured`(a), where a is some constant. We prove this claim by contradiction. If there is such an occurrence, then consider the database:

`emp`$(mary, shoe, 50)$, `insured`(a).

A^u derives `panic` with the above database, but C will not. Thereby contradicting the equivalence of C and A^u. Similarly, we can argue that C has no occurrence of the subgoals `insured`(E) & $E \neq a$.

Claim 4 C has at least one occurrence of the subgoal *not* `insured`(E). We prove this claim by contradiction. Let C have no occurrence of the subgoal *not* `insured`(E), *i.e.*, C is independent of relation `insured`. Now consider a database with which A^u, and thus C, derives `panic`. Let the set of constants in this database be S. To this database add the fact `insured`(a) for every constant $a \in S$. Thus, A^u will no longer derive `panic` but C will, thereby contradicting the equivalence of C and A^u.

The above four claims contradict the initial assumption that C is equivalent to A^u. Consider the database `emp`$(tom, shoe, 50)$ and `insured`(tom). A^u derives `panic` from this database, whereas C will not. ∎

Bibliography

[AHU83] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms.* Addison-Wesley, 1983.

[ABW88] Krzysztof R. Apt and Howard A. Blair and Adrian Walker. *Towards a Theory of Declarative Knowledge.* In *Foundations of Deductive Databases and Logic Programming.* Editor J. Minker, 1988 Morgan Kaufmann.

[AH88] Serge Abiteboul and Richard Hull. Data Functions, Datalog, and Negation. In *Proceedings of ACM SIGMOD 1988 International Conference on Management of Data*, Chicago, IL, June 1988.

[AHU83] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms.* Addison-Wesley, 1983.

[BC79] Peter O. Buneman and Eric K. Clemons. *Efficiently Monitoring Relational Databases.* In *ACM Transactions on Database Systems*, Vol 4, No. 3, 1979, 368-382.

[BCL89] J. A. Blakeley, N. Coburn, and P. Larson. Updating Derived Relations: Detecting Irrelevant and Autonomously Computable Updates. *ACM Transactions on Database Systems*, 14(3):369–400, 1989.

[BLR91] Veronique Benzaken, Christophe Lecluse, and Philippe Richard. Enforcing Integrity Constraints in Database Programming Languages. Technical Report Altair 68-91, Altair, France, 1991.

[BLT86] Jose A. Blakeley and P. Larson and Frank Wm. Tompa. *Efficiently Updating Materialized Views.* In *Proceedings of ACM SIGMOD 1986 International Conference on Management of Data*, Washington D.C., 61-71.

[BBC80] P. A. Bernstein, B. T. Blaustein, and E. M. Clarke. Fast Maintenance of Semantic Integrity Assertions Using Redundant Aggregate Data. In *Proceedings of the Sixth conference on Very Large Data Bases*, pages 126–136, 1980.

[BB82] P. A. Bernstein and B. T. Blaustein. Fast Methods for Testing Quantified Relational Calculus Assertions. In *Proceedings of ACM SIGMOD 1982 International Conference on Management of Data*, pages 39–50, 1982.

[BGM92] D. Barbara and H. Garcia-Molina. The Demarcation Protocol: A Technique for Maintaining Arithmetic Constraints in Distributed Database Systems. In *Extending Database Technology Conference, LNCS 580*, pages 373–397, Vienna, March, 1992.

[Bla81] B. T. Blaustein. *Enforcing Database Assertions: Techniques and Applications.* PhD thesis, Harvard University, Cambridge, Massachusetts, Division of Applied Sciences, 1981.

[BMM92] F. Bry, R. Manthey, and B. Martens. Integrity Verification in Knowledge Bases. In *Logic Programming, LNAI 592 (subseries of LNCS)*, pages 114–139, 1992.

[CG92] S. Ceri and F. Garzotto. Specification and Management of Database Integrity Constraints through Logic Programming. Technical Report 88-025, Dipartimento Di Elettronica - Politecnico Di Milano, 1992.

[CG85] S. Ceri and G. Gottlob. Translating SQL into Relational Algebra: Optimization, Semantics and Equivalence of SQL Queries. *IEEE Transaction of Software Engineering*, 11(4):324–345, April 1985.

[CP84] S. Ceri and G. Pelagatti. *Distributed Databases: Principles and Systems.* McGraw-Hill Book Company, New York, N.Y., 1984.

[CW90] Stefano Ceri and Jennifer Widom. Deriving Production Rules for Constraint Maintenance. In *Proceedings of the Sixteenth International Conference on Very Large Databases (VLDB)*, pages 566–577, 1990.

[CW91] Stefano Ceri and Jennifer Widom. Deriving Production Rules for Incremental View Maintenance. In *Proceedings of the 17th VLDB Conference, Barcelona, Spain*, 1991.

[CW92] Stefano Ceri and Jennifer Widom. Deriving Incremental Production Rules for Deductive Data. IBM RJ 9071, IBM Almaden, 1992.

[C88] A. K. Chandra. Theory of Database Queries. In *Proceedings of ACM SIGMOD 1988 International Conference on Management of Data*, pages 1–9. ACM, 1988.

[CLM81] A. K. Chandra, H. R. Lewise, and J. A. Makowsky. Embedded Implicational Dependencies and Their Inference Problem. In *Proc. Thirteenth Annual ACM Symposium on the Theory of Computing,*. pp. 342–354.

[CM77] A. K. Chandra and P. M. Merlin. Optimal Implementation on Conjunctive Queries in Relational Databases. In *9th ACM Symposium on Theory of Computing*, pages 77–90, ACM, 1977.

[Cha92] S. Chaudhuri and M. Vardi. On the Equivalence of Datalog Programs. In *Proceedings of the Eleventh Symposium on Principles of Database Systems (PODS)*, pages 55–66, San Diego, CA, 1992.

[Cou91] B. Courcelle. Recursive Queries and Context-free Graph Grammars. *Theoretical Computer Science*, 78:217–244, 1991.

[Dav87] E. Davis. Constraint Propagation with Interval Labels. *Artificial Intelligence*, (32):281–331, 1987.

[DAJ91] S. Dar, R. Agrawal, and H. V. Jagadish. Optimization of Generalized Transitive Closure. In *Seventh IEEE International Conference on Data Engineering*, Kobe, Japan, 1991.

[Doy81] Jon Doyle. A Truth Maintenance System. In *Readings In Artificial Intelligence*, pages 496–516. Morgan Kaufmann, 1981.

[DS92] Guozhu Dong and Jianwen Su. Incremental and Decremental Evaluation of Transitive Closure by First-Order Queries. Technical Report TRCS 92-18, University of California, Santa Barbara, 1992.

[DT92] Guozhu Dong and Rodney Topor. Incremental Evaluation of Datalog Queries. In *Proceedings of the International Conference on Database Theory (ICDT)*, 1992.

[Elk90] C. Elkan. Independence of Logic Database Queries and Updates. In *Proceedings of the Ninth Symposium on Principles of Database Systems (PODS)*, pages 154–160, Nashville, TN, 1990. ACM SIGACT-SIGMOD-SIGART.

[F82] R. Fagin. Horn Clauses and Database Dependencies. *Journal of the ACM*, 4(29):952–985, 1982.

[GB94] Ashish Gupta and J. A. Blakeley. Maintaining Views using Materialized Views . *unpublished document*, 1994.

[GKM92] Ashish Gupta, Dinesh Katiyar, and Inderpal Singh Mumick. Counting Solutions to the View Maintenance Problem . In *Workshop on Deductive Databases, JICLSP*, 1992.

[GM92] Ashish Gupta and Inderpal S. Mumick. Magic-Sets Transformation in Non-Recursive Systems . In *Proceedings of the Eleventh Symposium on Principles of Database Systems (PODS)*, 1992.

[GMS93] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining Views Incrementally. In *Proceedings of ACM SIGMOD 1993 International Conference on Management of Data*, pages 157–167.

[GSUW94] Ashish Gupta, Shuky Sagiv, Jeffrey D. Ullman, and Jennifer Widom. Constraint Checking with Partial Information. In *Proceedings of the Thirteenth Symposium on Principles of Database Systems (PODS)*, 1994, pages 45-55.

[GT93] Ashish Gupta and Sanjai Tiwari. Distributed Constraint Management for Collaborative Engineering Databases. In *Proceedings of the Second International Conference on Information and Knowledge Management (CIKM)*, Washington DC, November 1993.

[GT94] Ashish Gupta and Sanjai Tiwari. Constraint Management On Distributed Design Databases. *IEEE Data Engineering Bulletin, Special Issue on Database Constraint Management*, 17(2), June 1994.

[GU92] A. Gupta and J. D. Ullman. Generalizing Conjunctive Query Containment for View Maintenance and Integrity Constraint Checking. In *Workshop on Deductive Databases, JICLSP*, 1992.

[GW93] Ashish Gupta and Jennifer Widom. Local Checking of Global Integrity Constraints . In *Proceedings of ACM SIGMOD 1993 International Conference on Management of Data*, pages 49–59.

[HG95] Venky Harinarayan and Ashish Gupta. Optimization Using Tuple Subsumption. To appear in *ICDT 95*, January 1995.

[HG94] Venky Harinarayan and Ashish Gupta. Generalized Projections: A Powerful Query-Optimization Technique. Unpublished Document, October 1994.

[Hal91] K. Hall. *A Framework for Change Management in a Design Database.* PhD thesis, Stanford University, Department of Computer Science, (report number STAN-CS-91-1379), 1991.

[HD92] John V. Harrison and Suzanne Dietrich. Maintenance of Materialized Views in a Deductive Database: An Update Propagation Approach. In *Workshop on Deductive Databases, JICLSP 1992*, pages 56–65, 1992.

[HKG+94] H. C. Howard, A. M. Keller, A Gupta, Karthik Krishnamurthy, K. L. Law, P. M. Teicholz, Sanjai Tiwari, and J. D. Ullman. Versions, Configurations, and Constraints in CEDB. Working Paper CIFE-31, Center for Integrated Facilities Engineering, Stanford University, April 1994.

[ISO90] ISO_ANSI. ISO-ANSI Working Draft: Database Language SQL2 and SQL3; X3H2; ISO/IEC JTC1/SC21/WG3, 1990.

[JJ91] Manfred Jeusfeld and Matthias Jarke. From Relational to Object-Oriented Integrity Simplification. In *Second International Conference, Deductive and Object-Oriented Databases, LNCS 566*, 1991.

[KKR93] P. Kanellakis, G.M. Kuper, and P.Z. Revesz. Constraint Query Languages. Personal Communication, 1993.

[KL94] Karthik Krishnamurthy and Kincho Law. A Versioning and Configuration Scheme for Collaborative Engineering. In *First Congress on Computing in Civil Engineering, Washington, D.C.* ASCE, 1994.

[Klu88] A. Klug. On Conjunctive Queries Containing Inequalities. *Journal of the ACM*, 1(35):146–160, 1988.

[KSS87] R. Kowalski, F. Sadri, and P. Soper. Integrity Checking in Deductive Databases. In *Proceedings of the Thirteenth International Conference on Very Large Databases (VLDB)*, pages 61–69, 1987.

[Kuc91] V. Kuchenhoff. On the Efficient Computation of the Difference Between Consecutive Database States. In *Second International Conference, Deductive and Object-Oriented Databases, LNCS 566*, pages 478–502, 1991.

[LS93] A.Y. Levy and Y. Sagiv. Queries Independent of Updates. In *Proceedings of the Nineteenth International Conference on Very Large Data Bases*, pages 171–181, Dublin, Ireland, August 1993.

[Lloyd84] John W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1984.

[LST87] J.W. Lloyd, E. A. Sonenberg, and R. W. Topor. Integrity Constraint Checking in Stratified Databases. *Journal of Logic Programming*, 4(4):331–343, 1987.

[LY85] P. A. Larson and H. Z. Yang. Computing Queries from Derived Relations. In *Proceedings of the Eleventh International Conference on Very Large Databases (VLDB)*, pages 259–269, 1985.

[MR89] Michael Maher and Raghu Ramakrishnan. Déjàvu in Fixpoints of Logic Programs. In *NACLP*, October 16-20 1989.

[Mey92] Van Der Meyden. The Complexity of Querying Indefinite Data About Linearly Ordered Domains. In *Proceedings of the Eleventh Symposium on Principles of Database Systems (PODS)*, pages 331–345, San Diego, CA, 1992. ACM.

[MS92] Inderpal Singh Mumick and Oded Shmueli. Aggregation, Computability, and Complete Query Languages. In *Workshop on Structural Complexity, JICLSP 1992*, 1992.

[MPR90] Inderpal Singh Mumick, Hamid Pirahesh, and Raghu Ramakrishnan. The Magic of Duplicates and Aggregates. In *Proceedings of the Sixteenth International Conference on Very Large Databases (VLDB)*, pages 264–277, Brisbane, Australia, August 13-16 1990.

[MS93] Inderpal Singh Mumick and Oded Shmueli. Finiteness Properties of Database Queries. In *Proceedings of the Fourth Australian Database Conference (ADC)*, Brisbane, Australia, February 1-2 1993.

[Mum91] Inderpal Singh Mumick. *Query Optimization in Deductive and Relational Databases.* Ph.D. Thesis, Stanford University, Stanford, CA 94305, USA, 1991.

[Nic82] J. M. Nicolas. Logic for Improving Integrity Checking in Relational Data Bases. *Acta Informatica*, 18(3):227–253, 1982.

[NY83] J. M. Nicolas and Yazdanian. An Outline of BDGEN: A Deductive DBMS. In *Information Processing*, pages 705–717, 1983.

[OV91] T. M. Oszu and P. Valduriez. *Principles of Distributed Database Systems.* Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[Prz90] T. C. Przymusinski. *On the Declarative Semantics of Deductive Databases and Logic Programs.* In *Foundations of Deductive Databases and Logic Programming.* Editor J. Minker, 1988 Morgan Kaufmann.

[QW91] Xiaolei Qian and Gio Wiederhold. *Incremental Recomputation of Active Relational Expressions.* In *TKDE*, 1991.

[RSUV89] Raghu Ramakrishnan, Yehoshua Sagiv, Jeffrey D. Ullman, and Moshe Vardi. Proof-tree Transformation Theorems and Their Applications. In *Proceedings of the Eighth Symposium on Principles of Database Systems (PODS)*, Philadelphia, PA, 1989.

[R93] G. Ramkumar. Personal Communication, November, 1993.

[RS93] Torre Risch and Martin Sköld. Active Rules Based on Object-Oriented Queries. To Appear, *ACM TKDE*, 1993.

[Sag88] Y. Sagiv. Optimizing Datalog Programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 659–698, Washington D.C., 1988. Morgan Kaufmann.

[SY80] Yehoshua Sagiv and Mihalis Yannakakis. Equivalences Among Relational Expressions with the Union and Difference Operators. *Journal of the ACM*, 4(27):633–655, 1980.

[Sar90] Yatin P. Saraiya. Polynomial-time Program Transformations in Deductive Databases. In *Proceedings of the Ninth Symposium on Principles of Database Systems (PODS)*, pages 132–144, 1990.

[SPAM91] Ulf Schreier, Hamid Pirahesh, Rakesh Agrawal, and C. Mohan. $\mathcal{A}$lert: An Architecture for Transforming a Passive DBMS Into an Active DBMS. In *17th VLDB*, pages 469–478, 1991.

[Shm87] Oded Shmueli. Decidability and Expressiveness Aspects of Logic Queries. In *Proceedings of the Sixth Symposium on Principles of Database Systems (PODS)*, pages 237–249, San Diego, CA, March 1987. ACM SIGACT-SIGMOD-SIGART.

[SI84] Oded Shmueli and Alon Itai. Maintenance of Views. In *Proceedings of Annual Meeting, Sigmod Record*, Vol 14, No. 2, 1984, 240-255.

[SSMJ90] D. Stemple, E. Simon, S. Mazumdar, and M. Jarke. Assuring Database Integrity. *Journal of Database Administration*, 1(1):12–26, 1990.

[SKN89] Xian-He Sun, Nabil Kamel, and Lionel M. Ni. Solving Implication Problems in Database Applications. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 185–192, 1989.

[Tiw94] Sanjai Tiwari. *Managing Design Constraints on Distributed Engineering Databases.* PhD thesis, Stanford University, Department of Civil Engineering, 1994.

[TH93] Sanjai Tiwari and H. C. Howard. Constraint Management on Distributed AEC Databases. In *Fifth International Conference on Computing in Civil and Building Engineering*, pages 1147–1154. ASCE, 1993.

[TB88] F. W. Tompa and Jose A. Blakeley. Maintaining Materialized Views Without Accessing Base Data. *Information Systems*, 13(4):393–406, 1988.

[Ull89] J. D. Ullman. *Principles of Database and Knowledge-Base Systems*, Volumes 1 and 2. Computer Science Press, New York, 1989.

[UO92] Toni Urpi and Antoni Olive. A Method for Change Computation in Deductive Databases. In *Proceedings of the Eighteenth International Conference on Very Large Databases (VLDB)*, pages 225–237, Vancouver, British Columbia, 1992.

[VG86] Allen Van Gelder. Negation as Failure Using Tight Derivations for General Logic Programs. In *Third IEEE Symposium on Logic Programming*, 1986. Springer-Verlag.

[VT91] A. V. Gelder and R W. Topor. Safety and Translation of Relational Calculus Queries. *ACM Transactions on Database Systems*, 16(3):235–278, June 1991.

[Wal75] D. Waltz. Understanding Line Drawings of Scenes with Shadows. In *The Psychology of Computer Vision*, New York, 1975. McGraw-Hill.

[WDSY91] Ouri Wolfson and Hasanat M. Dewan and Salvatore J. Stolfo and Yechiam Yemini. Incremental Evaluation of Rules and its Relationship to Parallelism. In *Proceedings of ACM SIGMOD 1991 International Conference on Management of Data*, Denver, CO.

[ZO93] X. Zhang and M. Z. Ozsogoglu. On Efficient Reasoning with Implication Constraints. In *Conference on Deductive and Object Oriented Databases*, pages 236–252, Dec 1993.